

General Programming Concepts: Writing and Debugging Programs



General Programming Concepts: Writing and Debugging Programs

n	ın	m	0

Before using this information and the product it supports, read the information in Appendix B, "Notices," on page 579.

Ninth Edition (August 2004)

This edition applies to AIX 5L Version 5.2 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 Information Development, Department H6DS-905-6C006, 11501 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.

- (C) Copyright Apollo Computer, Inc., 1987. All rights reserved.
- (C) Copyright AT&T, 1984, 1989. All rights reserved.
- (C) Copyright Sun Microsystems, Inc., 1985, 1986, 1987, 1988. All rights reserved.
- (C) Copyright TITN, Inc., 1984, 1989. All rights reserved.
- (C) Copyright Regents of the University of California, 1986, 1987. All rights reserved.

© Copyright International Business Machines Corporation 1997, 2004. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About This Book	 		 		x
Who Should Use This Book	 		 		x
Highlighting					
Case-Sensitivity in AIX	 		 		X
ISO 9000	 		 		x
Related Publications	 		 		X
Chapter 1. Tools and Utilities					1
Entering a Program into the System					
Checking a Program					
Compiling and Linking a Program					
Subroutines					
Shell Commands					
Related Information					
Chapter 2. Curses Library					5
Terminology	 				5
Naming Conventions.	 		 •		5
Structure of a Curses Program					
Return Values					
Initializing Curses					
Windows in the Curses Environment					
Manipulating Window Data with Curses					
Controlling the Cursor with Curses					
Manipulating Characters with Curses	 	•	 	•	11
Understanding Terminals with Curses	 	•	 	•	18
Working with Color	 	•	 	•	24
Manipulating Video Attributes					
Manipulating Soft Labels					
Curses Compatibility					
List of Additional Curses Subroutines	 	•	 	٠	27
Chapter 3. Debugging Programs	 		 		29
adb Debug Program Overview	 		 		29
Getting Started with the adb Debug Program	 		 		29
Controlling Program Execution					
Using adb Expressions	 		 		35
Customizing the adb Debug Program	 		 		37
Computing Numbers and Displaying Text					
Displaying and Manipulating the Source File with the adb Program					
adb Debug Program Reference Information					
Example adb Program: adbsamp					
Example adb Program: adbsamp2					
Example adb Program: adbsamp3					
Example of Directory and i-node Dumps in adb Debugging					
Example of Data Formatting in adb Debugging					
Example of Tracing Multiple Functions in adb Debugging					
dbx Symbolic Debug Program Overview					
Using the dbx Debug Program					
Displaying and Manipulating the Source File with the dbx debug Progra					
Examining Program Data					
Debugging at the Machine Level with dbx					
Customizing the dbx Debugging Environment					
Castonianing the day bedagging Environment	 	•	 	•	

Developing for the dbx Plug-in Framew List of dbx Subcommands																						. 94
Related Information		•	•	•	٠	•		•	•	•	•	•	•	٠	•	•	•	•	•	•	•	. 97
Chapter 4. Error-Logging Overview .																						. 99
Error-Logging Facility																						. 99
Managing Error Logging																						
Error Notification																						
Error Logging Tasks																						
Error Logging and Alerts																						
Error Logging Controls																						
																						44-
Chapter 5. File Systems and Logical	volu	me	S	•		٠	٠	٠	•	٠	٠	•	٠	٠	•	٠	٠	٠	٠	٠	٠	. 11/
File Types																						
Working With JFS Directories																						
Working with JFS2 Directories																						
Working with JFS i-nodes																						
Working with JFS2 i-nodes																						. 124
Allocating JFS File Space																						
Allocating JFS2 File Space																						
JFS File System Layout																						
JFS2 File System Layout																						
Writing Programs That Access Large Fi	ies.	•	•	•		•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	. 100
Linking for Programmers		٠	•	•		٠	٠	٠	٠	٠	٠	٠		•	•	•	•	•	•	٠	•	. 140
Using File Descriptors																						
Creating and Removing Files																						. 145
Working with File I/O																						
File Status																						. 153
File Accessibility																						
Creating New File System Types																						
Logical Volume Programming																						
Related Information																						
rielated information		•	•	•		•		•	•	•	•	•	•	•	•	•	•	•	•	•	•	. 150
Chapter 6. Floating-Point Exceptions																						. 161
Floating-Point Exception Subroutines.																						. 161
Floating-Point Trap Handler Operation																						
Related Information																						
							-			-		-		-		-	-	-			-	
Chapter 7. Input and Output Handling	g .																					. 171
Low-Level I/O Interfaces																						. 171
Stream I/O Interfaces																						. 172
Terminal I/O Interfaces																						
Asynchronous I/O Interfaces																						
Chapter 8. Large Program Support																						175
Understanding the Large Address-Space																						
Understanding the Very Large Address-																						
Enabling the Large and Very Large Add																						
Executing Programs with Large Data A																						
Related Information		٠	•			٠	٠	٠	٠	•							•	٠				. 178
Chapter 9. Programming on Multipro	cess	or :	Sys	ste	ms																	. 179
Identifying Processors																						
Controlling Processor Use																						
Using Dynamic Processor Deallocation											-	-										183

Creating Locking Services.																					. 18	87
Related Information																					. 18	89
Chapter 10. Multi-Threade	d Progran	nmina												_							. 19	91
Understanding Threads and																						
Thread-Safe and Threaded																						
Creating Threads																						
Terminating Threads																						
Synchronization Overview .																						
Using Mutexes																						
Using Condition Variables .																						
Using Read-Write Locks .																						
Joining Threads																						
Scheduling Threads																						
Contention Scope and Cond																						
Synchronization Scheduling																						
One-Time Initializations																						
Thread-Specific Data																						
Creating Complex Synchror																						
Signal Management																						
Process Duplication and Te																						
Threads Library Options .																						
Writing Reentrant and Threa																						
Developing Multi-Threaded																						
Developing Multi-Threaded Developing Multi-Threaded																						
Developing Multi-Threaded Developing Multi-Threaded																						
Benefits of Threads																						
Related Information																						
rielated information			•		•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	. 2	<i>,</i> ¬
Chapter 11. lex and yacc I	Program I	nform	atio	n																	2	7C
Generating a Lexical Analyz																						
Using the lex Program with																						
Extended Regular Expression																						
Passing Code to the General																						
Defining lex Substitution Str																						
lex Library																						
Actions Taken by the Lexica																						
lex Program Start Condition																						
Creating a Parser with the																						
The yacc Grammar File																						
Using the yacc Grammar Fi																						
yacc Grammar File Declara																						
-																						
yacc Rules																						
yacc Actions																						
yacc Program Error Handlin																						
Parser Operation Generated																						
Using Ambiguous Rules in t																						
Turning on Debug Mode for																						
Example Program for the le																						
Related Information			•		٠	٠	٠			•	•	•			٠		٠	٠	•	•	. 30	J/
Ohamtau 10. waalaa Oassaa	m al																				0.4	^^
Chapter 12. make Comma																						
Creating a Description File																						
Internal Rules for the make																						
Defining and Using Macros	10 0 17000	untion	-110																		.ე.	15
Creating a Target File with t																						

Using the make Command with Source Code Control System (SCCS) Using the make Command with Non-Source Code Control System (SC Understanding How the make Command Uses the Environment Variable	CS) File	es						. 321
Example of a Description File									
Chapter 13. m4 Macro Processor Overview									. 325
Using the m4 Macro Processor									
Creating a User-Defined Macro									
Using a Predefined m4 Macro									
Additional m4 Macros									
Chapter 14. Object Data Manager (ODM)									
ODM Object Classes and Objects									
ODM Descriptors									
ODM Object Searches									
ODM Commands and Subroutines									. 345
ODM Example Code and Output									. 346
Related Information									. 349
Chapter 15. Dynamic Logical Partitioning									
DLPAR-Safe and -Aware Programs									
Processor Bindings									. 354
Integrating the DLPAR Operation into the Application									. 355
Actions Taken by DLPAR Scripts									
Making Kernel Extensions DLPAR-Aware									
Related Information									
Chapter 16. sed Program Information									. 365
Manipulating Strings with sed									
Chapter 17. Shared Libraries and Shared Memory									
Shared Objects and Runtime Linking									
Shared Libraries and Lazy Loading									. 373
Creating a Shared Library									. 375
Program Address Space Overview									
Understanding Memory Mapping									
Inter-Process Communication (IPC) Limits									
Creating a Mapped Data File with the shmat Subroutine									
Creating a Copy-On-Write Mapped Data File with the shmat Subroutin									
Creating a Shared Memory Segment with the shmat Subroutine									
Paging Space Programming Requirements									
List of Memory Manipulation Services									
List of Memory Mapping Services									
Related Information	٠		•	•	•	•	•	 •	. 389
Chapter 18. System Memory Allocation Using the malloc Subsyste									201
Working with the Heap in 32-bit Applications									
Working with the Heap in 64-bit Applications									
Understanding System Allocation Policy									
User-Defined Malloc Replacement									
Debug Malloc Tool									
Malloc Multiheap									
Malloc Buckets									
Malloc Report									
Malloc Trace									
Malloc Log									412

Malloc Disclaim															. 413
Chapter 19. Packaging Software for Installation															. 415
Installation Procedure Requirements															. 415
Package Control Information Requirements															. 416
Format of a Software Package															. 416
Package Partitioning Requirements															. 416
Software Vital Product Data (SWVPD)															. 416
Software Product Packaging Parts															. 417
Package and Fileset Naming Conventions															. 417
Fileset Revision Level Identification															. 419
Contents of a Software Package															. 419
The lpp_name Package Information File															. 420
The liblpp.a Installation Control Library File															
Further Description of Installation Control Files															
Installation Control Files Specifically for Repackaged Production															
Installation Files for Supplemental Disk Subsystems															
Format of Distribution Media															
Tape															
CD-ROM															
Diskette															
The Table of Contents File.															
The installp Processing of Product Packages															
Installation Commands Used During Installation and Update															
motaliation commands osca baring motaliation and opeate		,000	, iii	9.	•	•	•	•	•	•	•	•	•	•	. 110
Chapter 20. Documentation Library Service															451
Language Support															
Writing your HTML Documents															
Making your Documents Printable															
Calling the Documentation Library Service From Your Docu															
Creating Indexes of your Documentation															
Removing Indexes of your Documentation															
•															
Packaging your Application's Documentation	•		•	•	•	•	•	•	•	•	•	•	•	•	. 400
Chapter 21. Source Code Control System (SCCS)															171
Introduction to SCCS															
SCCS Flag and Parameter Conventions															
Creating, Editing, and Updating an SCCS File															
Controlling and Tracking SCCS File Changes															
Detecting and Repairing Damaged SCCS Files															
List of Additional SCCS Commands															
Related Information			٠	٠		•	•	٠	٠		٠	•		•	. 4/6
Observations Francis December and Libert															47-
Chapter 22. Subroutines, Example Programs, and Librar															
128-Bit Long Double Floating-Point Data Type															
List of Character Manipulation Subroutines															
List of Executable Program Creation Subroutines															
List of Files and Directories Subroutines															
List of FORTRAN BLAS Level 1: Vector-Vector Subroutines															
List of FORTRAN BLAS Level 2: Matrix-Vector Subroutines															
List of FORTRAN BLAS Level 3: Matrix-Matrix Subroutines															
List of Numerical Manipulation Subroutines															
List of Long Long Integer Numerical Manipulation Subroutin															
List of 128-Bit Long Double Numerical Manipulation Subrou															
List of Processes Subroutines															
List of Multi-threaded Programming Subroutines															. 488

List of Programmer's Workbench Library Subroutines
List of String Manipulation Subroutines
Programming Example for Manipulating Characters
Searching and Sorting Example Program
List of Operating System Libraries
librs2.a Library 498 Related Information 500 Chapter 23. System Management Interface Tool (SMIT) 50 SMIT Screen Types 50 SMIT Object Classes 50 SMIT Aliases and Fast Paths 50 SMIT Information Command Descriptors 50
Related Information 500 Chapter 23. System Management Interface Tool (SMIT) 500 SMIT Screen Types 500 SMIT Object Classes 504 SMIT Aliases and Fast Paths 507 SMIT Information Command Descriptors 508
Chapter 23. System Management Interface Tool (SMIT) 50° SMIT Screen Types 50° SMIT Object Classes 50° SMIT Aliases and Fast Paths 50° SMIT Information Command Descriptors 50°
SMIT Screen Types
SMIT Screen Types
SMIT Object Classes.
SMIT Aliases and Fast Paths
SMIT Information Command Descriptors
SMIT Command Generation and Execution 510
Adding Tasks to the SMIT Database
Debugging SMIT Database Extensions
Creating SMIT Help Information for a New Task
sm_menu_opt (SMIT Menu) Object Class
sm_name_hdr (SMIT Selector Header) Object Class
sm_cmd_opt (SMIT Dialog/Selector Command Option) Object Class
sm_cmd_hdr (SMIT Dialog/Selector Command Option) Object Class
SMIT Example Program
Related Information
Observatory O.4. Constant December Constraillery
Chapter 24. System Resource Controller
Subsystem Interaction with the SRC
SRC Objects
SRC Communication Types
Programming Subsystem Communication with the SRC
Defining Your Subsystem to the SRC
List of Additional SRC Subroutines
Related Information
Chapter 25. Trace Facility 558
Chapter 25. Trace Facility
The Trace Facility Overview
The Trace Facility Overview 555 Controlling the Trace . 555 Recording Trace Event Data . 556 Generating a Trace Report . 557 Extracting trace data from a dump . 557 Trace Facility Commands . 557 Start the Trace Facility . 567 Related Information . 567 Chapter 26. tty Subsystem . 567 TTY Subsystem Objectives . 567
The Trace Facility Overview 555 Controlling the Trace 555 Recording Trace Event Data 556 Generating a Trace Report 557 Extracting trace data from a dump 557 Trace Facility Commands 557 Start the Trace Facility 560 Related Information 560 Chapter 26. tty Subsystem 560 TTY Subsystem Objectives 560 Line Discipline Module (Idterm) 550
The Trace Facility Overview 555 Controlling the Trace 556 Recording Trace Event Data 556 Generating a Trace Report 557 Extracting trace data from a dump 557 Trace Facility Commands 557 Start the Trace Facility 560 Related Information 560 Chapter 26. tty Subsystem 560 TTY Subsystem Objectives 560 Line Discipline Module (Idterm) 560 Converter Modules 570
The Trace Facility Overview 558 Controlling the Trace 558 Recording Trace Event Data 556 Generating a Trace Report 557 Extracting trace data from a dump 557 Trace Facility Commands 557 Start the Trace Facility 560 Related Information 567 Chapter 26. tty Subsystem 567 TTY Subsystem Objectives 567 Line Discipline Module (Idterm) 567 Converter Modules 570 TTY Drivers 57
The Trace Facility Overview 555 Controlling the Trace 556 Recording Trace Event Data 556 Generating a Trace Report 557 Extracting trace data from a dump 557 Trace Facility Commands 557 Start the Trace Facility 560 Related Information 560 Chapter 26. tty Subsystem 560 TTY Subsystem Objectives 560 Line Discipline Module (Idterm) 560 Converter Modules 570
The Trace Facility Overview 555 Controlling the Trace 555 Recording Trace Event Data 556 Generating a Trace Report 557 Extracting trace data from a dump 557 Trace Facility Commands 557 Start the Trace Facility 560 Related Information 560 Chapter 26. tty Subsystem 560 TTY Subsystem Objectives 560 Line Discipline Module (Idterm) 560 Converter Modules 570 TTY Drivers 572 Related Information 572
The Trace Facility Overview 558 Controlling the Trace 558 Recording Trace Event Data 556 Generating a Trace Report 557 Extracting trace data from a dump 557 Trace Facility Commands 557 Start the Trace Facility 560 Related Information 567 Chapter 26. tty Subsystem 567 TTY Subsystem Objectives 567 Line Discipline Module (Idterm) 567 Converter Modules 570 TTY Drivers 57

Appendix A.	_															_																	- 77
family Real	1- I IM	e C	IOC	K	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	5//
Appendix B.																																	
Trademarks .			•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•		•	•	•	580
Index																																	583

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.

This edition supports the release of AIX 5L Version 5.2 with the 5200-04 Recommended Maintenance package. Any specific references to this maintenance package are indicated as AIX 5.2 with 5200-04.

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 navigating 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.

Case-Sensitivity in AIX

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

ISO 9000

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

Related Publications

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

- · AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts
- AIX 5L Version 5.2 Communications Programming Concepts
- AIX 5L Version 5.2 AIXwindows Programming Guide
- AIX 5L Version 5.2 System Management Guide: Operating System and Devices
- AIX 5L Version 5.2 System Management Guide: Communications and Networks
- AIX 5L Version 5.2 Commands Reference
- Keyboard Technical Reference
- AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 1
- AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2
- Understanding the Diagnostic Subsystem for AIX

Chapter 1. Tools and Utilities

This chapter provides an overview of the tools and utilities that you can use to develop C compiled language programs. Many tools are provided to help you develop C compiled programs. The tools provide help with the following programming tasks:

- · "Entering a Program into the System"
- · "Checking a Program"
- · "Compiling and Linking a Program"
- "Subroutines" on page 2
- "Shell Commands" on page 2

Subroutines and shell commands are provided for use in a C compiled 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

lint

Use the following commands to check the format of a program for consistency and accuracy:

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).

Checks for syntax and data type errors in a C language source program. The **lint** command might check these areas of a program more carefully than the C language compiler does, and displays many

messages that point out possible problems.

Compiling and Linking a Program

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

A *compiler* is a program that reads 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 and determines how to put the finished program into memory. To create this final form of the program, the system does the following:

- 1. If a file contains compiler source code, the compiler translates it into object code.
- 2. If a file contains assembler language, the assembler translates it into object code.
- 3. 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 C⁺⁺, FORTRAN, COBOL,nd Assembler and other compiler languages.

You can write parts of a program in different languages and have one main routine call and start the separate routines to execute. You can also use the compiler program to create object code and link the program.

Correcting Errors in a Program

You can use the following debugging tools that are provided with the base operating system:

- The **dbx** symbolic debug program allows you to debug programs written in C language, C⁺⁺, FORTRAN, COBOL and Assembler languages. For more information, see "dbx Symbolic Debug Program Overview" on page 61.
- The adb debug program provides subcommands you can use to examine, debug, and repair executable binary files and to examine non-ASCII data files. For more information, see "adb Debug Program Overview" on page 29.
- KDB Kernel Debugger and kdb command can help you determine errors in code running in the kernel. You can use this debug program to debug device drivers and kernel extensions.
- The trace facility helps isolate system problems by monitoring selected system events. For more information see Chapter 25, "Trace Facility," on page 555.

When syntax errors or parameter-naming inconsistencies are discovered in a program file, you can use a text editor or string-searching and string-editing programs 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

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

- The make command builds a program from source modules. Because 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.
- The Source Code Control System (SCCS) 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. For more information, see Chapter 21, "Source Code Control System (SCCS)," on page
 471

Subroutines

Subroutines from system libraries handle many complex or repetitive programming situations so that you can concentrate on unique programming situations. See Chapter 22, "Subroutines, Example Programs, and Libraries," on page 477 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.

Related Information

For further information on this topic, see the following:

- "Manipulating Strings with sed" on page 365
- "Generating a Lexical Analyzer with the lex Command" on page 279
- Chapter 12, "make Command," on page 309
- Chapter 22, "Subroutines, Example Programs, and Libraries," on page 477

Chapter 2. Curses Library

The curses library provides a set of functions that enable you to manipulate a terminal's display regardless of the terminal type. The curses library supports color. However, multibyte characters are not supported. All references to characters in the curses documentation refer to single-byte characters. 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 within it.

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 the relevant information it needs to carry out your instructions.

Terminology

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

Term current character current line curscr	Definition The character that the logical cursor is currently on. The line that the logical cursor is currently on. A virtual default window provided by curses. The cursor (current screen) is an internal representation of what currently appears on the terminal's external display. Do not modify the cursor.
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 pad is a window that is not restricted by the size of the screen
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 more than one version. 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 the following prefixes:

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

If a curses subroutine has multiple versions and does not include one of the preceding prefixes, the curses default window stdscr (standard screen) is used. 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 debug program, 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:

- 1. Start curses.
- 2. Check for color support (optional).
- 3. Start color (optional).
- 4. Create one or more windows.
- 5. Manipulate windows.
- 6. Destroy one or more windows.
- 7. Stop curses.

Some steps are optional, so 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 or an error.

Initializing Curses

Use the following commnads to initialize curses:

endwinTerminates the curses subroutine libraries and their data structuresinitscrInitializes the curses subroutine library and its data structures

isendwin Returns TRUE if the **endwin** subroutine has been called without any subsequent calls to the

wrefresh subroutine

newterm Sets up a new terminal

setupterm Sets up the TERMINAL structure for use by curses

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 and then call the **setupterm** subroutine to establish a curses terminal.

If you need to temporarily suspend curses, use a shell escape or subroutine. To resume after a temporary escape, 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 do not want to call the wrefresh subroutine needlessly. To determine if the endwin subroutine was called without any subsequent calls to the wrefresh subroutine, use 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: A pad is a window that is not restricted by the size of the screen. For more information, see "Pads" on page 8

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.

Default Window Structure

Curses provides a virtual default window structure 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. They 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.

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).

The cursor is reserved for internal use by curses. Do not manipulate the cursor.

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, 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 also refreshes the children. 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. Curses returns an error if you try to delete a window before first deleting 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 one time.

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.

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 user-defined windows.

Creating Windows

You can create your own window using the newwin subroutine.

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

You must supply coordinates for the subwindow relative to the terminal's display. The subwindow, created using the **subwin** subroutine, must fit within the bounds of the parent window. Otherwise, a null value is returned.

Pads

Use the following subroutines to create pads:

Creates a 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

Whenever 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 cursor to the contents of the user-defined or stdscr
- · Updates the cursor structure to match the user-defined or stdscr
- · Redraws the portion of the physical display that changed

Use the following subroutines to refresh windows:

refresh, or wrefresh wnoutrefresh or doupdate Updates the terminal and cursor to reflect changes made to a window. 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.

The prefresh subroutine updates both the current screen and the physical display, while the pnoutrefresh subroutine updates cursor 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. You can 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 cursor 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, by using the following subroutines:

Draws a box in or around a window box

copywin Provides more precise control over the **overlay** and **overwrite** subroutine

garbagedlines Indicates to curses that a screen line is discarded and should be thrown away before

having anything written over

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

To use the **overlay** and **overwrite** subroutines, the two windows must overlap. 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.

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

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

Support for Filters

The **filter** subroutine is provided for curses applications that are filters. This subroutine causes curses to operate as if the stdscr was only a single line. 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

The following types of cursors exist in the curses library:

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 logical cursor in a window. The following subroutines are provided for controlling the cursor:

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

move Moves the logical cursor associated with the stdscr

mvcur Moves the physical cursor

setsyxSets the virtual screen cursor to the specified coordinatewmoveMoves the logical cursor associated with a user-defined window

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 describes how you can add, remove, or change characters that appear in a curses window.

Character Size

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 produces 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 the scrollok subroutine 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:

addch macro Adds a character to the stdscr

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

waddch subroutine Adds a character 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. For more information, see "winch Subroutines" on page 18.

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

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

The typical effects of a nonspacing character associated with a spacing character called c, are as follows:

- The nonspacing character may modify the appearance of c. (For instance, there may be nonspacing characters that add diacritical marks to characters. However, there may also be spacing characters with built-in diacritical marks.)
- The nonspacing 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 nonspacing 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 nonspacing characters associated with it. A spacing complex character is a complex character that includes one spacing character and any nonspacing 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 nonspacing 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 nonspacing 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 the setchar subroutine, and its contents can be extracted using the **getchar** subroutine. The behavior of functions that take a cchar t value that was not initialized in this way are obtained from a curses function that has a cchar t output argument.

Special Characters

Some functions process special characters. 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 column toward

the start of the current line, and any characters after the Backspace are added or inserted

starting there.

Carriage return

newline

Tab

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. 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 wcIrtoeol subroutine), 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 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 uppercase letter if it is alphabetic) and write this string to the window in place of the control character. The functions 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 is 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	l	vertical line
ACS_PLUS	+	plus
ACS_S1	-	scan line 1
ACS_S9	_	scan line 9
ACS_DIAMOND	+	diamond
ACS_CKBOARD	:	checkerboard (stipple)
ACS_DEGREE	,	degree symbol
ACS_PLMINUS	#	plus/minus
ACS_BULLET	0	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:

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

waddstr subroutine Adds a character string to a user-defined window

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:

insch macro Inserts a character in the stdscr

Moves the logical cursor to a specified location in the stdscr before inserting a mvinsch macro

mvwinsch macro Moves the logical cursor to a specified location in a user-defined window before

inserting a character

winsch subroutine Inserts a character in a user-defined window

winsertIn Subroutines

The winsertIn subroutines insert a blank line above the current line in a window. The insertIn subroutine inserts a line in the stdscr. The bottom line of the window is lost. The winsertIn 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 **printw** family is made up of the following:

mvprintw macro Moves the logical cursor to a specified location in the stdscr before replacing any

characters

Moves the logical cursor to a specified location in a user-defined window before mvwprintw macro

replacing any characters

printw macro Replaces a series of characters in the stdscr

Replaces a series of characters in a user-defined window wprintw subroutine

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

Use the following subroutines to enable 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

Sets a software scrolling region within a window setscrreg or

wsetscrreg

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 newline 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 remains 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 **setscrreq** macro to define a scrolling region in the stdscr. Otherwise, you use the **wsetscrreq** subroutine to define scrolling regions in user-defined windows.

Note: Unlike the idlok subroutine, the setscrreg subroutines have no bearing on the use of the physical scrolling region capability that the terminal may 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

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

Use the following subroutines to clear the screen:

clear, or wclear Clears the screen and sets a clear flag for the next refresh.

Determines whether curses clears a window on the next call to the refresh or clearok

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

The cirtoeol macro operates in the stdscr, while the wcirtoeol subroutine performs the same action within a user-defined window.

wclrtobot Subroutines

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

wdelch Subroutines

Use the following subroutines to delete characters from the screen:

delch macro Deletes the current character from the stdscr

mvdelch macro Moves the logical cursor before deleting a character from the stdscr

Moves the logical cursor before deleting a character from a user-defined window mvwdelch macro

Deletes the current character in a user-defined window wdelch subroutine

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

The deleteIn subroutines delete the current line and move all lines below the current line up one line. This action 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. For more information, see wgetch Subroutine, in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2.

The **waetch** subroutine family is made up of the following:

getch macro Gets a character from the stdscr

mvgetch macro Moves the cursor before getting a character from the stdscr

Moves the cursor before getting a character from a user-defined window mvwgetch macro

wgetch subroutine Gets 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.

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 waetch subroutines in each type of terminal mode:

DELAY mode 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 newline 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,

mode the character is also echoed to the window.

NODELAY mode Returns a value of ERR if there is no input waiting.

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. For a list of function keys, see getch, mygetch, mywgetch, or wgetch Subroutine, in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2.

Getting Function Kevs: If your program enables the keyboard with the kevpad subroutine, and the user presses a function key, the token for that function key is returned instead of raw characters. The /usr/include/curses.h file defines the possible function keys. 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 Esc key there is a delay before

the escape is returned to the program. Avoid using the Esc 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 **ESCDELAY** environment variable.

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 waetch 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 winch subroutines:

inch macro Gets the current character from the stdscr

mvinch macro Moves the logical cursor before calling the **inch** subroutine on the stdscr

Moves the logical cursor before calling the winch subroutine in the user-defined window mvwinch macro

winch subroutine Gets the current character from a 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:

Moves the logical cursor before scanning the stdscr mvscanw macro

mvwscanw macro Moves the logical cursor in the user-defined window before scanning

scanw macro Scans the stdscr

wscanw subroutine Scans a user-defined window

The vwscanw subroutine scans a window using a variable argument list. For information about manipulating variable argument lists, see the varargs macros in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2.

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. If your application sends output to more than one terminal, use the newterm subroutine. Call the newterm subroutine for each terminal. 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:

has ic Determines whether a terminal has the insert-character capability has il Determines whether a terminal has the insert-line capability

Returns the verbose name of the terminal longname

The longname subroutine returns a pointer to a static area containing a verbose description of the current terminal. This static area is defined only after a call to the initscr or newterm subroutine. If you intend to use the longname subroutine with multiple terminals, 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. The following mutually-exclusive curses modes let the application control the effect of the input characters:

Cooked Mode

This mode achieves normal line-at-a-time processing with all special characters handled outside the application, achieving the same effect as canonical-mode input processing. The state of the ISIG and IXON flags is 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, regardless of the width of the character.

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 noncanonical mode, Case B input processing (with MIN set to 1 and ICRNL cleared). The state of the ISIG and IXON flags is 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 noncanonical mode, Case C input processing (with TIME set to the value specified by the application). The state of the ISIG and IXON flags is not changed upon entering this mode.

Raw Mode

Raw mode gives maximum control to the application over terminal input. The application sees each character as it is typed. This achieves the same effect as noncanonical 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 the **initscr** or **newterm** subroutines to initialize curses and restores these settings when the endwin subroutine 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

The following 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 mode

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 Mode 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 the addch subroutine was 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 display on the screen at the point where the cursor is positioned.

The following are a part of the echo processing family of subroutines:

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

Returns ERR if no input was typed after blocking for a specified amount of halfdelay

time

nl or nonl

raw or noraw

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 Places the terminal into or out of 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 RAW 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 delay_output subroutine sets the output delay to the specified number of milliseconds. Do not use this subroutine excessively because it uses padding characters instead of a processor pause.

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 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.

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

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. This information is usually 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 must 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 parameterized strings through the tparm subroutine to instantiate them. Use the tputs or putp subroutine to print all terminfo strings and the output of the tparm subroutine.

Provides a shortcut to the tputs subroutine putp 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, 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:

ripoffline Strips a single line from the stdscr

scr dump Dumps the contents of the virtual screen to a specified file Initializes the curses data structures from a specified file scr_init

scr restore Restores the virtual screen to the contents of a previously dumped file

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.

tgetflagReturns the boolean entry for a termcap identifier.tgetnumReturns the numeric entry for a termcap identifier.tgetstrReturns 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.

Note: Although it may be possible to send arbitrarily 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 that makes use of single-character input.

Output

The curses interface can be used 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, note the following points:

- 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 the point where 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, 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.

Checks to see if the terminal supports colors and changing of the color definition can change color

has colors Checks that the terminal supports colors

Initializes the eight basic colors and two global variables, COLORS and start_color

COLOR_PAIRS

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

Manipulating Video Attributes

Your program can manipulate a number of video attributes.

Video Attributes, Bit Masks, and Default Colors

Curses enables you to control the following attributes:

A ALTCHARSET Alternate character set.

A BLINK Blinking.

A BOLD Extra bright or bold.

A DIM Half-bright. A_NORMAL Normal attributes. A_REVERSE Reverse video.

Terminal's best highlighting mode. **A_STANDOUT**

A UNDERLINE Underline.

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_ATTRIBUTES Extracts attributes A_CHARTEXT Extracts a character

A_COLOR Extracts color-pair field information A NORMAL Turns all video attributes off

The following 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 the following default colors:

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 other attributes. 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 the following 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 follows:

```
wattrset(win, COLOR PAIR(1));
```

If you then write the string Let's add Color to the terminal, the string displays 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 the following alarm subroutines to signal the user:

beep Sounds an audible alarm on the terminal 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 19 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_clearClears soft labels from the screen.slk_initInitializes soft function key labels.

slk_label Returns the current label.

slk_noutrefresh Refreshes soft labels. This subroutine is functionally equivalent to the wnoutrefresh

subroutine.

slk_refresh Refreshes 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 **LINES** environment variable 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 to left-justify, right-justify, or center the string on the label. 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.

Curses Compatibility

The following compatibility issues need to be considered:

- In AIX 4.3, curses is not compatible with AT&T System V Release 3.2 curses.
- Applications compiled, rebound, or relinked may need source code changes for compatibility with the AIX Version 4 of curses. The curses library does not have or use AIX extended curses functions.
- Applications requiring multibyte support might still compile and link with extended curses. Use of the
 extended curses library is, however, discouraged except for applications that require multibyte support.

List of Additional Curses Subroutines

The following sections describe additional curses subroutines:

- · "Manipulating Windows"
- · "Manipulating Characters"
- "Manipulating Terminals" on page 28
- "Manipulating Color" on page 28
- "Miscellaneous Utilities" on page 28

Manipulating Windows

Use the following subroutines to manipulate windows:

scr_dumpwrites the current contents of the virtual screen to the specified filescr_initUses 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

Use the following subroutines to manipulate 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.

insertIn or **winsertIn** 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

Use the following subroutines to manipulate terminals:

def_prog_modeIdentifies the current terminal mode as the in-curses modedef_shell_modeSaves 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

Use the following subroutines to manipulate colors:

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

The following miscellaneous utilities are available:

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++, 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.

The following articles provide information on the **adb** and **dbx** debug programs:

- · "adb Debug Program Overview"
- "dbx Symbolic Debug Program Overview" on page 61

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:

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 35.

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 -qList 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 41). 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:

```
d = 236
e = 268435512
m = 264
                  l r15,32(r1)
             .f+8
breakpoint
```

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:

```
[Aaddress] [,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)(dash)Bitwise complementationInteger 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
=1n"====== adb Variables ======"
$v
=1n"====== C Stack Backtrace ======"
$C
=1n"====== C External Variables ======="
$e
=1n"====== Registers ======"
$r
0$s
=1n"====== 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
        e2 = 0 f2 = 0
b2 = 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
rl 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
10000000: 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: cd0e 60 6520 a424 a432 c84e 3f 8df0 4e
100000e0: cd0e 60 6520 a424 a432 c84e 3f 8df0
100000f0: 31 cd0e 64 c820 14 8df0 2b cd0e
```

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 implementation of the POWER architecture.
pwrx	Specifies the instruction set and mnemonics for the POWER2 implementation of the POWER family.
ррс	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 PowerPC
	mnemonics.

The \$n subcommand accepts the following values:

pwr	Specifies the instruction set and mnemonics for the POWER implementation of the POWER architecture.
ррс	Specifies the mnemonics for the PowerPC architecture.

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

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

This section describes how to 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 addressAddress [, Count] = FormatDisplay instructionAddress [, Count] ? FormatDisplay value of variableAddress [, 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

DisplacementsOther 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:

```
<br/><b+0x2000=D<br/>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 a	Format The current symbolic address
b	One byte in octal (displays data associated with instructions, or the high or low byte of a register)
С	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
0	Halfword in octal (short variables)
0	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:

main=o 1560 main=O 4000001560 main=d 880 main=D 536871792 main=x 370 main=X 20000370	Command	Response
main=d 880 main=D 536871792 main=x 370	main=o	1560
main=D 536871792 main=x 370	main=O	4000001560
main=x 370	main=d	880
	main=D	536871792
main=X 20000370	main=x	370
	main=X	20000370
main=u 880	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 \mathbf{a} , \mathbf{n} , \mathbf{r} , and \mathbf{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 48). 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:

```
10000470
.0?m
       10000100
       100
               100
/m
$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 ?.

```
20000270
                           20000374
/*m
        200
                  200
                           200
$m
 [0] : ?map : 'adbsamp3'
b1 = 0 \times 10000100, 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 48) subcommands.

Locating Values in a File

Locate specific values in a file by using the I and L subcommands. See ("adb Debug Program Reference Information" on page 48). The subcommands have the form:

?I Value

OR

/I Value

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

The **?I** 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:

```
?1 .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 \mathbf{w} and \mathbf{W} subcommands. See ("adb Debug Program Reference Information" on page 48). 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":

```
?1 .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
е	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 48). 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**:

\$٧

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:

```
<br/>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 52) program:

```
0.
```

Displaying External Variables

Use the **\$e** ("adb Debug Program Reference Information" on page 48) 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 53) 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
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" on page 48). 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 = 0x00000000, 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.

Increases the value of . (period) by the current increment. + (plus) ^ (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

Specifies the ASCII value of up to 4 characters. A \ (backslash) can be used to `Cccc'

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.

.Svmbol 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.

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~Expression#ExpressionPerforms bit-wise complement.Performs logical negation.

Binary Operators

Expression1+Expression2Performs integer addition.Expression1-Expression2Performs integer subtraction.Expression1*Expression2Performs integer multiplication.Expression1&Expression2Performs integer division.Expression1&Expression2Performs bit-wise conjunction.Expression1|Expression2Performs 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.

IFormat 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.

С 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. Prints 2 bytes in octal. 0 0 Prints 4 bytes in octal.

Prints the addressed value in symbolic form using the same rules for symbol lookup as the a p

Prints 2 bytes in the current radix, unsigned. q Q Prints 4 unsigned bytes in the current radix.

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.

Prints as an unsigned decimal number. u

U Prints a long unsigned decimal. X Prints 2 bytes in hexadecimal. Χ Prints 4 bytes in hexadecimal. Υ 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.

Repeats the previous command incremented with a Count of 1. newline

> 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

[?/]IValue Mask

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 I, 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.

Assigns a . (period) to the variable or register specified by the Name parameter.

>Name 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.</p>
- << 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.
- 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.

- I Changes the default directory as specified by the -I flag to the *Name* parameter value.
- **m** Prints the address map.

n mnem_set

Selects the mnemonics to be used for disassembly.

- 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.
- 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:

bCommand

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.
- 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.
- sSignal 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

The following sample program is used in this example:

```
/* 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

The following sample program is used in this example:

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
           .estate
       27
       28
          adbsamp
       29
           adbsamp.c
       30
           calc.lex
       31
           calc.yacc
       32 cbtest
       68 .profile
       66 .profile.bak
       46 adbsamp2.c
       52 adbsamp2
           adbsamp.s
       35
       34
           adbsamp2.s
           forktst1.c
       48
       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
           st r0, 0x8(r1)
.main+4:
.main+8:
             sturs, (r1)
            li l r4, 0x1
.main+c:
.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
                 1 r0, 0x40(r1)
                 ai r1, r1, 0x38
.main+20:
                 mtlr r0
                 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:
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:
                 mflr 0
.main:
                 7c0802a6
                   st r0, 0x8(r1)
                 9001008
                   st r1, -56(r1)
                 9421ffc8
                   lil r4, 0x1
                 38800001
                   oril r3, r4, 0x0
                 60830000
                   bl - .f
                 4bffff71
                   1 \text{ 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 **Inum** as a 4-byte decimal, octal, and hexadecimal number by entering the commands:

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
number: 0
              1c8 0
                        0
                              0
                                  4d2
                                        0
             0 0
                      0 5468 6973 2069 7320
       3fa0
       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:

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:
        5468 6973 2069 7320
                                This is
str1:
        6120 6368 6172 6163
str1+8:
                               a charac
str1+10: 7465 7220 7374 7269 ter stri
                               ng~@~@~@~@~@~A
str1+18: 6e67
                  0
                       1
             0
number:
                         0
                                ~@~@~A~<c8>~@~@~@~@
         0
              1c8 0
number:
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:

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:

```
.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 an application program at two levels: the source-level and the assembler language-level. Source level debugging allows you to debug your C, C++, 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 application 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 an application 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 64
- · "Examining Program Data" on page 68
- "Debugging at the Machine Level with dbx" on page 74
- "Customizing the dbx Debugging Environment" on page 76
- "Developing for the dbx Plug-in Framework" on page 78

Using the dbx Debug Program

This section contains 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 77) 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 5.2 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**, **next**i, 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. This is only true if the process stopped in a location compiled for debugging.

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 [/]
? RegularExpression [?]

Searches forward in the current source file for the given expression. 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 threadThe 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 66).

All threads resume execution during the step, next, and nexti subcommands. If step, next, nexti

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

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 \text{ 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:

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
                  running 12755 u no pro main
*$t1
       run
                       running 12501 k no sys thread_1
>$t2
        run
(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, detached state = 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)

[] (brackets) or () (parentheses)

. (period)

Denotes subscript array expressions.

Use this field reference operator with pointers and structures. This makes the C operator -> (arrow) unnecessary, although it is allowed.

& (ampersand)

. (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, II, &&

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 \setminus 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 = ' \setminus 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 *).

The following sections contain more information on debugging at the machine level with the **dbx** program.

- · "Using Machine Registers"
- "Examining Memory Addresses" on page 75
- "Running a Program at the Machine Level" on page 75
- "Displaying Assembly Instructions" on page 76

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 **\$rNumber**, 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.
- Prints a byte as a character. C
- D Prints a long word in decimal.
- d Prints a short word in decimal.
- f Prints a single-precision floating-point number.
- Prints a double-precision floating-point number. g
- Prints a byte in hexadecimal. h
- 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.
- 0 Prints a long word in octal.
- Prints a short word in octal.
- Prints an extended-precision floating-point number. q
- Prints a string of characters terminated by a null byte. S
- X Prints a long word in hexadecimal.
- 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
100002d0 \text{ (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[0xYYYYYYY]
```

In this example, 0xRRRRRRR 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"
- "Creating dbx Subcommand Aliases" on page 77
- "Using the .dbxinit File" on page 77

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
         stop (in given thread at specified function)
bfth
blth
         stop (in given thread at specified source line)
         cont
         condition
cv
d
         delete
         edit
P
h
         help
         status
         list
         map
m
mu
         mutex
         next
n
         print
p
q
         quit
         run
         step
S
st
         stop
         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
    where . . .
dbq-->
```

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.

Developing for the dbx Plug-in Framework

dbx provides a plug-in framework for developers who want to add new dbx subcommands and event handlers. Any dbx user can create a plug-in that enhances dbx with application or library specific commands to aid in debugging.

Notes:

- 1. Care should be taken not to confuse dbx callback routines and plug-in interface routines.
- 2. dbx callback routines are the set of services offered by dbx to the plug-in. The plug-in is given access to these routines through a set of function pointers.
- 3. plug-in interface routines are the set of methods dbx requires to be implemented by the plug-in.
- · "File Format"
- "Naming"
- · "Location" on page 79
- · "Loading" on page 79
- "Unloading" on page 79
- "Version Control" on page 80
- "Header File" on page 80
- "Plug-in Interface" on page 80
- "dbx callback routines" on page 82
- "Example" on page 93

File Format

Each plug-in must be a shared object file.

Naming

To correctly redirect subcommand input, **dbx** requires each plug-in to have a *unique name*.

The file name of the plug-in communicates this unique name to **dbx**. Upon initialization, **dbx** searches a set of predefined and user-specified directories for files whose base name matches the regular expression: ^libdbx .+\.so\$

The following table shows examples of file names that are valid and not valid for **dbx plug-ins**. The corresponding *unique name* is shown for all valid examples:

File Name	Valid	Unique Name
libdbx_sample.so	Yes	sample
libdbx_xyz.so	Yes	xyz
libdbx_my_app.so	Yes	my_app
libdbx.so	No	
libdbxso	No	
libdbx_sample.so.plugin	No	
plugin_libdbx_sample.so	No	

Location

dbx allows the user to specify a list of directories to search using the DBX_PLUGIN_PATH environment variable. Each directory in the list should be separated by a colon. In the following example, the colon separates two directories.

```
$ export dbx PLUGIN PATH=$HOME/dbx plugins:/mnt/share/dbx plugins
```

Upon initialization, **dbx** searches for plug-ins. **dbx** also searches the directory of the executable file (if known). This directory is searched after the user-defined directories are searched.

Note: When you use **dbx** to attach to a process, the full path to the executable file cannot be determined.

Loading

A plug-in can be loaded in one of the following ways:

- A plug-in can be automatically loaded and initialized by placing it in a directory that is searched by dbx
 (as described in "Location"). This occurs at dbx initialization time.
- A plug-in can be manually loaded and initialized by specifying its location to the pluginload dbx subcommand. This can occur at any time during the dbx session.

After a successful automatic or manual plug-in load, a message similar to the following is displayed:

```
(dbx) pluginload /home/user/dbx_plugins/libdbx_sample.so
plug-in "/home/user/dbx plugins/libdbx sample.so" loaded
```

Any plug-in whose *unique name* is identical to that of a currently active plug-in is discarded and a warning message similar to the following is displayed.

```
(dbx) pluginload /mnt/share/dbx_plugins/libdbx_sample.so
could not load plug-in
"/mnt/share/dbx_plugins/libdbx_sample.so":
plug-in "/home/user/dbx plugins/libdbx sample.so" already loaded.
```

Unloading

Any plug-in, regardless of how it was loaded, can be manually unloaded by specifying its name to the **pluginunload dbx** subcommand. After a plug-in is successfully unloaded, a message similar to the following is displayed.

```
(dbx) pluginunload sample
plug-in "/home/user/dbx plugins/libdbx sample.so" unloaded.
```

Version Control

If changes are made to the plug-in framework that would otherwise break backward compatibility with existing plug-ins, a new version identifier will be created. This is true of any significant changes or additions to the **Plug-in Interface** or **Plug-in dbx callback routine**.

To minimize the need for frequent plug-in version changes, some **Plug-in dbx callback routines** require an additional parameter that represents the size of the buffer. This practice is used for buffer parameters that are based on system structures whose size is not controlled by **dbx**. This allows the size of the system structures to change without requiring updates to the plug-in version.

Currently, the only version identifier is DBX_PLUGIN_VERSION_1.

Header File

dbx Plug-in developers can find function prototypes, data structure definitions and macro definitions in the following header file:

/usr/include/sys/dbx plugin.h

Plug-in Interface

Refer to the dbx plugin.h header file for prototypes and definitions for the Plug-in Interface routines.

Each plug-in *must* implement and export all of the following routines:

- int dbx_plugin_version(void)
 For more information on this routine, see "int dbx_plugin_version(void)"
- int dbx_plugin_session_init(dbx_plugin_session_t session, constdbx_plugin_service_t *servicep)
 - For more information on this routine, see "int dbx_plugin_session_init(dbx_plugin_session_t session, constdbx_plugin_service_t *servicep)"
- void dbx_plugin_session_command(dbx_plugin_session_t session, int argc, char *const argv[]) For more information on this routine, see "void dbx_plugin_session_command(dbx_plugin_session_t session, int argc, char *const argv[])" on page 81
- void dbx_plugin_session_event(dbx_plugin_session_t session, int event, dbx plugin event info t *event infop)
 - For more information on this routine, see "void dbx_plugin_session_event(dbx_plugin_session_t session, int event, dbx_plugin_event_info_t *event_infop)" on page 81

int dbx_plugin_version(void)

This routine should return the **dbx Plug-in** version identifier corresponding to the version to which the plug-in conforms. Currently, the only version identifier is DBX PLUGIN VERSION 1.

int dbx_plugin_session_init(dbx_plugin_session_t session, constdbx_plugin_service_t *servicep)

This routine should perform any initialization needed for the plug-in to function properly before returning control back to **dbx**. This includes setting up any aliases for plug-in subcommands, if desired.

This routine should create a plug-in session that associates the given session identifier with the application program or core file. To identify the process or core file, the session identifier is used by **dbx** in **Plug-in Interface** calls and by the plug-in for **plugin dbx callback routine** requests. This routine also accepts the callback routine structure that contains pointers to the "dbx callback routines" on page 82.

This routine should return zero for successful initialization. If initialization is not successful, **dbx** unloads and discards the plug-in.

void dbx_plugin_session_command(dbx_plugin_session_t session, int argc, char *const argv[])

This routine should accept input from the **dbx** user in the form of arguments provided to the **plugin** subcommand. The syntax of the **plugin** subcommand is as follows:

plugin Name [arg0 arg1 arg2 ... argn]

This allows the **dbx** user to provide any input to any single plug-in. The plug-in has full control over what it accepts as input.

The **plugin** subcommand passes the command specified by the arg^* parameters to the plug-in specified by the *Name* parameter. (For example, the plug-in name could be <code>libdbx_Name.so</code>) Using this routine, **dbx** passes arg0 through argn to the plug-in. argv[0] corresponds to arg0, argv[1] to arg1, and so on.

In most circumstances, *arg0* would represent the name of a subcommand defined by the plug-in and *arg1* through *argn* would represent additional flags or arguments. However, this is not a requirement.

Developers are encouraged to implement a **help** subcommand which displays usage information for the plug-in.

void dbx_plugin_session_event(dbx_plugin_session_t session, int event, dbx_plugin_event_info_t *event_infop)

In response to application program events, this routine should perform any internal processing required by the plug-in. The routine is invoked once by **dbx** upon the occurrence of each event. The following table describes the event types for which a plug-in is notified:

ID (event)	Associated Data (event_infop)	Cause
DBX_PLUGIN_EVENT_RESTART	None	The dbx user executed the run subcommand.
DBX_PLUGIN_EVENT _EXIT	Exit code	The application program ended through the exit routine.
DBX_PLUGIN_EVENT _TERM	Terminating signal number	The application program terminated because of an unhandled signal.
DBX_PLUGIN_EVENT _LOAD	dbx_plugin_modinfo_t structure of loaded module	A module was loaded into the application program.
DBX_PLUGIN_EVENT _UNLOAD	dbx_plugin_modinfo_t structure of unloaded module	A module was unloaded from the application program.
DBX_PLUGIN_EVENT_BP	None	The application program has stopped because of a user or internal dbx breakpoint or data watchpoint.
DBX_PLUGIN_EVENT_SIGNAL	Signal number	The application program stopped because of a signal delivery.

ID (event)	Associated Data (event_infop)	Cause
DBX_PLUGIN_EVENT_SWTHRD	Handle of current pthread	The dbx user executed the thread current <handle> subcommand resulting in a change in the current pthread.</handle>

The DBX_PLUGIN_EVENT_BP and DBX_PLUGIN_EVENT_SIGNAL events imply that the application program was started but has stopped. These events are meant to signify that any cached data that the plug-in possesses might no longer be valid. Upon notification of these events, it is more efficient for plug-ins to simply invalidate any cached data rather than refreshing the data. A complete refresh of cached data should only occur when the data is needed. This is especially relevant because some signals might be ignored by dbx and some breakpoints might be internal breakpoints. If the user has no opportunity to run subcommands before the application program starts again, repeatedly refreshing data wastes resources.

void dbx plugin session destroy(dbx plugin session t session)

This routine should perform any final cleanup and memory management tasks required by the plug-in.

dbx callback routines

The following are the dbx callback routines provided for each plug-in through the dbx plugin session init routine.

- · "session"
- "process" on page 83
- "fds" on page 83
- "modules" on page 84
- "regions" on page 85
- "threads" on page 86
- · "pthreads" on page 87
- "get_thread_context" on page 87
- "set thread context" on page 88
- · "get pthread context" on page 88
- "set pthread context" on page 89
- "read_memory" on page 90
- "write memory" on page 90
- "locate symbol" on page 90
- · "what_function" on page 91
- "print" on page 92
- "alias" on page 92

session

The dbx session callback routine allows you to get characteristics of the dbx session. dbx fills in the flagsp parameter.

```
typedef int (*dbx_plugin_session_service_t)(dbx_plugin_session_t session,
                                            dbx plugin session flags t *flagsp).
```

The dbx **session** callback routine parameters are:

Parameter	Description
session	Session identifier.
flagsp	 Session characteristics in any combination of: DBX_PLUGIN_SESSION_64BIT If set, the session represents a 64-bit application program. Otherwise, the session represents a 32-bit application program. DBX_PLUGIN_SESSION_CORE If set, the session represents a core file. Otherwise, the session represents a live process.

The dbx **session** callback routine return codes are:

- DBX PLUGIN SUCCESS
- DBX_PLUGIN_BAD_SESSION session is not valid
- DBX_PLUGIN_BAD_POINTER flagsp is NULL

process

The dbx **process** callback routine allows you to get information on the process being debugged. **dbx** populates the *infop* parameter.

The dbx process callback routine parameters are:

Parameter	Description
session	Session identifier
infop	Allocated dbx_plugin_procinfo_t structure
procinfo_size	Size of dbx_plugin_procinfo_t structure

The dbx **process** callback routine return codes are:

- DBX PLUGIN SUCCESS
- DBX_PLUGIN_BAD_SESSION session is not valid
- DBX_PLUGIN_BAD_POINTER infop is NULL
- DBX_PLUGIN_BAD_ARG procinfo_size is not valid
- DBX_PLUGIN_UNAVAILABLE process not active or info not in core

fds

The dbx **fds** callback routine allows you to get information on file descriptors for the process. You can either:

- · Call iteratively to get information separately on each file descriptor. Or,
- Call once to get the total number of file descriptors and call once again to get information on all file descriptors simultaneously.

If the plug-in passes a non-NULL *infop* buffer, **dbx** populates the buffer with the number of entries requested in *countp, starting with the file descriptor referenced by *indexp.

If the plug-in passes a *countp that is greater than the number of remaining entries, **dbx** retrieves all remaining entries. **dbx** updates countp to reflect the actual number of entries retrieved and indexp to reflect the next module index. If the last file descriptor was retrieved, indexp is set to -1. If the plug-in passes a NULL infop buffer, indexp and countp are still updated — just as if infop were non-NULL.

```
typedef int (*dbx plugin fds service t)(dbx plugin session t session,
                                        dbx plugin fdinfo t *infop,
                                        size t fdinfo size,
                                        unsigned int *indexp,
                                        unsigned int *countp)
```

The dbx **fds** callback routine parameters are:

Parameter	Description
session	Session identifier
infop	Allocated array of dbx_plugin_fdinfo_t structures or NULL
fdinfo_size	Size of a single dbx_plugin_fdinfo_t structure
indexp	Starting/next file descriptor (where zero corresponds to the first file descriptor)
countp	number of file descriptors

The dbx fds callback routine return codes are:

- DBX PLUGIN SUCCESS
- DBX PLUGIN BAD SESSION session is not valid
- DBX PLUGIN BAD POINTER indexp is NULL or countp is NULL
- DBX_PLUGIN_BAD_ARG fdinfo_size is not valid or * countp == 0
- DBX PLUGIN UNAVAILABLE process not active or info not in core

modules

The dbx modules callback routine allows you to get information on loaded modules for the process. You can either:

- · Call iteratively to get information separately on each module. Or,
- Call once to get the total number of modules and call once again to get information on all modules simultaneously.

If the plug-in passes a non-NULL infop buffer, **dbx** populates the buffer with the number of entries requested in *countp, starting with the module referenced by *indexp.

If the plug-in passes a *countp that is greater than the number of remaining entries, dbx retrieves all remaining entries. dbx updates countp to reflect the actual number of entries retrieved and updates indexp to reflect the next module index. If the last module was retrieved, indexp is set to -1. If the plug-in passes a NULL infop buffer, indexp and countp are still updated — just as if infop were not NULL.

Note: This routine allocates memory to hold the file name and member character strings. The caller must free this memory when it is no longer needed.

```
typedef int (*dbx plugin modules service t)(dbx plugin session t session,
                                             dbx plugin modinfo t *infop,
                                             size t modinfo size,
                                             unsigned int *indexp,
                                             unsigned int *countp)
```

The dbx **modules** callback routine parameters are:

Parameter	Description
session	Session identifier
infop	Allocated array of dbx_plugin_modinfo_t structures or NULL
modinfo size	Size of a single dbx plugin modinfo t structure

indexp Starting/next module (where zero corresponds to the first module)

countp Number of modules

The dbx modules callback routine return codes are:

- DBX_PLUGIN_SUCCESS
- DBX_PLUGIN_BAD_SESSION session is not valid
- DBX_PLUGIN_BAD_POINTER indexp is NULL or countp is NULL
- DBX_PLUGIN_BAD_ARG modinfo_size is not valid or *countp == 0

regions

The dbx regions callback routine allows you to get information on memory regions for the process.

Retrieved regions can include:

- Main thread stack region (DBX_PLUGIN_REGION_STACK)
- User data region (DBX PLUGIN REGION DATA)
- Process private data region (DBX_PLUGIN_REGION_SDATA)
- Memory mapped region (DBX_PLUGIN_REGION_MMAP)
- Shared memory region (DBX_PLUGIN_REGION_SHM)

You can either:

- Call iteratively to get information separately on one region. Or,
- Call once to get the total number of regions and call once again to get information on all regions simultaneously.

If the plug-in passes a non-NULL *infop* buffer, **dbx** populates the buffer with the number of entries requested in *countp, starting with the region referenced by *indexp.

If the plug-in passes a *countp that is greater than the number of remaining entries, **dbx** retrieves all remaining entries. **dbx** updates countp to reflect the actual number of entries retrieved and *indexp* to reflect the next region index.

If the last region was retrieved, *indexp* is set to -1. If the plug-in passes a NULL *infop* buffer, *indexp* and *countp* are still updated — just as if *infop* were non-NULL.

Note: Currently, this routine is only implemented for sessions representing core files. Sufficient information is not available to **dbx** for sessions representing live processes. Calls for such sessions return DBX_PLUGIN_UNAVAILABLE.

The dbx **regions** callback routine parameters are:

Parameter	Description
session	Session identifier
infop	Allocated array of dbx_plugin_region_t structures or NULL
reginfo_size	Size of a single dbx_plugin_reginfo_t structure
indexp	Starting/next region (where zero corresponds to the first region)
countp	Number of regions

The dbx **regions** callback routine return codes are:

- DBX PLUGIN SUCCESS
- DBX_PLUGIN_BAD_SESSION session is not valid
- DBX_PLUGIN_BAD_POINTER indexp is NULL or countp is NULL
- DBX PLUGIN BAD ARG reginfo size is not valid or *countp == 0
- DBX_PLUGIN_UNAVAILABLE session represents a live processes and regions not accessible

threads

The dbx threads callback routine allows you to get information on the kernel threads in the process.

You can either:

- Call iteratively to get information separately on one thread. Or,
- · Call once to get the total number of threads and call once again to get information on all threads simultaneously.

If the plug-in passes a non-NULL infop buffer, dbx populates the buffer with the number of entries requested in *countp, starting with the thread referenced by *indexp.

If the plug-in passes a *countp that is greater than or equal to the number of remaining entries, dbx retrieves all remaining entries and updates countp to reflect the actual number of entries retrieved.

If the last entry was retrieved, and countp is less than its passed value, indexp is set to -1. Otherwise, *indexp* is updated to reflect the thread id for the next request.

Note: If the value of *countp* passed is equal to the number of available entries, *countp* remains the same, but indexp is not set to -1.

If the plug-in passes a NULL infop buffer, indexp and countp are updated — just as if infop were non-NULL.

```
typedef int (*dbx plugin threads service t)(dbx plugin session t session,
                                               dbx plugin thrdinfo t *infop,
                                               size_t thrdinfo_size,
                                               tid6\overline{4}_t *indexp,
                                               unsigned int *countp)
```

The dbx **threads** callback routine parameters are:

Parameter	Description
session	Session identifier
infop	Allocated array of dbx_plugin_thrdinfo_t structures or NULL
thrdinfo_size	Size of a single dbx_plugin_thrdinfo_t structure
indexp	Starting/next thread id (where, on input, zero corresponds to the first thread)
countp	Number of threads

The dbx threads callback routine return codes are:

- DBX_PLUGIN_SUCCESS
- DBX_PLUGIN_BAD_SESSION session is not valid
- DBX_PLUGIN_BAD_POINTER indexp is NULL or countp is NULL
- DBX_PLUGIN_BAD_ID *indexp is not a valid id
- DBX_PLUGIN_BAD_ARG thrdinfo_size is not valid or *countp ==0
- DBX_PLUGIN_UNAVAILABLE process not active or entries not in core

pthreads

The dbx **pthreads** callback routine allows you to get information on pthreads in the process, including any kernel thread associations.

You can either:

- · Call iteratively to get information separately on one pthread. Or,
- Call once to get the total number of pthreads and call once again to get information on all pthreads simultaneously.

If the plug-in passes a non-NULL *infop* buffer, **dbx** populates the buffer with the number of entries requested in *countp, starting with the pthread referenced by *indexp.

If the plug-in passes a *countp that is greater than the number of remaining entries, **dbx** retrieves all remaining entries. **dbx** updates countp to reflect the actual number of entries retrieved and *indexp* to reflect the pthread handle for the next request.

If the last entry was retrieved, *indexp* is set to -1. If the plug-in passes a NULL *infop* buffer, *indexp* and *countp* are still updated — just as if *infop* were non-NULL.

If the first pthread is requested and countp is updated to zero, the process is not pthreaded.

The dbx **pthreads** callback routine parameters are:

Parameter	Description
session	Session identifier
infop	Allocated array of dbx_plugin_pthinfo_t structures or NULL
pthrdinfo_size	Size of a single dbx_plugin_pthrdinfo_t structure
indexp	Starting/next pthread handle (where, on input, zero corresponds to the first pthread and DBX_PLUGIN_PTHREAD_CURRENT corresponds to the current pthread in dbx)
countp	Number of pthreads

The dbx **pthreads** callback routine return codes are:

- DBX_PLUGIN_SUCCESS
- · DBX PLUGIN BAD SESSION session is not valid
- DBX PLUGIN BAD POINTER indexp is NULL or countp is NULL
- DBX_PLUGIN_BAD_ARG pthrdinfo_size is not valid or *countp == 0

get_thread_context

The dbx **get_thread_context** callback routine allows you to read a kernel thread's general purpose, special purpose, and floating point registers. **dbx** populates the *contextp* parameter.

The dbx **get_thread_context** callback routine parameters are:

Parameter	Description
session	Session identifier
reg_flags	Logical OR of at least one of DBX_PLUGIN_REG_GPRS, DBX_PLUGIN_REG_SPRS, DBX_PLUGIN_REG_FPRS
id	Kernel thread tid (tid64_t)
contextp	Allocated dbx_plugin_context_t structure
context_size	Size of dbx_plugin_context_t structure

The dbx **get_thread_context** callback routine return codes are:

- DBX_PLUGIN_SUCCESS.
- DBX_PLUGIN_BAD_SESSION session is not valid.
- DBX_PLUGIN_BAD_ID id is not valid.
- DBX PLUGIN BAD ARG reg flags is not valid or context size is not valid.
- DBX_PLUGIN_BAD_POINTER contextp is NULL
- DBX PLUGIN UNAVAILABLE process is not active or thread is in kernel mode and registers are not accessible.

set thread context

The dbx set_thread_context callback routine allows you to write to a kernel thread's general purpose, special purpose and floating point registers.

```
typedef int (*dbx_plugin_reg_service_t) (dbx_plugin_session_t session,
                    uint64 t reg flags,
                                  uint64 t id,
                                  dbx plugin context t *contextp,
                                  size t context_size)
```

The dbx **set_thread_context** callback routine parameters are:

Parameter	Description
session	Session identifier
reg_flags	Logical OR of at least one of DBX_PLUGIN_REG_GPRS, DBX_PLUGIN_REG_SPRS, DBX_PLUGIN_REG_FPRS
id	Kernel thread tid (tid64_t)
contextp	Allocated dbx_plugin_context_t structure
context_size	Size of dbx_plugin_context_t structure

The dbx set_thread_context callback routine return codes are:

- DBX_PLUGIN_SUCCESS
- DBX PLUGIN BAD SESSION session is not valid
- DBX_PLUGIN_BAD_ID id is not valid
- DBX_PLUGIN_BAD_ARG reg_flags is not valid or context_size is not valid
- DBX PLUGIN BAD POINTER contextp is NULL
- DBX_PLUGIN_UNAVAILABLE process is not active or thread is in kernel mode and registers are not accessible

get_pthread_context

The dbx get_pthread_context callback routine allows you to read a pthread's general purpose, special purpose and floating point registers. **dbx** populates the *contextp* parameter.

The dbx **get_pthread_context** callback routine parameters are:

Parameter	Description
session	Session identifier
reg_flags	logical OR of at least one of DBX_PLUGIN_REG_GPRS, DBX_PLUGIN_REG_SPRS, DBX_PLUGIN_REG_FPRS
id	pthread handle (pthdb_pthread_t)
contextp	Allocated dbx_plugin_context_t structure
context_size	size of dbx_plugin_context_t structure

The dbx **get_pthread_context** callback routine return codes are:

- DBX_PLUGIN_SUCCESS
- DBX_PLUGIN_BAD_SESSION session is not valid
- DBX_PLUGIN_BAD_ID id is not valid.
- DBX_PLUGIN_BAD_ARG reg_flags is not valid or context_size is not valid
- DBX_PLUGIN_BAD_POINTER contextp is NULL
- DBX_PLUGIN_UNAVAILABLE process is not active or thread is in kernel mode and registers are not accessible

set_pthread_context

The dbx **set_pthread_context** callback routine allows you to write to a pthread's general purpose, special purpose and floating point registers.

The dbx **set pthread context** callback routine parameters are:

Parameter	Description
session	Session identifier
reg_flags	Logical OR of at least one of DBX_PLUGIN_REG_GPRS, DBX_PLUGIN_REG_SPRS, DBX_PLUGIN_REG_FPRS
id	Pthread handle (pthdb_pthread_t)
contextp	Allocated dbx_plugin_context_t structure
context_size	Size of dbx_plugin_context_t structure

The dbx **set_pthread_context** callback routine return codes are:

- DBX_PLUGIN_SUCCESS
- DBX_PLUGIN_BAD_SESSION session is not valid
- DBX_PLUGIN_BAD_ID id is not valid
- DBX_PLUGIN_BAD_ARG reg_flags is not valid or context_size is not valid
- DBX_PLUGIN_BAD_POINTER contextp is NULL

 DBX_PLUGIN_UNAVAILABLE process is not active or kernel thread associated with pthread is in kernel mode and registers are not accessible

read_memory

The dbx **read_memory** callback routine allows you to read from the process's address space. **dbx** populates the buffer parameter.

The dbx read_memory callback routine parameters are:

Parameter	Description
session	Session identifier
addr	Address to read from
buffer	Allocated buffer to hold memory contents
len	Number of bytes to read

The dbx read_memory callback routine return codes are:

- DBX_PLUGIN_SUCCESS
- DBX_PLUGIN_BAD_SESSION session is not valid
- DBX_PLUGIN_BAD_POINTER buffer is NULL
- DBX_PLUGIN_UNAVAILABLE unable to read from addr

write_memory

The dbx write_memory callback routine allows you to write to the process's address space.

The dbx write_memory callback routine parameters are:

Parameter	Description
session	Session identifier
addr	Address to write to
buffer	Allocated and initialized buffer
len	Number of bytes to write

The dbx write_memory callback routine return codes are:

- DBX PLUGIN SUCCESS
- DBX PLUGIN BAD SESSION session is not valid
- DBX PLUGIN BAD POINTER buffer is NULL
- DBX_PLUGIN_UNAVAILABLE unable to write to addr

locate_symbol

The dbx locate symbol callback routine allows you to convert symbol names to addresses.

The plug-in must initialize the *name* and *mod* fields of each entry in the symbols parameter array. The *name* field specifies the name of the symbol to be located. The *mod* field specifies the module index of the module in which the lookup should occur. A *mod* field initialized to -1 denotes that all modules should be searched.

dbx populates the *addr* field. Any unknown symbols have an address of zero. If the symbol is located and all modules searched, **dbx** updates the *mod* field with the actual module index of the symbol.

The dbx locate_symbol callback routine parameters are:

Parameter	Description
session	Session identifier
symbols	Allocated array of dbx_plugin_sym_t structures with the <i>name</i> and <i>mod</i> fields initialized
syminfo_size	Size of dbx_plugin_sym_t structure
count	Number of symbols to locate

The dbx **locate_symbol** callback routine return codes are:

- DBX_PLUGIN_SUCCESS
- DBX_PLUGIN_BAD_SESSION session is not valid
- DBX_PLUGIN_BAD_ARG syminfo_size is not valid
- DBX_PLUGIN_BAD_POINTER symbols is NULL

what_function

The dbx what_function callback routine allows you to convert text addresses to symbols.

The plug-in must initialize the *addr* field of each entry in the symbols parameter array. The *addr* field specifies an instruction address within the function to be identified.

dbx populates the *name* field. Any unknown text address has a name of NULL. **dbx** populates the *mod* field with the actual module index of the text address.

The dbx what_function callback routine parameters are:

Parameter	Description
session	Session identifier
symbols	Allocated array of dbx_plugin_sym_t structures with the addr field(s) initialized with text address(es)
syminfo_size	Size of dbx_plugin_sym_t structure
count	Number of addresses to convert

The dbx what_function callback routine return codes are:

- DBX_PLUGIN_SUCCESS
- DBX_PLUGIN_BAD_SESSION session is not valid

- · DBX PLUGIN BAD ARG syminfo size is not valid
- DBX PLUGIN BAD POINTER symbols is NULL

print

The dbx print callback routine allows you to display informational output or error output.

```
typedef int (*dbx plugin print service t)(dbx plugin session t session,
                                          int print mode.
                                          char *message)
```

The dbx **print** callback routine parameters are:

Parameter	Description
session	session identifier
print_mode	Either DBX_PLUGIN_PRINT_MODE_OUT or DBX_PLUGIN_PRINT_MODE_ERR
message	Character string for dbx to display

The dbx **print** callback routine return codes are:

- DBX_PLUGIN_SUCCESS
- DBX_PLUGIN_BAD_SESSION session is not valid
- DBX_PLUGIN_BAD_ARG print_mode is not valid
- DBX_PLUGIN_BAD_POINTER message is NULL

alias

The dbx alias callback routine allows you to create an alias for a plug-in subcommand.

The syntax of the plugin dbx subcommand requires the dbx user to type a prefix of plugin Name for each plug-in subcommand invocation. To provide a way to shorten such invocations, dbx allows plug-ins to create new aliases.

The alias and expansion parameters should provide a description of the new alias. The syntax is the same as the syntax defined for the alias dbx subcommand.

The following are example invocations of the dbx alias callback routine:

```
alias("intprt", "plugin xyz interpret");
alias("intprt2(addr, count, format)", "addr / count format; plugin xyz interpret addr");
```

Note: If you try to create an alias that has the same name as an existing alias, the request is denied and a warning message is displayed. Plug-in developers are encouraged to carry out alias creation in a way that allows users to correct alias conflicts. One way to accomplish this is by reading alias definitions from a configuration file that is packaged with the plug-in.

```
typedef int (*dbx plugin alias service t)(dbx plugin session t session,
                                          const char *alias,
                                          const char *expansion)
```

The dbx alias callback routine parameters are:

Parameter	Description
session	Session identifier
alias	Character string representing the alias name and optional parameter
expansion	Character string representing the alias expansion

The dbx alias callback routine return codes are:

- DBX_PLUGIN_SUCCESS
- DBX_PLUGIN_BAD_SESSION session is not valid
- DBX_PLUGIN_BAD_ARG alias is not valid
- DBX_PLUGIN_BAD_POINTER alias is NULL or expansion is NULL
- DBX_PLUGIN_UNAVAILABLE an alias with an identical name already exists

Example

void

The following example defines a **help** subcommand and a **hello** subcommand.

```
example.c:
#include <sys/dbx plugin.h>
dbx_plugin_session_t sid;
dbx_plugin_service_t dbx;
static void usage(void);
static void hello cmd(void);
dbx plugin version(void) {
             return dbx PLUGIN VERSION 1;
int dbx_plugin_session_init(dbx_plugin_session_t session,
                            const dbx_plugin_service_t *servicep) {
             /* record session identifier */
             sid= session;
             /* record dbx service */
             memcpy(&dbx, servicep, sizeof(dbx plugin service t));
             (*(dbx.alias))(sid, "hello", "plugin example hello");
             return 0;
dbx plugin session command(dbx plugin session t session,
                           int argc,
                           char *const argv[]) {
    if (argc == 0 || (argc == 1 && strcmp(argv[0], "help") == 0)) {
                usage();
                return;
    if (argc == 1 && strcmp(argv[0], "hello") == 0) {
                hello cmd();
                return;
    (*(dbx.print))(sid,dbx PLUGIN PRINT MODE ERR,
                       "unrecognized command\n");
dbx plugin session event(dbx plugin session t session,
                         int event,
                         dbx_plugin_event_info_t *event_infop) {
             /* ignore event notifications */
}
```

```
dbx plugin session destroy(dbx plugin session t session){
             /* no clean up to perform */
static
void
usage(void) {
              (*(dbx.print))(sid,dbx PLUGIN PRINT MODE OUT,
                        "Subcommands for Plug-in \"example\":\n\n" \
                        " help - displays this output\n" \
" hello - displays a greeting\n" \
}
static
void
hello cmd(void) {
              (*(dbx.print))(sid,dbx_PLUGIN PRINT MODE OUT,
                        "Hello dbx World!\n");
example.exp:
dbx plugin version
dbx_plugin_session init
dbx plugin session command
dbx plugin session event
dbx_plugin_session_destroy
To compile the example plug-in, you type:
   cc -o libdbx_example.so example.c -bM:Sre -bE:example.exp -bnoentry
```

List of dbx Subcommands

The commands and subcommands for the **dbx** debug program are located in the AIX 5L Version 5.2 Commands Reference.

The **dbx** debug program provides subcommands for performing the following task categories:

- "Setting and Deleting Breakpoints"
- "Running Your Program" on page 95
- "Tracing Program Execution" on page 95
- · "Ending Program Execution" on page 95
- "Displaying the Source File" on page 95
- "Printing and Modifying Variables, Expressions, and Types" on page 95
- "Thread Debugging" on page 96
- · "Multiprocess Debugging" on page 96
- · "Procedure Calling" on page 96
- "Signal Handling" on page 96
- · "Machine-Level Debugging" on page 96
- "Debugging Environment Control" on page 96

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.

listDisplays lines of the current source file.listiLists instructions from the application.moveChanges 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 gotoiDisplays the contents of memory.
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.

stepistopiRuns one source instruction.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.

changes the dbx prompt to the specified string.creenOpens 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.

Related Information

The **adb** command.

Chapter 4. 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 to the **errlog** subroutine. This error information is then 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. Entries can also be retrieved using the services provided in **liberrlog**, **errlog_open**, **errlog_close**, **errlog_find_first**, **errlog_find_next**, **errlog_find_sequence**, **errlog_set_direction**, and **errlog_write**. **errlog_write** provides a limited update capability.

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 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.

You should be familiar with the following terms:

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 information 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.

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:

- · Chapter 4, "Error-Logging Overview"
- "Managing Error Logging" on page 100
- "Error Logging Tasks" on page 106
- "Error Logging and Alerts" on page 113

"Error Logging Controls" on page 114

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 following services for logging errors to the error log file:

- · errlog subroutines
- · errsave and errlast kernel service
- error device driver (/dev/error)
- · error daemon
- · errstop command

The commands required for licensed program installation (**errinstall** and **errupdate**) are also included in **bos.rte**. 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."

Managing Error Logging

Error logging is automatically started by the **rc.boot** script during system initialization and is automatically stopped by the **shutdown** script during system shutdown. The error log analysis performed by the **diag** command analyzes hardware error entries. The default length of time that hardware error entries remain in the error log is 90 days. If you remove hardware error entries less than 90 days old, you can limit the effectiveness of this error log analysis.

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 to 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 -1

You can transfer the file to another system in a number of ways. You can:

- · Copy the file to a remotely mounted file system using the cp command
- Copy the file across the network connection using the rcp, ftp, or tftp commands
- 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 -I. 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.

Customizing the Log File Location

To change the file name 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, type:

/usr/lib/errdemon -s LogSize

The specified size limit for the log file is saved in the error-log configuration database, and the error daemon is immediately restarted. If the size limit for the log file 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, 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, type:

/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 that the error daemon is started after the system is rebooted. The buffer cannot be made smaller than the hard-coded default of 8 KB. The size you specify is rounded up to the next integral multiple of the memory page size (4 KBs). 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).

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 can correct the problem by enlarging the buffer.

Customizing Duplicate Error Handling

By default, starting with AIX 5.1, the error daemon eliminates duplicate errors by looking at each error that is logged. An error is a duplicate if it is identical to the previous error, and if it 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 accumulate 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 is logged at that point rather than waiting for the time interval to expire or a unique error to occur.

For example, if a device handler starts logging many identical errors rapidly, most will not appear in the log. Rather, the first occurrence will be logged. Subsequent occurrences will not be logged immediately, but are only 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 the number of duplicates.

Note: The time interval refers to the time since the last error, not the time since the first occurrence of this error, that is, it is reset each time an error is logged. Also, 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, or 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

A crontab file provided with the system deletes hardware errors older than 90 days and other errors older than 30 days. To display the **crontab** entries for your system, type:

crontab -1 Command

To change these entries, type:

crontab -e 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, type:

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, type:

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

To change the current settings for an event, you can use the errupdate command The necessary input to the errupdate command can be in a file or from standard input.

The following example uses standard input. To disable the reporting of the ERRLOG OFF event (error ID 192AC071), type the following to run the **errupdate** command:

```
errupdate <Enter>
=192AC071: <Enter>
Report=False <Enter>
<Ctrl-D>
<Ctrl-D>
```

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.

The ras_logger command provides a way to log any error from the command line. It can be used to test newly created error templates and provides a way to log an error from a shell script.

Redirecting syslog Messages to Error Log

Some applications use syslog for logging errors and other events. To list error log messages and syslog messages in a single report, redirect 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 odmadd /tmp/syslog.add command (you must be logged in as root user).

For more information about concurrent error notification, see "Error Notification."

Error Notification

The Error Notification object class specifies the conditions and actions to be taken when errors are recorded in the system error log. The user specifies these conditions and actions in an Error Notification object.

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. Only processes running with the root user authority can add objects to the Error Notification object class. Error Notification objects contain the following descriptors:

en_alertflg

Identifies whether the error can be alerted. This descriptor is provided for use by alert agents associated with network management applications using the SNA Alert Architecture. The valid alert descriptor values are:

TRUE can be alerted FALSE cannot be alerted 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

en_dup

en_err64

en_label

en_method

Specifies the error identifier associated with a particular error. An error identifier can be any numeric value that is valid as a Predefined Attribute (PdAt) object class attribute value. The **errpt** command displays error identifiers as hexadecimal. For example, to select an entry that the **errpt** command displays with IDENTIFIER: 67581038, specify en crcid = 0x67581038.

If set, identifies whether duplicate errors as defined by the kernel should be matched. The valid **en_dup** descriptor values are:

TRUE Error is a duplicate.

FALSE Error is not a duplicate.

If set, identifies whether errors from a 64-bit or 32-bit environment should be matched. The valid **en_err64** descriptors value are:

TRUE Error is from a 64-bit environment.

FALSE Error is from a 32-bit environment.

Specifies the label associated with a particular error identifier as defined in the output of the **errpt -t** command.

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. This unique name is used when removing the object.

en_persistenceflg

en_pid

en_rclass

en_resource

en symptom

en_rtype

en_type

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 that send a signal to another process should not persist across system restarts. 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)

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.

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.

Identifies the name of the failing resource. For the hardware error class, a resource name is the device name.

Identifies the type of the failing resource. For the hardware error class, a resource type is the device type by which a resource is known in the devices object class.

Enables notification of an error accompanied by a symptom string when set to **TRUE**.

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 errors that can be alerted.

FALSE Matches errors that cannot be alerted.

0 Removes the Error Notification object at system restart.

non-zero

Retains the Error Notification object at system restart.

Examples

 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, type:

```
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, type:

```
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_rtype = ""
    en_rtype = ""
    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, type:

```
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.

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 108
- "Example of a Summary Error Report" on page 111
- "Generating an Error Report" on page 112
- "Stopping an Error Log" on page 112
- "Cleaning an Error Log" on page 112
- "Copying an Error Log to Diskette or Tape" on page 113
- "Using the liberrlog Services" on page 113

Reading an Error Report

To obtain a report of all errors logged in the 24 hours prior to the failure, type:

```
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 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 Machine ID Node ID Class Unique number for the event.

Identification number of your system processor unit.

Mnemonic name of your system.

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 analyzes the error log entries related to it to determine the state of the device.)

S Software.

Informational messages.

U Undetermined (for example, a network).

Severity of the error that has occurred. The following 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. 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. General class of the resource that detected the failure (for example, a device class of disk).

Type of the resource that detected the failure (for example, a device type of 355mb).

Path to the device. There may be up to four fields, which refer to drawer, slot, connector, and port, respectively.

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.

Summary of the error.

List of some of the possible sources of the error.

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.

Description of actions for correcting a user-caused error.

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.

Description of actions for correcting an installation-caused error.

Type

Resource Name

Resource Class

Resource Type

Location Code

VPD

Description
Probable Cause
User Causes

Recommended Actions Install Causes

Recommended Actions

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 installation 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 installation 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.

To display a shortened version of the detailed report produced by the -a flag, use the -A flag. The -A flag is not valid with the -a, -g, or -t flags. The items reported when you use -A to produce the shortened version of the report are:

- Label
- Date and time
- Type
- · Resource name
- Description
- Detail data

The example output of this flag is in the following format:

STOK RCVRY EXIT I ABFI: Date/Time: Tue Dec 14 15:25:33 TEMP Resource Name: tok0 Type:

Description PROBLEM RESOLVED

Detail Data FILE NAME line: 273 file: stok wdt.c

SENSE DATA

0000 0000 0000 0000 0000 0000 DEVICE ADDRESS 0004 AC62 25F1

Reporting can be turned off for some errors. To show which errors have reporting turned off, type:

```
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, type:

```
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 re-create 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 hardware problem (for example, with a SCSI adapter device driver) and could not recover from it. Diagnostic information might be associated with this type of error. If so, it displays at the end of the error listing, as illustrated in the following example of a problem encountered with a device driver:

LABEL: SCSI ERR1 ID: 0502F666

Date/Time: Jun 19 22:29:51

Sequence Number: 95

Machine ID: 123456789012 Node ID: host1 Class: Н PERM Type: Resource Name: scsi0 Resource Class: adapter Resource Type: hscsi 00-08 Location: VPD:

Device Driver Level......00 Diagnostic Level.....00 Displayable Message.....SCSI 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

 $0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000$

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

80-00 SCSI2 FRU: 30F8834

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: Н

PEND Type: Resource Name: TokenRing Resource Class: tok0 Adapter Resource Type: TokenRing Location:

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

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 $0000 \ 0100 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000$ 0000 0000

An error class value of **O** indicates that an informational message has been logged.

OPMSG LABEL: ID: AA8AB241

Date/Time: Jul 16 03:02:02

Sequence Number: 26042 Machine Id: 123456789012

Node Id: host1 Class: 0 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

To create an error report of software or hardware problems do the following:

1. Determine if error logging is on or off by determining 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 typing:

/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, type:

```
errpt -N hdisk1
```

3. Generate an error log report using SMIT. For example, use the **smit errpt** command:

smit errpt

- a. Select 1 to send the error report to standard output, or select 2 to send the report to the printer.
- b. Select **yes** to display or print error log entries as they occur. Otherwise, select **no**.
- c. Specify the appropriate device name in the Select resource names option (such as hdisk1).
- d. Select Do.

Stopping an Error Log

This procedure describes how to stop the error-logging facility.

To turn off error logging, use the errstop command. You must have root user authority to use this command.

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."

Turn off the error-logging facility when you are 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.

Cleaning an Error Log

Error-log cleaning is normally done for you as part of the daily cron command. If it is not done automatically, 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, you can clean just the old disk's 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, type:

```
errclear -d S 0
```

The errclear command deletes entries from the error log that are older than a specified number of days. The 0 in the previous example indicates that you want to delete entries for all days.

Use the smit errclear command:

Copying an Error Log to Diskette or Tape

Copy an error log by doing one of the following:

To copy the error log to diskette, use the Is and backup commands. Insert a formatted diskette into the
diskette drive and type:

```
ls /var/adm/ras/errlog | backup -ivp
```

• To copy the error log to tape, insert a tape in the drive and type:

```
ls /var/adm/ras/errlog | backup -ivpf/dev/rmt0
```

• To gather system configuration information in a **tar** file and copy it to diskette, use the **snap** command. Insert a formatted diskette into the diskette drive and type:

```
snap -a -o /dev/rfd0
```

Note: To use the **snap** command, you need root user authority.

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. The /dev/rfd0 names your disk drive.

To gather all configuration information in a tar file and copy it to tape, type:

```
snap -a -o /dev/rmt0
```

The /dev/rmt0 names your tape drive.

Using the liberrlog Services

The **liberrlog** services allow you to read entries from an error log, and provide a limited update capability. They are especially useful from an error notification method written in the C programming language, rather than a shell script. Accessing the error log using the **liberrlog** functions is much more efficient than using the **errpt** command.

The services are errlog_open, errlog_close, errlog_find_first, errlog_find_next, errlog_find_sequence, errlog_set_direction, and errlog_write.

Error Logging and Alerts

If the **Alert** field of an error record template is set to True, programs that 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

To control the error-logging facility, you can use error-logging commands, subroutines and kernel services, as well as files.

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 Allows the kernel and kernel extensions to write to the error log

errlog_openOpens an error logerrlog_closeCloses an error log

errlog_find_firstFinds the first occurrence of an error log entryerrlog_find_nextFinds the next occurrence of an error log entry

errlog_find_sequence Finds the error log entry with the specified sequence number

errlog_set_direction Sets the direction for the error log find functions

errlog_write Updates an error log entry

errresume Resumes error logging after an errlast command was issued.

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

Contains structures defined as arguments to the errsave kernel service /usr/include/sys/err_rec.h

and the errlog subroutine

Defines the interface to the liberrlog subroutines /usr/include/sys/errlog.h

/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

For further information on this topic, see the following:

• Error Logging Special Files in AIX 5L Version 5.2 Files Reference.

• The errsave kernel service in AIX 5L Version 5.2 Technical Reference: Kernel and Subsystems Volume

Subroutine References

The errlog subroutine in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 1.

Commands References

The crontab command in AIX 5L Version 5.2 Commands Reference, Volume 1.

The errclear command, errdead command, errdemon daemon, errinstall command, errlogger command, errmsg command, errpt command, errstop command, errupdate command in AIX 5L Version 5.2 Commands Reference, Volume 2.

The odmadd command, odmdelete command, odmget command in AIX 5L Version 5.2 Commands Reference, Volume 4.

The **snap** command in AIX 5L Version 5.2 Commands Reference, Volume 5.

Chapter 5. File Systems and Logical Volumes

A file is a one-dimensional array of bytes that can contain ASCII or binary information. AIX files can contain data, shell scripts, and programs. File names are also used to represent abstract objects such as sockets or device drivers.

Files are represented internally by index nodes (*i-nodes*). Within the journaled file system (JFS), an i-node is a structure that contains all access, timestamp, ownership, and data location information for each file. An i-node is 128-bytes in JFS and 512-bytes in the enhanced journaled file system (JFS2). 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.

JFS and JFS2 are supported by this operating system. The file system links the file and directory data to the structure used by storage and retrieval mechanisms.

This chapter contains the following topics about file systems:

- · "File Types"
- · "Working With JFS Directories" on page 119
- · "Working with JFS2 Directories" on page 121
- · "Working with JFS i-nodes" on page 123
- · "Working with JFS2 i-nodes" on page 124
- "Allocating JFS File Space" on page 126
- · "Allocating JFS2 File Space" on page 129
- "JFS File System Layout" on page 131
- "JFS2 File System Layout" on page 132
- "Writing Programs That Access Large Files" on page 133
- · "Linking for Programmers" on page 140
- "Using File Descriptors" on page 142
- · "Creating and Removing Files" on page 145
- "Working with File I/O" on page 146
- "File Status" on page 153
- "File Accessibility" on page 153
- "JFS File System Layout" on page 131
- "Creating New File System Types" on page 154

This chapter also contains information about programming considerations concerning Logical Volume Manager (LVM). See "Logical Volume Programming" on page 157.

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. For a list of subroutines used to control files, see "Working with Files" on page 119.

The kernel does not distinguish record boundaries in regular files, so programs can establish their own boundary markers. For example, many programs use line-feed characters to mark the end of lines. "Working with Files" on page 119 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 123 or "Working with JFS2 i-nodes" on page 124.

The journaled file system supports the following file types:

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 (IPC) channel. 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 is slightly less than 64 gigabytes (68589453312). In other file systems that are enabled for large files and in other JFS file system types, all files not listed as regular in the previous table have a maximum file size of 2 gigabytes minus 1 (2147483647). The maximum size of a file in JFS2 is limited by the size of the file system itself.

The architectural limit on the size of a JFS2 file system is 2⁵² bytes, or 4 petabytes. In AIX 5.2, the maximum JFS2 file size supported by the 32-bit kernel is 2^{40} - 4096 bytes, or just under 1 terabyte. The maximum file size supported by the 64-bit kernel is 2⁴⁴ - 4096 bytes, or just less than 16 terabytes.

The maximum length of a file name is 255 characters, and the maximum length of a path name is 1023 bytes.

Working with Files

The operating system provides many subroutines that manipulate files. For brief descriptions of the most common file-control subroutines, see the following:

- · "Creating Files"
- "Manipulating Files (Programming)"

Creating Files

The following subroutines are used when creating files:

creat Creates a new, empty, regular file

link Creates an additional name (directory entry) for an existing file

mkdir Creates a directorymkfifo Creates a named pipe

mknod Creates a file that defines a device

open Creates a new, empty file if the O_CREAT flag is set

pipe Creates an IPC socket Creates a socket

Manipulating Files (Programming)

The following subroutines can be used to manipulate files:

access Determines the accessibility of a file.

chmod Changes the access modes of a file.

chown Changes ownership of a file.

close Closes open file descriptors (including sockets).

fclearCreates space in a file.fcntl, dup, or dup2Control open file descriptors.

fsync Writes changes in a file to permanent storage.

ioctl Controls functions associated with open file descriptors, including special files,

sockets, and generic device support, such as the termio general terminal

interface.

lockf or **flock** Control open file descriptors.

Iseek or **Ilseek** Move the I/O pointer position in an open file.

openReturns a file descriptor used by other subroutines to refer to 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 Gets data from an open file if the appropriate permissions (O_RDONLY or

O_RDWR) were set by the **open** subroutine.

rename Changes the name of a file.

rmdir Removes directories from the file system.

stat Reports the status of a file, including the owner and access modes.

truncate Changes the length of a file.

write Puts data into an open file if the appropriate permissions (O_WRONLY or

O_RDWR) were set by the **open** subroutine.

For more information on types and characteristics of file systems, see File Systems Overview in AIX 5L Version 5.2 System Management Guide: Operating System and Devices.

Working With JFS Directories

Directories provide a hierarchical structure to the file system, link files, and i-node subdirectory names. 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 can 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 of variable length to allow file names the greatest flexibility. However, all directory space is allocated at all times.

No directory entry can 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 that the file occupied in the directory structure is added to the preceding directory entry. The information about the removed directory remains until a new entry fits into the space vacated.

Every 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.

Directories have the following access modes:

Allows a process to read directory entries read

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 JFS Directories (Programming)

The following is a list of subroutines available for working with directories:

closedir

Closes a directory stream and frees the structure associated with the DirectoryPointer parameter

mkdir Creates directories

opendir

Opens the directory designated by the *DirectoryName* parameter and associates a directory stream with it

readdir

Returns a pointer to the next directory entry

rewinddir

Resets the position of the specified directory stream to the beginning of the directory

rmdir Removes directories

seekdir

Sets the position of the next **readdir** subroutine operation on the directory stream

telldir Returns the current location associated with the specified directory stream

Changing the 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

You can cause the directory named by a process *Path* parameter to become the effective root directory by using 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 JFS 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 Changes the effective root directory chroot Gets path to current directory getcwd or getwd

Creates a directory mkdir

opendir, readdir, telldir, seekdir, rewinddir, or closedir

Perform various actions on directories

rename Renames a directory rmdir Removes a directory

Working with JFS2 Directories

Directories provide a hierarchical structure to the file system, link files, and i-node subdirectory names. There is no limit on the depth of nested directories. Disk space is allocated for directories in file system blocks.

Processes can read directories as regular files. However, the kernel can 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 that describe the objects contained in the directory. A directory entry has a fixed length and contains the following:

- · The i-node number
- The name (up to 22 bytes long)
- A name length field
- · A field to continue the entry if the name is longer than 22 bytes

The directory entries are stored in a B+ tree sorted by name. The self (.) and parent (..) information is contained in the i-node instead of in a directory entry. For more information about B+ trees, see "B+ Trees" on page 130.

Directories have the following access modes:

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 JFS2 Directories (Programming)

The following is a list of subroutines available for working with directories:

closedir

Closes a directory stream and frees the structure associated with the *DirectoryPointer* parameter

mkdir Creates directories

opendir

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.

readdir

Returns a pointer to the next directory entry

rewinddir

Resets the position of the specified directory stream to the beginning of the directory

rmdir Removes directories

seekdir

Returns to a position previously obtained with the telldir subroutine

telldir Returns the current location associated with the specified directory stream

Do not use the open, read, Iseek, and close subroutines to access directories.

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 JFS2 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 chroot Changes the current working directory Changes the effective root directory

opendir, readdir, telldir, seekdir, rewinddir, or closedir

Perform various actions on directories

Gets path to current directory

mkdirCreates a directoryrenameRenames a directoryrmdirRemoves a directory

Working with JFS i-nodes

getcwd or getwd

Files in the journaled file system (JFS) are represented internally as index nodes (i-nodes). JFS 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.

To determine the number of available i-nodes, use the **df -v** command. Disk i-nodes are defined in the /usr/include/jfs/ino.h file.

Disk i-node Structure for JFS

Each disk i-node in 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 Access permissions for the user ID i_uid i gid Access permissions for the group ID Number of blocks allocated to the file i nblocks i mtime Last time the file was modified Last time the file was accessed i_atime i ctime Last time the i-node was modified Number of hard links to the file i_nlink i_rdaddr[8] Real disk addresses of the data i_rindirect Real disk address of the indirect block, if any

You cannot change file data 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_mode) 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 "Allocating JFS File Space" on page 126.

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 determine the i-node number assigned to a file, use the **Is -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.

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.

JFS In-core i-node Structure

When a file is opened, the operating system creates an in-core i-node. The in-core i-node contains a copy of all the fields defined in the disk i-node, plus additional fields for tracking and managing access to the in-core i-node. When a file is opened, the information in the disk i-node is copied into an in-core i-node for easier access. In-core i-nodes are defined in the /usr/include/jfs/inode.h file. Some of the additional information tracked by the in-core i-node is as follows:

- 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 example, 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 JFS2 are represented internally as index nodes (i-nodes). JFS2 disk i-nodes exist in a static form on the disk and 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. Disk-inodes are defined in the /usr/include/j2/j2 dinode.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/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

Field Contents

Last time the file was modified di mtime di atime Last time the file was accessed di ctime Last time the 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

You cannot change the file data 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 "Allocating JFS2 File Space" on page 129.

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 determine the i-node number assigned to a file, use the Is -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.

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.

JFS2 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 that manage access to the disk i-node's valuable data. The fields of the in-core i-node are defined in the i2 inode.h file. Some of the additional information tracked by the in-core i-node is as follows:

- 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 example, 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).

Allocating JFS File Space

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* for JFS refers to the division of a file or directory's contents into 4096-byte 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 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 131 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 32 KB or larger in size. 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 eight 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 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 bytes 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

Allocation in Compressed JFS 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 32 KB) 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 JFS File Systems Enabled for Large Files

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 an ENOSPC error condition 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 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). Three fragments are 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.

JFS 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 the following 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 32 KB 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 32 KB, 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 (4 MB), or 1024 x 4096 bytes.

Double Indirect Method

The double indirect disk-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 (2 GB), or 512(1024 x 4096) bytes.

Note: The maximum file size that the read and write system calls would allow is 2 GB minus 1 (231-1). When memory map interface is used, 2 GB can be addresed. See "Writing Programs That Access Large Files" on page 133 for more information.

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 that any single user or group can monopolize.

The quotactI subroutine 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 the following types of limits:

Maximum limit allowed. When a process hits its hard limit, requests for more space fail. hard

Practical limit. If a process hits the soft limit, a warning is printed to the user's terminal. The soft 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 guota system allows processes access to the higher hard limit when more resources are temporarily required.

Allocating JFS2 File Space

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 a 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.

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 block sizes are: 512, 1024, 2048, and 4096 bytes. You can use only one block size in a file system. See "JFS2 File System Layout" on page 132 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.

JFS2 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 contiguous variable-length sequence of 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.

The length and address values are needed to define an extent. The length is measured in units of the file system block size. A 24-bit value represents the length of an extent, so an extent can range in size from 1 to 2²⁴ -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 separate 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

The B+ tree data structure is used for file layout. The most common operations that JFS2 performs are reading and writing extents. B+ trees are used to help with performance of these operations.

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 byte address the extent represents, and a flags field. The xad t structure is defined in the /usr/include/j2/j2 xtree.h file.

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 in length. 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.

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, JFS file systems contain a boot block, a superblock, bitmaps, and one or more allocation groups. Each file system occupies one logical volume.

JFS 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.

JFS Superblock

The superblock is 4096 bytes in size and starts at byte offset 4096 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

JFS Allocation Bitmaps

The file system contains the following 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.

JFS 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. "Allocating JFS File Space" on page 126 explains how the operating system allocates fragments.

JFS 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 that occupy some of the group's fragments. These fragments are reserved 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 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, but can be as large as 64 MB. These 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 to achieve optimum 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.

JFS 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 that contains the additional addresses.

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. Each file system occupies one logical volume.

JFS2 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

JFS2 Allocation Maps

The file system contains the following allocation maps:

The i-node allocation map records the location and allocation of all i-nodes in the file system.

The block allocation map records the allocation state of each file system block.

JFS2 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.

JFS2 Allocation Groups

Allocation groups divide the space on a file system into chunks. Allocation groups are used only for a problem-solving technique in which the most appropriate solution, found by attempting alternative methods, is selected at successive stages of a program for using in the next step of the program. Allocation groups allow JFS2 resource-allocation policies to use well-known methods for achieving optimum I/O performance. First, the 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 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 accommodate disk locality.

Allocation groups within a file system are identified by a zero-based allocation group index, the allocation group number.

Allocation group sizes must be selected that yield allocation groups that are large enough to provide for contiguous resource allocation over time. Allocation groups are limited to a maximum number of 128 groups. 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 that the nonexistent disk blocks will be marked as allocated in the block allocation map.

Writing Programs That Access Large Files

AIX supports files that are larger than 2 gigabytes (2 GB). This section assists programmers in understanding the implications of large files on their applications and to assist them in modifying their applications. Application programs can be modified, through programming interfaces, to be aware of large files. The file system programming interfaces generally are based on the off t data type.

Implications for Existing Programs

The 32-bit application environment that all applications used prior to AIX 4.2 remains unchanged. However, existing application programs cannot handle large files.

For example, the st_size field in the stat structure, which is used to return 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 use the stat subroutine with 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 the file size is irrelevant.

The errno EOVERFLOW can also be returned by an Iseek pointer and by the fcntl subroutine if the values that need to be returned are larger than the data type or structure that the program is using. For Iseek, if the resulting offset is larger than LONG_MAX, Iseek will fail and errno will be set to EOVERFLOW. For the fcntl subroutine, 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 existing application programs could have unexpected behavior, including data corruption, if allowed to operate on large files. AIX uses an open-protection scheme to protect applications from this class of failure.

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 GB boundary, the write subroutines will transfer data only up to 2 GB 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 the mmap, ftruncate, and fclear subroutines 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 GB threshold, only the data up to 2 GB 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.

Because 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

AIX provides two methods for applications to be enabled for large-file access. Application programmers must decide which approach best suits their needs:

- · Define _LARGE_FILES, which carefully redefines all of the relevant data types, structures, and subroutine names to their large-file enabled counterparts. Defining LARGE FILES has the advantage of maximizing application portability to other platforms because the application is still written to the normal POSIX and XPG interfaces. It has the disadvantage of creating some ambiguity in the code because the size of the various data items cannot be determined from looking at the code.
- Recode the application to explicitly call the large-file enabled subroutines. Recoding the application has the 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.

In either case, the application program *must* be carefully audited to ensure correct behavior in the new environment. Some of the common programming problems are discussed in "Common Pitfalls in Using the Large File Environment" on page 137.

Using LARGE FILES

In the default compilation environment, the off t data type is defined as a signed, 32-bit long. 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.

The following table shows the redefinitions that occur in the _LARGE_FILES environment:

Entity	Redefined As	Header File
off_t Object	long long	<sys types.h=""></sys>
fpos_t Object	long long	<sys types.h=""></sys>
struct stat Structure	struct stat64	<sys stat.h=""></sys>
stat Subroutine	stat64()	<sys stat.h=""></sys>
fstat Subroutine	fstat64()	<sys stat.h=""></sys>
Istat Subroutine	Istat64()	<sys stat.h=""></sys>
mmap Subroutine	mmap64()	<sys mman.h=""></sys>
lockf Subroutine	lockf64()	<sys lockf.h=""></sys>
struct flock Structure	struct flock64	<sys flock.h=""></sys>
open Subroutine	open64()	<fcntl.h></fcntl.h>
creat Subroutine	creat64()	<fcntl.h></fcntl.h>
F_GETLK Command Parameter	F_GETLK64	<fcntl.h></fcntl.h>
F_SETLK Command Parameter	F_SETLK64	<fcntl.h></fcntl.h>
F_SETLKW Command Parameter	F_SETLKW64	<fcntl.h></fcntl.h>
ftw Subroutine	ftw64()	<ftw.h></ftw.h>
nftw Subroutine	nftw64()	<ftw.h></ftw.h>
fseeko Subroutine	fseeko64()	<stdio.h></stdio.h>
ftello Subroutine	ftello64()	<stdio.h></stdio.h>
fgetpos ubroutine	fgetpos64()	<stdio.h></stdio.h>
fsetpos Subroutine	fsetpos64()	<stdio.h></stdio.h>
fopen Subroutine	fopen64()	<stdio.h></stdio.h>
freopen Subroutine	freopen64()	<stdio.h></stdio.h>
Iseek Subroutine	Iseek64()	<unistd.h></unistd.h>
ftruncate Subroutine	ftruncate64()	<unistd.h></unistd.h>
truncate Subroutine	truncate64()	<unistd.h></unistd.h>
fclear Subroutine	fclear64()	<unistd.h></unistd.h>
pwrite Subroutine	pwrite64()	<unistd.h></unistd.h>
pread Subroutine	pread64()	<unistd.h></unistd.h>
struct aiocb Structure	struct aiocb64	<sys aio.h=""></sys>
aio_read Subroutine	aio_read64()	<sys aio.h=""></sys>
aio_write Subroutine	aio_write64()	<sys aio.h=""></sys>
aio_cancel Subroutine	aio_cancel64()	<sys aio.h=""></sys>
aio_suspend Subroutine	aio_suspend64()	<sys aio.h=""></sys>
aio_return Subroutine	aio_return64()	<sys aio.h=""></sys>
aio_error Subroutine	aio_error64()	<sys aio.h=""></sys>
liocb Structure	liocb64	<sys aio.h=""></sys>
lio_listio Subroutine	lio_listio64()	<sys aio.h=""></sys>

Using 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>
                fgetpos64(FILE *, fpos64 t *);
extern int
extern FILE
                *fopen64(const char *, const char *);
extern FILE
                *freopen64(const char *, const char *, FILE *);
               fseeko64(FILE *, off64_t, int);
extern int
               fsetpos64(FILE *, fpos64_t *);
extern int
extern off64 t ftello64(FILE *);
<unistd.h>
extern off64 t lseek64(int, off64 t, int);
extern int
                ftruncate64(int, off64 t);
               truncate64(const char *, off64 t);
extern int
extern off64 t fclear64(int, off64 t);
extern ssize_t pread64(int, void *, size_t, off64_t);
extern ssize t pwrite64(int, const void *, size t, off64 t);
extern int
                fsync range64(int, int, off64 t, off64 t);
<sys/flock.h>
struct flock64;
<sys/lockf.h>
extern int lockf64 (int, int, off64 t);
<sys/mman.h>
                *mmap64(void *, size t, int, int, off64 t);
extern void
<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 *[]);

struct liocb64
int     lio listio64(int, struct liocb64 *[], int, void *);
```

Common Pitfalls in 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. Some of the more common problems and solutions are discussed in this section.

Note: In the following examples, **off_t** is assumed to be a 64-bit file offset.

Improper Use of Data Types

A common 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. To avoid this problem, 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;
```

Avoiding Parameter Mismatches

When you are passing 64-bit integers to functions as arguments or when you are 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.

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.

You can avoid problems by using function prototypes carefully. 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 because 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 because the source code remains portable between 32-bit 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, the application can remain vulnerable to failures due to arithmetic overflows. This problem usually occurs when the application performs an arithmetic operation 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 correctly for small values, but fails for large values. To fix this problem, type 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 occur when passing values based on fixed constants to functions that expect 64-bit parameters. In the following example, 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 and ftell Subroutines

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);
```

Including Correct Header Files

To see the function and data type redefinitions, application programs must include the correct 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. Because these programs typically hardcode the open modes, run-time failures occur when the program is compiled in the large-file environment. The program does call the correct **open** subroutine, and the resulting file descriptor is not enabled for large-file access. Ensure that programs include the correct header files, especially in the large-file environment, to get visibility to the redefinitions of the environment.

Incorrect:

```
fd = open("afile",2);

Better:
#include <fcntl.h>
fd = open("afile",0_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. Ensure that programs that do these conversions use the correct format specifier. This is especially difficult when the application needs to be portable between 32-bit and 64-bit environments because there is no portable format specifier between the two environments. You can handle this problem by writing offset converters that use the correct format for the size of **off t**.

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 such cases, the programmer must decide if the application 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 **soff_t** data type, 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, 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 that limits the 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, as follows.

```
struct rlimit r;
r.rlim cur = r.rlim max = RLIM INFINITY;
setrlimit(RLIMIT FSIZE,&r);
```

To set this limit from the Korn shell, run the following command:

```
ulimit -f unlimited
```

To set this value permanently for a specific user, use the **chuser** command, as shown in the following example:

```
chuser fsize hard = -1 root
```

The maximum size of a file is ultimately a characteristic of the file system itself, not just the file size limit or the environment, as follows:

- 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 (0x7ffffff). Attempts to write a file in excess of the maximum file size in any file system format will fail, and errno will be set to EFBIG.
- · For the JFS2, the maximum file size is limited by the size of 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 (see "Working with JFS i-nodes" on page 123 or "Working with JFS2 i-nodes" on page 124) 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 (see "Using File Descriptors" on page 142). 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. AIX supports 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.

Hard links must link file names and i-nodes within the same file system because 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. 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.

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. To change file ownership, use the **chown** subroutine.

Symbolic Links

Symbolic links are implemented as a file that contains a path name by using the symlink command. When a process encounters a symbolic link, the path contained in the symbolic link is prepended to the path that 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.

Subroutine created to report on the status of the file containing the symbolic link and does not follow Istat

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 Is -I command listing, an 1 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 -> 01dname).

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

Subroutine that also removes a file name by calling either the unlink or rmdir subroutine. remove

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. The following illustrates an example of the empty, well-formed directory /u/joe/foo and the i_nlink values.

/u

68		j	0	е	0

/u/joe mkdir ("foo", 0666)

68		n	0	0	0
		n	n	0	0
235		f	0	0	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).

Using File Descriptors

A file descriptor is an unsigned integer used by a process to identify an open file. The number of file descriptors available to a process is limited by the <code>/OPEN_MAX</code> control in the <code>sys/limits.h</code> file. The number of file descriptors is also controlled by the <code>ulimit -n</code> flag. The <code>open</code>, <code>pipe</code>, <code>creat</code>, and <code>fcntl</code> subroutines all generate file descriptors. File descriptors are generally unique to each process, but they can be shared by child processes created with a <code>fork</code> subroutine or copied by the <code>fcntl</code>, <code>dup</code>, and <code>dup2</code> 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.

File Descriptor Tables and System Open File Tables

The file descriptor and open file table structures track each process' access to a file and ensure data integrity.

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 the following fields: the flags area and the file pointer. There are up to OPEN_MAX file descriptors. The structure of the file descriptor table is as follows:

```
struct ufd
       struct file *fp;
       int flags;
} *u ufd
```

system open file table

Contains entries for each open file. A file table entry tracks 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 table 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 **Iseek** 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 any of the following:

- · 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 open file table entry. Separate table entries ensure each process has 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, in which bytes are not read or written sequentially.

Duplicating File Descriptors

File descriptors can be duplicated between processes in the following ways: the dup or dup2 subroutine, the **fork** subroutine, and the **fcntl** (file descriptor control) subroutine.

dup and dup2 Subroutines

The **dup** subroutine 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.

The dup2 subroutine scans for the requested descriptor assignment and closes the requested file descriptor if it is open. It allows the process to designate which descriptor entry the copy will occupy, if a specific descriptor-table entry is required.

fork Subroutine

The fork subroutine 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.

fcntl (File Descriptor Control) Subroutine

The **fcntl** subroutine manipulates file structure and controls open file descriptors. It 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 as follows:

- 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 example, the following changes the default assignments for file descriptors 0 and 1 from the terminal to the appropriate files:

```
prog < FileX > FileY
```

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)
        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*/
}
```

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 set at 2000. The limit can be changed by the ulimit command or the setrlimit subroutine. The maximum size is defined by the constant OPEN_MAX.

Creating and Removing Files

This section describes the internal procedures performed by the operating system when creating, opening, or closing files.

Creating a File

Different subroutines create specific types of files, as follows:

Subroutine Type of File Created creat Regular Regular (when the **O_CREAT** flag is set) open Regular, first-in-first-out (FIFO), or special mknod Named pipe (FIFO) mkfifo Unnamed pipe pipe socket Sockets Directories mkdir 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 does the following:

- 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 action 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. The current I/O offset designates a byte offset that is constantly tracked for every open file. The current I/O offset 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 Iseek subroutine. For more information, see "File Descriptor Tables and System Open File Tables" on page 142.

Manipulating the Current Offset

Read and write operations can access a file sequentially 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 Iseek subroutine.

Iseek

Allows a process to position the offset at a designated byte. The **Iseek** subroutine positions the pointer at the byte designated by the Offset variable. The Offset value can be calculated from the following 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 **Iseek** subroutine is the current value of the pointer's position in the file. For example:

```
cur off= lseek(fd, 0, SEEK CUR);
```

The Iseek subroutine is implemented in the file table. All subsequent 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

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 does the following:

- 1. Ensures 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 action 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 ensures 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.

To avoid an extra iteration in the read loop, start read requests at the beginning of data block boundaries and to be multiples of the data block size. If a process reads blocks sequentially, the operating system assumes all subsequent reads will also be sequential.

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 nnulls;
        int i;
        char buf[PAGESIZE];
                               /*A convenient buffer size*/
        nnulls=0;
        if ((fd = open("foo", 0 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.

If you write to a file that does not contain a block corresponding to the byte offset resulting from the write process, the write subroutine allocates a new block. This new block is added to the i-node information that defines the file. Adding the new block might allocate more than one block if the underlying file system needs to add blocks for addressing the file blocks.

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. The data is then 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 /tmp directory 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.

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. Lastly, it makes file writes asynchronous, allowing the application to continue processing instead of waiting for I/O requests to complete.

Direct I/O is an alternative caching policy that causes the file data to be transferred directly between the disk and the user's buffer. Direct I/O for files is functionally equivalent to raw I/O for devices. Applications can use direct I/O on JFS or JFS2 files.

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 that must use synchronous writes because these writes have to go to disk. In both of these cases, CPU usage is reduced because the data copy is eliminated.

A second benefit of direct I/O is that it allows applications to avoid diluting the effectiveness of caching of other files. Anytime a file is read or written, that file competes for space in the cache. This situation may cause other file data to be pushed out of the cache. If the newly cached data has very poor reuse characteristics, 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 releasing more cache space for files where the policies are effective.

Performance Costs of Direct I/O

Although direct I/O can reduce CPU usage, using it typically results in the process taking longer to complete, 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 read 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 the aio read subroutine.

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 that are converted to use direct I/O.

Conflicting File Access Modes

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 through the **shmat** or **mmap** system calls, the file will stay in normal cached mode.

The JFS or JFS2 will attempt to move the file into direct I/O mode anytime the last conflicting or non-direct access is eliminated (either by the close, munmap, or shmdt subroutines). Changing the file from normal mode to direct I/O mode can be rather expensive because 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 open subroutine. This flag is defined in the fcntl.h file. Applications must be compiled with _ALL_SOURCE enabled to see the definition of O_DIRECT.

Offset/Length/Address Alignment Requirements of the Target Buffer

For direct I/O to work efficiently, the request should be suitably conditioned. Applications can guery the offset, length, and address alignment requirements by using the finfo and ffinfo subroutines. When the FI_DIOCAP command is used, the finfo and ffinfo subroutines return information in the diocapbuf structure as described in the sys/finfo.h file. This structure contains the following fields:

dio_offset Recommended offset alignment for direct I/O writes to files in this file system dio max Recommended maximum write length for direct I/O writes to files in this system Recommended minimum write length for direct I/O writes to files in this file system dio_min Recommended buffer alignment for direct I/O writes to files in this file system dio_align

Failure to meet these requirements may cause file reads and writes to use the normal cached model and may cause direct I/O to be disabled for the file. Different file systems may have different requirements, as the following table illustrates.

File System Format	dio_offset	dio_max	dio_min	dio_align
JFS fixed, 4 K blk	4 K	2 MB	4 K	4 K
JFS fragmented	4 K	2 MB	4 K	4 K
JFS compressed	n/a	n/a	n/a	n/a
JFS big file	128 K	2 MB	128 K	4 K
JFS2	4 K	4 GB	4 K	4 K

Direct I/O Limitations

Direct I/O is not supported for files in a compressed-file file system. 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 that need this feature should use O DSYNC in addition to O_DIRECT. O_DSYNC guarantees that all of the data and enough of the metadata (for example, 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 metadata.

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 I (vertical) symbol represents a pipe between processes. In the following example, the output of the **Is** command is printed to the screen:

Pipes are treated as regular files as much 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 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 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.

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 correctly. 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 ascending 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 exit 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 and JFS2 file systems support the following types of synchronous I/O:

- 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.
- 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 still exists. 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.
- 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 and JFS2 file systems, only the combination of O_RSYNC I O_SYNC has meaning, indicating 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.

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.

Istat Subroutine that provides information about a symbolic link, and the **stat** subroutine returns information about the file referenced by the link.

fstat Returns information from an open file using the file descriptor.

The statfs, fstafs, and ustat subroutines return status information about a file system.

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.

statfs Returns information about the file system that contains the file specified by the *Path*

parameter.

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 by changing 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:

```
Existing file access mode
                             Reverse (umask) of original
                       permissions
File creation access mode
666 = R W _ R W _ R W _
ANDED TO
750 = R W X R _ X _ _ _ _
                             The umask value
640 = R W _ R _ _ _ _
                             Resulting file access mode
```

Investigates and reports on the accessibility mode of the file named in the Pathname access subroutine

> 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.

chmod and fchmod Changes file access permissions.

subroutines

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.

umask Sets and gets the value of the file creation mask.

In the following 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.

```
$ 1s -1
total 0
              1 root system 8290 Jun 09 17:07 special
-r-s--x
-rw----
              1 root system 1833 Jun 09 17:07 secrets
$ cat secrets
cat: cannot open secrets
```

The access subroutine must be used by any set-uID or set-gID program to forestall this type of intrusion. Changing file ownership and access modes are actions that affect the i-node, not the data in the file. To make these changes, the owner of the process must have root user authority or own the file.

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. This section provides information about the implementation specifics and execution syntax of file system helpers 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 this information to a back-end program. 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 helpers and mount helpers.

To determine the appropriate file system helper, the front-end command looks for a helper under the /sbin/helpers/vfstype/command file, 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.

One required file system helper that needs to be provided, called fstype, does not match a command name. This helper is used to identify if a specified logical volume contains a file system of the vfstype of the helper.

- The helper returns 0 if the logical volume does not contain a file system of its type. A return value of 0 indicates the logical volume does not contain a log.
- The helper returns 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. A return value of 1 indicates the logical volume does contain a log.
- The helper returns 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 -I flag is specified, the fstype helper should check for a log of its file system type on the specified logical volume.

Obsolete File System Helper Mechanism

This section describes the obsolete file system helper mechanism that 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
                            fsck
#define FSHOP CHGSIZ
                            chfs
#define FSHOP MAKE
                            mkfs
```

#define FSHOP STATFS df

#define FSHOP NAMEI 10 ff

File System Helper Execution Syntax

The execution syntax of the file system helper is as follows:

OpName OpKey FilsysFileDescriptor PipeFileDescriptor Modeflags DebugLevel OpFlags

Definition

OpName Specifies the arg0 parameter when the program invokes the helper. The value of the

OpName field appears in a list of processes (see the **ps** command).

OpKey Corresponds to the available helper operations. Thus, if the OpKey value is 1, the

fsck (file system check) operation is being requested.

FilsysFileDescriptor PipeFileDescriptor

Modeflags

Indicates the file descriptor on which the file system has been opened by the program. Indicates the file descriptor of the pipe (see the pipe subroutine) that is open between the original program and the helper program. This channel allows the helper to communicate with the front-end program.

Example: The helper sends an indication of its success or failure through the pipe, which can affect further front-end processing. Also, if the debug level is high enough, the helper can have additional information to send to the original program.

Provides an indication of how the helper is being invoked and can affect the behavior of the helper, especially in regard to error reporting. Mode flags are defined in the

/usr/include/fshelp.h file:

	•	
Fla	ags	Indicator
#define	FSHMOD_INTERACT_FLAG	"i"
#define	FSHMOD FORCE FLAG	"f"
#define	FSHMOD_NONBLOCK_FLAG	"n"
#define	FSHMOD_PERROR_FLAG	"p"
#define	FSHMOD_ERRDUMP_FLAG	"e"
#define	FSHMOD_STANDALONE_FLAG	"s"
#define	FSHMOD_IGNDEVTYPE_FLAG	"I"

Example: The **FSHMOD_INTERACT** flag indicates whether the command is being run interactively (determined by testing the isatty subroutine on the standard input). Not

every operation uses all (or any) of these modes.

DebugLevel Determines the amount of debugging information required: the higher the debugging

level, the more detailed the information returned.

OpFlags Includes the actual device (or devices) on which the operation is to be performed and

any other options specified by the front end.

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 types the following, the mount command checks the /etc/filesystems file for the stanza that describes the /test file system.

mount /test

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 as follows:

/test:

```
dev
                = /export
vfs
                = nfs
               = host1
nodename
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.

Mount Helper Execution Syntax

The following is a sample of the execution syntax of the mount helper:

/etc/helpers/nfsmnthelp M 0 host1 /export /test ro,fg,hard,intr

Both the mount and unmount commands have six parameters. The first four parameters are the same for both commands:

Indicates operation requested of the helper. Values are either M (mount operation), Q (query operation

operation), or U (unmount operation). The guery operation is obsolete.

Gives the numeric parameter for the -D flag. Neither the mount nor the unmount commands debuglevel

support the -D flag, so the value is 0.

Names the node for a remote mount or a null string for local mounts. The mount or unmount nodename

commands do not invoke a mount helper if the nodename parameter is null.

object Names the local or remote device, directory, or file that is being mounted or unmounted. Not all

file systems support all combinations. For example, most remote file systems do not support

device mounts, while most local file systems do not support anything else.

The remaining parameters for the **mount** command are as follows:

mount point

Names the local directory or file where the object is to be mounted.

options

Lists any file system-specific options, separated by commas. Information for this parameter comes from the options field of the relevant stanza in the /etc/filesystems file or from the -o Options flag on the command line (mount -o Options). The mount command also recognizes

the -r (read-only) flag and converts it to the string ro in this field.

The remaining parameters for the **unmount** command are as follows:

vfsNumber

Gives the unique number that identifies the mount being undone. The unique number is returned by the **vmount** call and can be retrieved by calling the **mntctl** or **stat** subroutine. For the mount helper, this parameter is used as the first parameter to the uvmount subroutine call that actually does the unmount.

flag

Gives the value of the second parameter to the uvmount subroutine. The value is 1 if the unmount operation is forced using the -f flag (umount -f). Otherwise, the value is 0. Not all file systems

support forced unmounts.

Logical Volume Programming

The Logical Volume Manager (LVM) consists of the library of LVM subroutines and the logical volume device driver, described as follows:

- · Library of LVM subroutines. These subroutines define volume groups and maintain the logical and physical volumes of volume groups.
- 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.

For a description of the readx and writex extension parameters and those ioctl operations specific to the logical volume device driver, see the AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts.

For more information about logical volumes, see the AIX 5L Version 5.2 System Management Guide: Operating System and Devices.

Library of Logical Volume Subroutines

LVM subroutines define and maintain the logical and physical volumes of a volume group. System management commands use these subroutines 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 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

online.

Related Information

For further information on this topic, see the following:

- Chapter 5, "File Systems and Logical Volumes," on page 117
- · Chapter 7, "Input and Output Handling," on page 171
- Chapter 18, "System Memory Allocation Using the malloc Subsystem," on page 391
- Linking Files and Directories, Processes Overview in AIX 5L Version 5.2 System User's Guide: Operating System and Devices
- File Systems, Logical Volume Storage: Overview in AIX 5L Version 5.2 System Management Concepts: Operating System and Devices
- Special Files Overview, Header Files Overview in AIX 5L Version 5.2 Files Reference
- · Understanding Generic I-nodes (G-nodes), Virtual File System Overview, Programming in the Kernel Environment, Understanding the Logical Volume Device Driver in AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts
- · Command Support for Files Larger than 2 Gigabytes in AIX 5L Version 5.2 Commands Reference, Volume 6
- File System Management Tasks in AIX 5L Version 5.2 System Management Guide: Operating System and Devices

Subroutine References

The access, accessx, or faccessx subroutine, chdir subroutine, chmod or fchmod subroutine, chown subroutine, chroot subroutine, close subroutine, exec, execl, execv, execle, execve, execlp, execvp or exect subroutine, fclear subroutine, fcntl, dup, or dup2 subroutine, fsync subroutine, ioctl or ioctlx subroutine, link subroutine, lseek subroutine, mknod or mkfifo subroutine, open, openx, open64, creat, or creat64 subroutine in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 1.

The read, readv, or readvx subroutine, readlink subroutine, remove subroutine, rmdir subroutine, statx, stat, Istat, fstatx, fstat, fullstat, ffullstat, stat64, Istat64, or fstat64 subroutine, symlink subroutine, truncate or ftruncate subroutines, umask subroutine, unlink subroutine, utimes or utime subroutine, write, writex, writev, or writevx subroutine in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2.

JFS File System Subroutines

The most commonly used JFS subroutines are as follows:

getfsent, getfsspec, getfsfile, getfstype, setfsent, or

endfsent Iseek

fscntl

mntctl

statfs, fstsfs, or ustat

sync

vmount or mount

Controls file system control operations Obtains information about a file system

Moves the read-write pointer Returns mount status information Reports file system statistics Updates file systems to disk Makes a file system ready for use

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 Subroutines

The most commonly used JFS2 subroutines are as follows:

fscntl getfsent, getfsspec, getfsfile, getfstype, setfsent, or

endfsent

lseek mntctl

statfs, fstsfs, or ustat

sync

vmount or mount

Controls file system control operations Obtains information about a file system

Moves the read-write pointer
Returns mount status information
Reports file system statistics
Updates file systems to disk
Makes a file system ready for use

Other subroutines are designed for use on virtual file systems (VFS):

getvfsent, getvfsbytype, getvfsbyname, getvfsbyflag,

sevfsent, or endvfsent

umount or uvmount

Retrieves a VFS entry

Removes VFS from the file tree

Commands References

The Is command, mkfs command in AIX 5L Version 5.2 Commands Reference, Volume 3.

The pr command in AIX 5L Version 5.2 Commands Reference, Volume 4.

Files References

The fullstat.h file, stat.h file, statfs.h file.

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

sigaction

The following subroutines are provided to accomplish these tasks:

fp_any_xcp or fp_divbyzero
fp_enable or fp_enable_all
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
fp_sh_info
fp_sh_set_stat
fp_trap
fp_underflow

Test the exception sticky flags Enable the signaling of exceptions Test the exception sticky flags

Determines which exceptions caused the signal Disables exceptions or clear flags
Changes the execution state of the process
Tests the exception sticky flags
Installs signal handler

© Copyright IBM Corp. 1997, 2004

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

Installs signal handler sigaction

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.
\star The handler will return the default signal return
* mechanism longjmp().
#include <signal.h>
#include <set.jmp.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
                                  /* 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_INEXACT"}
      { FP_DIV_BY_ZERO,
      { 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++)</pre>
    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);
                      FP UNDERFLOW);
TEST_IT("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);
FP_INV_IMZ);
TEST IT ("INF * ZERO",
                            FP_INV_CMP);
TEST_IT("invalid compare",
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");
  case FP TRAP IMP REC:
    puts("Fast mode for this platform is IMPRECISE RECOVERABLE");
    break;
    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);
```

Related Information

For further information on this topic, see the following:

- · Chapter 1, "Tools and Utilities," on page 1
- · Chapter 6, "Floating-Point Exceptions," on page 161

Subroutine References

The fp_clr_flag, fp_set_flag, fp_read_flag, or fp_swap_flag subroutine, fp_raise_xcp subroutine, fp_sh_trap_info or fp_sh_set_stat subroutine, fp_trap subroutine in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 1.

The **sigaction**, **sigvec** or **signal** subroutine in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2.

Chapter 7. Input and Output Handling

This chapter provides an introduction to programming considerations for input and output handling and the input and output (I/O) handling subroutines.

The 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 I/O 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 by 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 The low-level interface provides basic open and close functions for files and devices.

For more information, see "Low-Level I/O Interfaces."

The stream interface provides read and write I/O for pipes and FIFOs. For more Stream

information, see "Stream I/O Interfaces" on page 172.

The terminal interface provides formatted output and buffering. For more information, **Terminal**

see "Terminal I/O Interfaces" on page 173.

Asynchronous The asynchronous interface provides concurrent I/O and processing. For more

information, see "Asynchronous I/O Interfaces" on page 174.

The input language interface uses the lex and yacc commands to generate a lexical **Input Language**

analyzer and a parser program for interpreting I/O. For more information, see

"Generating a Lexical Analyzer with the lex Command" on page 279.

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:

Prepare a file, or other path object, for reading and writing by means of an assigned file open, openx, or creat

read, readx, readv, or

readvx

Read from an open file descriptor

write, writex, writev, or Write to an open file descriptor writevx 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 database, 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 occurs, 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, as well as 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 block sizes 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. 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. No record boundaries exist when reading and writing using stream I/O. For example, a process reading 100 bytes from a pipe cannot determine 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 process group ID. This field identifies the process group to receive the signals associated with the terminal. Terminal I/O interfaces can be 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 system console.

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 the terminal

termdef

Queries terminal characteristics

tcdrain

Waits for output to complete

tcflow Performs flow control functions

tcflush

Discards data from the specified queue

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 can provide 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 that is 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). When using LEGACY AIO, the kernel notifies the user process with the SIGIO signal. When using POSIX AIO, the sigevent structure is used by the programmer to determine which signal for the kernel to use to notify the user process. Signals include SIGIO, SIGUSR1, and SIGUSR2.

To use asynchronous I/O, a process must perform the following steps:

- 1. Establish a handler for the **SIGIO** signal. This step is necessary only 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 necessary only 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 Retrieves the error status of an asynchronous I/O request aio error

aio fsync Synchronizes asynchronous files.

aio_nwait Suspends the calling process until a certain number of asynchronous

I/O requests are completed.

Reads asynchronously from a file descriptor aio_read

Retrieves the return status of an asynchronous I/O request aio return 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

Defines the structure and flags used by the aio_read, aio_write, and aio.h

aio_suspend subroutines

Chapter 8. Large Program Support

This chapter provides information about using the large and very large address-space models to accommodate programs requiring data areas that are larger than those provided by the default address-space model. The large address-space model is available on AIX 4.3 and later. The very large address-space model is available on AIX 5.1 and later.

Note: The discussion in this chapter applies only 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 377 and Chapter 18, "System Memory Allocation Using the malloc Subsystem," on page 391.

The virtual address space of a 32-bit process is divided into 16 256-megabyte areas (or *segments*), each addressed by a separate hardware register. The operating system refers to segment 2 (virtual addresses 0x20000000-0x2FFFFFFF) as the *process-private* segment. By default, this segment contains the user stack and data, including the heap. The process-private segment also contains the u-block of the process, which is used by the operating system and is not readable by an application.

Because a single segment is used for both user data and stack, their maximum aggregate size is slightly less than 256 MB. Certain programs, however, require large data areas (initialized or uninitialized), or they need to allocate large amounts of memory with the **malloc** or **sbrk** subroutine. Programs can be built to use the large or very large address-space model, allowing them to use up to 2 GB of data.

It is possible to use either the large or very large address-space model with an existing program, by providing a non-zero *maxdata value*. The **maxdata** value is obtained either from the **LDR_CNTRL** environment variable or from a field in the executable file. Some programs have dependencies on the default address-space model, and they will break if they are run using the large address-space model.

Understanding the Large Address-Space Model

The large address-space model allows specified programs to use more than 256 MB of data. Other programs continue to use the default address-space model. To allow a program to use the large address-space model, specify a non-zero **maxdata** value. You can specify a non-zero **maxdata** value either by using the **Id** command when you're building the program, or by exporting the **LDR_CNTRL** environment variable before executing the program.

When a program using the large address-space model is executed, the operating system reserves as many 256 MB segments as needed to hold the amount of data specified by the **maxdata** value. Then, beginning with segment 3, the program's initialized data is read from the executable file into memory. The data read begins in segment 3, even if the **maxdata** value is smaller than 256 MB. With the large address-space model, a program can have a maximum of 8 segments or 2 GB or 3.25 GB of data respectively.

In the default address-space model, 12 segments are available for use by the **shmat** or **mmap** subroutines. When the large address-space model is used, the number of segments reserved for data reduces the number of segments available for the **shmat** and **mmap** subroutines. Because the maximum size of data is 2 GB, at least two segments are always available for the **shmat** and **mmap** subroutines.

The user stack remains in segment 2 when the large address-space model is used. As a result, the size of the stack is limited to slightly less than 256 MB. However, an application can relocate its user stack into a shared memory segment or into allocated memory.

While the size of initialized data in a program can be large, there is still a restriction on the size of text. In the executable file for a program, the size of the text section plus the size of the loader section must be

less than 256 MB. This is required so that these sections will fit into a single, read-only segment (segment 1, the TEXT segment). You can use the **dump** command to examine section sizes.

Understanding the Very Large Address-Space Model

The very large address-space model enables large data programs in much the same way as the large address-space model, although there are several differences between them. To allow a program to use the very large address-space model, you must specify a maxdata value and the dynamic segment allocation (dsa) property. Use either the Id command or the LDR CNTRL environment variable to specify a maxdata value and the **DSA** option.

If a maxdata value is specified, the very large address-space model follows the large-address space model in that a program's data is read into memory starting with segment 3, and occupies as many segments as needed. The remaining data segments, however, are not reserved for the data area at execution time, but are obtained dynamically. Until a segment is needed for a program's data area, it can be used by the **shmat** or **mmap** subroutines. With the very large address-space model, a program can a maximum of 13 segments or 3.25 GB of data. Of these 13 segments, 12 segments or 3 GB, are available for **shmat** and **mmap** subroutine purposes.

When a process tries to expand its data area into a new segment, the operation succeeds as long as the segment is not being used by the **shmat** or **mmap** subroutines. A program can call the **shmdt** or munmap subroutine to stop using a segment so that the segment can be used for the data area. After a segment has been used for the data area, however, it can no longer be used for any other purpose, even if the size of the data area is reduced.

If a **maxdata** value is not specified (maxdata = 0) with the dsa property, a slight variation from the above behaviour is achieved. The process will have its data and stack in segment 2, similiar to a regular process. The process will not have access to the global shared libraries, so all shared libraries used by the process will be loaded privately. The advantage to running this way is that a process will have all 13 segments (3.25 GB) available for use by the **shmat** and **mmap** subroutines.

To reduce the chances that the **shmat** or **mmap** subroutines will use a segment that could be used for the data area, the operating system uses a different rule for choosing an address to be returned (if a specific address is not requested). Normally, the **shmat** or **mmap** subroutines return an address in the lowest available segment. When the very large address-space model is used, these subroutines will return an address in the highest available segment. A request for a specific address will succeed, as long as the address is not in a segment that has already been used for the data area. This behaviour is followed for all process that specify the dsa property.

With the very large address-space model, a maxdata value of zero or a value of up to 0xD0000000 can be specified. If a maxdata value larger than 0xAFFFFFFF is specified, a program will not use globally loaded shared libraries. Instead, all shared libraries will be loaded privately. This can affect program performance.

Enabling the Large and Very Large Address-Space Models

The large address space model is used if any non-zero value is specified for the maxdata value, and the dynamic segment allocation (dsa) property is not specified. The very large address-space model is used if any maxdata value is given and the dsa property is specified. Use the Id command with the -bmaxdata flag to specify a maxdata value and to set the dsa property.

Use the following command to link a program that will have the maximum 8 segments reserved for its data:

cc -bmaxdata:0x80000000 sample.o

To link a program with the very large-address space model enabled, use the following command:

```
cc -bmaxdata:0xD0000000/dsa sample.o
```

You can cause existing programs to use the large or very large address-space models by specifying the maxdata value with the LDR_CNTRL environment variable. For example, use the following command to run the **a.out** program with 8 segments reserved for the data area:

```
LDR CNTRL=MAXDATA=0x80000000 a.out
```

The following command runs the a.out program using the very large address-space model, allowing the program's data size to use up to 8 segments for data:

```
LDR CNTRL=MAXDATA=0x800000000DSA a.out
```

You can also modify an existing program so that it will use the large or very large address-space model. To set the maxdata value in an existing 32-bit XCOFF program, a.out, to 0x80000000, use the following command:

```
/usr/ccs/bin/ldedit -bmaxdata:0x80000000 a.out
```

If an existing 32-bit XCOFF program, a.out, with a maxdata value of 0x80000000 does not already have the **DSA** property, you can add the property with the following command:

/usr/ccs/bin/ldedit -bmaxdata:0x80000000/dsa a.out

You can use the **dump** command to examine the **maxdata** value, or to determine whether a program has the dsa property.

Some programs have dependencies on the default address-space model. These programs terminate if a non-zero maxdata value has been specified, either by modifying the executable file of the program or by setting the LDR CNTRL environment variable.

Executing Programs with Large Data Areas

When you execute a program that uses the large address-space model, the operating system attempts to modify the soft limit on data size to match the maxdata value. If the maxdata value is larger than the current hard limit on data size, either the program will not execute if the environment variable XPG_SUS_ENV has the value set to ON, or the soft limit will be set to the current hard limit.

If the maxdata value is smaller than the size of the program's static data, the program will not execute.

After placing the program's initialized and uninitialized data in segments 3 and beyond, the break value is computed. 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, a process can move the break value to increase the size of the data area.

For example, if the maxdata value specified by a program is 0x68000000, then the maximum break value is in the middle of segment 9 (0x98000000). The brk subroutine extends the break value across segment boundaries, but the size of the data area cannot exceed the current soft data limit.

The setrlimit subroutine allows a process to set its soft data limit to any value that does not exceed the hard data limit. The maximum size of the data area, however, is limited to the original maxdata value, rounded up to a multiple of 256 MB.

The majority of subroutines are unaffected by large data programs. The **shmat** and **mmap** subroutines are the most affected, because they have fewer segments available for use. If a large data-address model program forks, the child process inherits the current data resource limits.

Special Considerations

Programs with large data spaces require a large amount of paging space. For example, if a program with a 2-GB address space tries to access every page in its address space, the system must have 2 GB of paging space. The operating system 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 using the large data model is no different than debugging other programs. The dbx command can debug these large programs actively or from a core dump. A full core dump from a large-data program can be quite large. To avoid truncated core files, be sure the coredump resource limit is large enough, and make sure that there is enough free space in the file system where your program is running.

Some application programs might be written in such a way that they rely on characteristics of the default address space model. These programs might not work if they execute using the large or very large address-space model. Do not set the LDR_CNTRL environment variable when you run these programs.

Processes using the very large address-space model must make code changes to their programs in order to move the break value of the address-space in chunks larger than 2 GB. This is a limitation of the sbrk system call which takes a signed value as the parameter. As a workaround, a program can call sbrk more than one time to move the break value to the desired position.

Related Information

For further information on this topic, see the following:

- "Program Address Space Overview" on page 377
- Bourne Shell Special Commands in AIX 5L Version 5.2 System User's Guide: Operating System and Devices

Subroutine References

- The brk or sbrk subroutine, exec subroutine, fork subroutine, malloc subroutine in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 1
- The setrlimit subroutine in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2

Commands References

- The dd command in AIX 5L Version 5.2 Commands Reference, Volume 2
- The Id command in AIX 5L Version 5.2 Commands Reference, Volume 3

Files References

The XCOFF Object (a.out) File Format in AIX 5L Version 5.2 Files Reference

Chapter 9. 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 can be improved by running different process threads on different processors. However, an individual program cannot take advantage of multiprocessing, unless it has multiple threads.

Multiprocessing is not apparent to most users because it is handled completely by the operating system and the programs it runs. Users can 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.

Identifying Processors

Symmetric multiprocessor (SMP) 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 must 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, peripheral devices such as printers or terminals, disks, memory boards, and processor boards. For more information about ODM, see Chapter 14, "Object Data Manager (ODM)," on page 335.

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 follows:

- 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.

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 "cpu_state Command" on page 180) 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 the following states:

Description State

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 the following ways:

- · A system administrator can control the availability of the processors for the whole system.
- A user can force a process or kernel threads to run on a specific processor.

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, shown in the form procx, where x is the physical processor number. For

more information, see "ODM Processor Names" on page 179.

CPU Logical processor number of enabled processor. For more information, see "Logical Processor

Numbers" on page 179.

Status Processor state of the next boot. For more information, see "ODM Processor States" on page 179. Location ODM processor location code, shown in the form AA-BB-CC-DD. "ODM Processor Names" on page

179

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 (enabled or disabled). If the processor does not respond, it is either faulty (an ODM state) or a communication error occurred. In this case, the cpu_state command displays No Reply.

Examples of 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 functioning and enabled. Consider a simple two-processor system with both processors enabled. The various ODM and number conventions are shown in the following table.

ODM Card Name	ODM Processor Name	3	ODM Current Processor State	cpu_state Status Field
cpucard0	proc0	0	Enabled	Enabled
cpucard0	proc1	1	Enabled	Enabled

For the preceding configuration, the **cpu_state -I** command produces a listing similar to the following:

Cpu Status Name Location Enabled 00-0P-00-00 Θ proc0 Enabled Property of the Control of t 00-0P-00-01 proc1 1

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.

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 the preceding, the cpu_state -I 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 Property of the Proper	00-0Q-00-00
proc3	3	Enabled Property of the Proper	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 in the following table shows the proc1 processor as disabled.

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 the preceding configuration, the cpu_state -I command now displays the status of proc1 as Disabled:

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 Property of the Proper	00-00-00-01

When the system is rebooted, the processor configuration is as shown in the following table.

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 -I** 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 Properties of the Contract of the Cont	00-00-00-01

Faulty Processor Configurations: The following example uses the last processor configuration discussed in the previous section. The system is rebooted with processors proc0 and proc3 failing their power-on tests. The processor configuration is as shown in the following table.

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 -I 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	00-00-00-01

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.

The process itself is not bound, but rather its kernel threads are bound. After 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 situation 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. After 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. To list available logical processor numbers, use the bindprocessor -q command. For a system with four enabled processors, this command produces output similar to the following:

```
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.

Using Dynamic Processor Deallocation

Starting with machine type 7044 model 270, the hardware of all systems with two or more processors can 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 an unrecoverable failure in the near future. This prediction is made by the firmware based-on-failure rates and threshold analysis.

AIX 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 the distinct probability that this system component will fail, the firmware returns an error report to AIX and 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 firmware flags the processor for persistent deallocation for subsequent reboots, until service personnel replace the processor.

Potential Impact to Applications

Processor deallocation is not apparent 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 bindprocessor interface for binding processes or threads to processors uses bind CPU numbers. The bind 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 the CPU is the "last" (highest numbered) bind CPU to be deallocated. For instance, on an 8-way SMP, the bind 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, CPU 7 seems to have disappeared, regardless of which physical processor failed.

Note: 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 forcibly 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 responsible for moving their bound threads and associated resources (such as timer request blocks) away form the last bind CPU ID 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 bind CPU ID, 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 that are binding threads to CPUs to get the notification of an impending processor deallocation, and to 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.

Flow of Events for Processor Deallocation

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, starts the deallocation process.
- 3. AIX notifies non-kernel processes and threads bound to the last bind CPU.
- 4. AIX waits for all the bound threads to move away from the last bind CPU. If threads remain bound, AIX eventually times out (after ten minutes) and aborts the deallocation. Otherwise, AIX invokes the previously registered High Availability Event Handlers (HAEHs). An HAEH may return an error that will abort the deallocation. Otherwise, AIX continues with the deallocation process and ultimately stops the failing processor.

In case of failure at any point of the deallocation, AIX logs the failure, indicating 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 continue this time) and restart the application.

Programming Interfaces Dealing with Individual Processors

The following sections describe available programming interfaces:

- · "Interfaces to Determine the Number of CPUs on a System"
- "Interfaces to Bind Threads to a Specific Processor" on page 185

Interfaces to Determine the Number of CPUs on a System

sysconf Subroutine

The **sysconf** subroutine returns a number of processors using the following parameters:

- _SC_NPROCESSORS_CONF: Number of processors configured
- SC NPROCESSORS ONLN: Number of processors online

For more information, see sysconf Subroutine in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2.

The value returned by the sysconf subroutine for SC NPROCESSORS CONF will remain constant between reboots. Uniprocessor (UP) machines are identified by a 1. Values greater than 1 indicate multiprocessor (MP) machines. The value returned for the _SC_NPROCESSORS_ONLN parameter will be the count of active CPUs and will be decremented every time a processor is deallocated.

The **system configuration.ncpus** field identifies the number of CPUs active on a machine. This field is analogous to the SC NPROCESSOR ONLN parameter. For more information, see systemcfq.h File in AIX 5L Version 5.2 Files Reference.

For code that must recognize how many processors were originally available at boot time, the ncpus_cfg field is added to the system configuration table, which remains constant between reboots.

The CPUs are identified by bind CPU IDs in the range [0..(ncpus-1)]. The processors also have a physical CPU number that depends on which CPU board they are on, in which order, and so on. The commands and subroutines dealing with CPU numbers always use bind CPU numbers. To ease the transition to varying numbers of CPUs, the bind CPU numbers are contiguous numbers in the range [0..(ncpus-1). 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") bind CPU is disappearing, regardless of which physical processor failed. **Note:** To avoid problems, use the *ncpus_cfg* variable to determine what the highest possible bind CPU number is for a particular system.

Interfaces to Bind Threads to a Specific Processor

The **bindprocessor**command and the **bindprocessor** programming interface allow you to bind a thread or a process to a specific CPU, designated by its bind CPU number. Both interfaces will allow you to bind threads or processes only to active CPUs. Those programs that 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 5.2 Commands Reference, Volume 1* or **bindprocessor** Subroutine in *AIX 5L Version 5.2 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 bind CPU than it is for kernel extensions.

Notification in User Mode

Each thread of a user mode application that is bound to the last bind CPU is sent the **SIGCPUFAIL** and **SIGRECONFIG** signals. These applications need to be modified to catch these signals and dispose of the threads bound to the last bind CPU (either by unbinding them or by binding them to a different CPU). For more information on the SIGRECONFIG signal, see Chapter 15, "Dynamic Logical Partitioning," on page 351

Notification in Kernel Mode

The drivers and kernel extensions that must be notified of an impending processor deallocation must 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 that 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 5.2 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 system call argument is a pointer to a structure describing the kernel extension's HAEH. This structure is defined in a header file, named **sys/high_avail.h**, as follows:

The private data field 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 function when the field 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 must be unique among all the registered kernel extensions. This name is listed 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: The following parameters call the HAEH routine:

- The value of the data field of the ha handler ext t structure passed to register HA handler.
- A pointer to a ha event t structure defined in the sys/high avail.h file as:

```
typedef struct {
                                     /* High-availability related event */
    uint magic;
                                    /* Identifies the kind of the event */
                                    /* "CPUF" */
#define HA CPU FAIL 0x40505546
   union {
                                    /* Predictive processor failure */
        struct {
            cpu_t dealloc_cpu;
                                  /* CPU bind 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 returns one of the following codes, also defined in the sys/high avail.h file:

```
#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 reaction is highly application-dependent. To allow AIX to proceed with the deconfiguration, they must move the threads that are bound to the last bind 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 action (such as stopping them) and restart their timer requests when the threads are bound to a new CPU, for instance.

Canceling the Registration of a High-Availability Event Handler: To keep the system coherent and prevent system crashes, the kernel extensions that register an HAEH must cancel the registration when they are unconfigured and are going to be unloaded. The interface is as follows:

```
int unregister HA handler(ha handler ext t *)
```

This interface returns 0 in case of success. Any non-zero return value indicates an error.

For more information on the system call, see unregister_HA_handler in AIX 5L Version 5.2 Technical Reference: Kernel and Subsystems Volume 1.

Deallocating a Processor in the Test Environment

To test any of the modifications made in applications or kernel extensions to support this processor deallocation, use the following command 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 that 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 the **compare and swap** subroutine.

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 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 and are defined as follows:

Import Fences

The *import fence* is a special machine instruction that delays until all previously issued instructions are complete. When used in conjunction with a lock, this prevents speculative execution of instructions until the lock is obtained.

Export Fences

The export fence guarantees that the data being protected is visible to all other processors prior to the lock being released.

To mask this complexity and provide independence from these machine-dependent instructions, the following subroutines are defined:

Conditionally updates a single word variable atomically, issuing an import fence for _check_lock

multiprocessor systems. The **compare_and_swap** subroutine is similar, but it does not issue

an import fence and, therefore, is not usable to implement a lock.

Atomically writes a single word variable, issuing an *export fence* for multiprocessor systems. _clear_lock

Kernel Programming

For complete details about kernel programming, see AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts. This section highlights the major differences 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. New or ported code should use the disable lock and unlock enable kernel services, which use simple locks in addition to interrupt control, instead of the i disable kernel service. These kernel services can also be used for uniprocessor systems, on which they simply use interrupt services without locking. For detailed information, see Locking Kernel Services in AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts.

Device drivers by default run in a logical uniprocessor environment, in what is called *funneled* 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. For detailed information, see Using Multiprocessor-Safe Timer Services in AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts.

Example of Locking Services

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
#define LOCK TAKEN
typdef struct {
        atomic_p
                        lock; /* lock word */
                        owner; /* identifies the lock owner */
        tid t
                        /* 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->lock, LOCK FREE);
        return SUCCESS;
} /* end of my mutex unlock */
```

Related Information

Locking Kernel Services in AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts

Using Multiprocessor-Safe Timer Services in AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts

Subroutine References

The bindprocessor, compare_and_swap, fork, pthread_mutex_lock, pthread_mutex_unlock subroutines.

Kernel Services

The disable_lock, i_disable, and unlock_enable kernel services.

Commands References

The **bindprocessor** command, **cpu_state** command.

Chapter 10. Multi-Threaded Programming

This chapter provides guidelines for writing multi-threaded programs using the threads library (**libpthreads.a**). The AIX threads library is based on the X/Open Portability Guide Issue 5 standard. For this reason, the following information presents the threads library as the AIX implementation of the XPG5 standard.

Parallel programming uses the benefits of multiprocessor systems, while maintaining a full binary compatibility with existing uniprocessor systems. The parallel programming facilities are based on the concept of threads.

The advantages of using parallel programming instead of serial programming techniques are as follows:

- · Parallel programming can improve the performance of a program.
- Some common software models are well-suited to parallel-programming techniques. For more information, see "Software Models" on page 192.

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.

The following information introduces threads and the associated programming facilities. It also discusses general topics concerning parallel programming:

Note: In this book, 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.

To learn how to write programs using multiple threads, read the topics in sequential order. If you are looking for specific information, see one of the following topics:

- "Thread-Safe and Threaded Libraries in AIX" on page 196
- "Synchronization Overview" on page 206
- "Threads Library Options" on page 247

Understanding Threads and Processes

A *thread* is an independent flow of control that operates within the same address space as other independent flows of controls within a process. Traditionally, 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.

The following sections discuss the differences between a thread and a process.

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.

Threads have some limitations and cannot be used for some special purposes that require multi-processed programs. For more information, see "Limitations" on page 274.

Process Properties

A process in a multi-threaded system is the changeable entity. It must be considered as an execution frame. It has 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, as follows:

- 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 control of flow. 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 errno error indicator. 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.

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.

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 (mutual exclusion locks) 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. 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 the number of slaves 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 iob at a time on a printer, while it also handles flow control and other printing details. The spooler may support job cancelation or other features that 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 an example of 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.

Kernel Threads and User Threads

A kernel thread is the schedulable entity, which means that the system scheduler handles kernel threads. These threads, known by the system scheduler, are strongly implementation-dependent. To facilitate the writing of portable programs, libraries provide 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 you are writing kernel extensions or device drivers. For more information about kernel programming, see AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts.

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 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 POSIX-standards compliant 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 VP behaves like a CPU. In the library, the VP 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. This model can be set by setting the **AIXTHREAD_SCOPE** environment variable to S.

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. This is the default model.

Threads Library API

This section provides general information about the threads library API. Although the following information is not required for writing multi-threaded programs, it can 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 and also allows implementation changes between two releases of AIX, necessitating only that programs be recompiled. 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 the following kinds of objects (opaque data types): threads, mutexes, rwlocks, and condition variables. These objects have attributes that specify the object properties. When creating an object, the attributes must be specified. In the threads library, these creation attributes are themselves objects, called *threads attributes objects*.

The following pairs of objects are manipulated by the threads library:

- · Threads and thread-attributes objects
- · Mutexes and mutex-attributes objects

- · Condition variables and condition-attributes objects
- · Read-write locks

An attributes object is created with attributes having default values. Attributes can then be individually modified by using subroutines. This ensures that a multi-threaded program will not be affected by the introduction of new attributes or by 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 is done by creating the object using the class attributes object.

Naming Convention for the Threads Library

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 type identifiers end with _t. Subroutine and macro names 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) that concerns 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.

pthread Implementation Files

The following AIX files provide the implementation of pthreads:

/usr/include/pthread.h /usr/include/sched.h /usr/include/unistd.h /usr/include/sys/limits.h /usr/include/sys/pthdebug.h /usr/include/sys/sched.h /usr/include/sys/signal.h

/usr/include/sys/types.h /usr/lib/libpthreads.a

/usr/lib/libpthreads_compat.a

/usr/lib/profiled/libpthreads.a

/usr/lib/profiled/libpthreads_compat.a

C/C++ header with most pthread definitions.

C/C++ header with some scheduling definitions. C/C++ header with **pthread_atfork()** definition.

C/C++ header with some pthread definitions.

C/C++ header with most pthread debug definitions.

C/C++ header with some scheduling definitions.

C/C++ header with pthread_kill() and

pthread_sigmask() definitions.

C/C++ header with some pthread definitions. 32-bit/64-bit library providing UNIX98 and POSIX

1003.1c pthreads.

32-bit only library providing POSIX 1003.1c Draft 7

pthreads.

Profiled 32-bit/64-bit library providing UNIX98 and

POSIX 1003.1c pthreads.

Profiled 32-bit only library providing POSIX 1003.1c

Draft 7 pthreads.

Thread-Safe and Threaded Libraries in AIX

By default, all applications are now considered "threaded," even though most are of the case "single threaded." These thread-safe libraries are as follows:

libbsd.a libc.a libm.a libsvid.a libtli.a libxti.a libnetsvc.a

POSIX Threads Libraries

The following POSIX threads libraries are available:

libpthreads.a POSIX Threads Library

The libpthreads.a library is 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 be ported for use with another POSIX threads library; only the performance and very few subroutines of the threads library are implementation-dependent. To enhance the portability of the threads library, the POSIX standard made the implementation of several programming facilities optional. For more information about checking the POSIX options, see "Threads Library Options" on page 247.

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 relinking. The libpthreads_compat.a library is only provided for backward compatibility with applications written using the Draft 7 of the POSIX Thread Standard. All new applications should use the libpthreads.a library, which supports both 32-bit and 64-bit applications. The libpthreads compat.a library only supports 32-bit applications. Beginning with AIX 5.1, the libpthreads.a library provides support for the Single UNIX Specification, Version 2 which includes the final POSIX 1003.1c Pthread Standard. For more information, see "Developing Multi-Threaded Programs" on page 261.

Creating Threads

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.

A thread has attributes, which specify the characteristics of the thread. 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. The object 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.

Creating and Destroying the Thread Attributes Object

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 can release 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 call the pthread_join subroutine 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. For more information on the pthread join subroutine, see "Calling the pthread join Subroutine" on page 225.

Other Threads Attributes

AIX also defines the following attributes, which are intended for advanced programs and may require special execution privilege to take effect. Most programs operate correctly with the default settings. The use of the following attributes is explained in "Using the inheritsched Attribute" on page 228.

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 the following stack attributes is explained in "Stack Attributes" on page 247.

Stacksize Specifies the size of the thread's stack Stackaddr Specifies the address of the thread's stack

Specifies the size of the guard area of the thread's stack Guardsize

Creating a Thread using the pthread_create Subroutine

A thread is created 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 following 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 the following:
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. For more information about using this pointer, see "Returning Information from a Thread" on page 226.

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. For more information, see "Example of a Multi-Threaded Program" on page 264.

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.

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, using a mechanism called cancelation. Because all threads share the same data space, a thread must perform cleanup operations at termination time; the threads library provides cleanup handlers for this purpose.

Exiting a Thread

A process can exit at any time when a thread calls 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 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.

To avoid implicitly calling the **exit** subroutine, to use the **pthread exit** subroutine to exit a thread.

Exiting the initial thread (for example, by calling the pthread_exit subroutine from the main routine) does not terminate the process. It terminates only 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 is 0.

The following 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;
```

The **pthread_exit** subroutine releases any thread-specific data, including the thread's stack. Any data allocated on the stack becomes invalid, because 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, because they may be used by other threads.

Canceling a Thread

The thread cancelation 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 is being canceled) can hold cancelation requests pending in a number of ways and perform application-specific cleanup processing when the notice of cancelation is acted upon. When canceled, the thread implicitly calls the **pthread_exit((void *)-1)** subroutine.

The cancelation 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 cancelation request. Each thread controls its own cancelability state and type with the **pthread_setcancelstate** and **pthread_setcanceltype** subroutines.

The following possible cancelability states and cancelability types lead to three possible cases, as shown in the following table.

Cancelability State	Cancelability Type	Resulting Case
Disabled	Any (the type is ignored)	Disabled cancelability
Enabled	Deferred	Deferred cancelability
Enabled	Asynchronous	Asynchronous cancelability

The possible cases are described as follows:

- Disabled cancelability. Any cancelation 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.
- Deferred cancelability. Any cancelation request is set pending, until the thread reaches the next cancelation point. It is the default cancelability state.

This cancelability state ensures that a thread can be cancelled, but limits the cancelation to specific moments in the thread's execution, called cancelation points. A thread canceled on a cancelation 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, use cleanup handlers or disable cancelability within critical regions. For more information, see "Using Cleanup Handlers" on page 204.

· Asynchronous cancelability. Any cancelation 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. For more information about async-cancel safety, see "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 cancelation 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 cancelation 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 that you use asynchronous cancelability only if you are sure only to perform operations that do not hold resources and only to call library routines that are async-cancel safe.

The following subroutines are async-cancel safe; they ensure that cancelation will be handled correctly, 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 cancelation points by calling the pthread testcancel subroutine.

Cancelation Points

Cancelation points are points inside of certain subroutines where a thread must act on any pending cancelation request if deferred cancelability is enabled. All of these subroutines may block the calling thread or compute indefinitely.

An explicit cancelation point can also be created by calling the pthread_testcancel subroutine. This subroutine simply creates a cancelation point. If deferred cancelability is enabled, and if a cancelation request is pending, the request is acted upon and the thread is terminated. Otherwise, the subroutine simply returns.

Other cancelation 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 cancelation point. If they did, all functions calling these subroutines (and many functions do) would provide a cancelation point. Having too many cancelation 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. For more information about these subroutines, see "Using Mutexes" on page 206.

Cancelation points occur when a thread is executing the following functions:

aio suspend close fcntl creat fsync getmsg lockf getpmsg mq_receive mq_send msgrcv msgsnd msync nanosleep open pause poll pread

pthread_cond_timedwait pthread_join pthread_testcancel

putpmsq pwrite read readv select sem wait sigpause sigsuspend sigtimedwait sigwait sigwaitinfo sleep system tcdrain wait usleep wait3 waitid waitpid write

writev

A cancelation point can 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 diclose dlopen endgrent endpwent **fwprintf fwrite**

fwscanfgetcgetc_unlockedgetchargetchar_unlockedgetcwdgetdategetgrentgetgrgidgetgrgid_rgetgrnamgetgrnam_r

getgrgid_r getgrnam getlogin_r popen

printf putc putc_unlocked

putchar putchar_unlocked puts pututxline putw putwc readdir readdir r putwchar 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 seekdir scanf 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 Iseek mkstemp nftw openlog pclose opendir perror ungetc ungetwc unlink vfprintf vfwprintf vwprintf wprintf vprintf wscanf

The side effects of acting upon a cancelation 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 cancelation cleanup handlers are called.

Whenever a thread has cancelability enabled and a cancelation request has been made with that thread as the target and the thread calls the **pthread_testcancel** subroutine, the cancelation request is acted upon before the **pthread_testcancel** subroutine returns. If a thread has cancelability enabled and the thread has an asynchronous cancelation request pending and the thread is suspended at a cancelation point waiting for an event to occur, the cancelation request will be acted upon. However, if the thread is suspended at a cancelation point and the event that it is waiting for occurs before the cancelation request is acted upon, the sequence of events determines whether the cancelation request is acted upon or whether the request remains pending and the thread resumes normal execution.

Cancelation Example

In the following example, both "writer" threads are canceled after 10 seconds, and after they have written their message at least five times.

```
/* include file for pthreads - the 1st */
#include <pthread.h>
#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);
                return -1;
        rc = pthread create(&f th, NULL, Thread, (void *)f str);
        if (rc)
                return -1;
        /* sleeps a while */
        sleep(10);
        /* requests cancelation */
        pthread cancel(e_th);
        pthread_cancel(f_th);
        /* sleeps a bit more */
        sleep(10);
        pthread exit(NULL);
}
```

Timer and Sleep Subroutines

Timer routines execute in the context of the calling thread. 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.

Using Cleanup Handlers

Cleanup handlers provide 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; they are stored in a thread-specific LIFO (last-in, first-out) stack. Cleanup handlers 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 cancelation 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 can specify a pointer to some data that 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 can be stopped at any cancelation point. In that case, a cleanup handler is established to release the buffer.

```
/* 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 cancelation 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 cancelation 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 cancelation point. When popping the cleanup handler, the handler is executed, releasing 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 */ \
```

Adhere to the balancing rule for the pthread_cleanup_push and pthread_cleanup_pop subroutines to avoid compiler errors or 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, because 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_cancel	Requests the cancelation 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_create	Creates a new thread, initializes its attributes, and makes it runnable.
pthread_equal	Compares two thread IDs.

pthread_exit pthread_self pthread_setcancelstate pthread_setcanceltype pthread_testcancel Terminates the calling thread.
Returns the calling thread's ID.
Sets the calling thread's cancelability state.
Sets the calling thread's cancelability type.
Creates a cancelation point in the calling thread.

Synchronization Overview

One main benefit of using threads is the ease of using synchronization facilities. To effectively interact, threads must synchronize their activities. This includes:

- · Implicit communication through the modification of shared data
- Explicit communication by informing each other of events that have occurred

More complex synchronization objects can be built using the primitive objects. For more information, see "Creating Complex Synchronization Objects" on page 239.

The threads library provides the following synchronization mechanisms:

- Mutexes (See "Using Mutexes.")
- Condition variables (See"Using Condition Variables" on page 212.)
- · Read-write locks (See "Using Read-Write Locks" on page 217.)
- Joins (See "Joining Threads" on page 225.)

Although primitive, these powerful mechanisms can be used to build more complex mechanisms.

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.

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.

Creating and Destroying the Mutex Attributes Object

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 release 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:

The same attributes object can be used to create several mutexes. It can also be modified between mutex creations. When the mutexes are created, the attributes object can be destroyed without affecting the mutexes created with it.

Mutex Attributes

The following mutex attributes are defined:

Specifies the protocol used to prevent priority inversions for a mutex. This attribute **Protocol**

depends on either the priority inheritance or the priority protection POSIX option.

Process-shared Specifies the process sharing of a mutex. This attribute depends on the process sharing

POSIX option.

For more information on these attributes, see "Threads Library Options" on page 247 and "Synchronization Scheduling" on page 232.

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 following 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 the following:
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 might be a pointer or another data type.

A mutex must be created once. However, avoid calling the pthread_mutex_init subroutine more than once with the same *mutex* parameter (for example, in two threads concurrently executing the same code). Ensuring the uniqueness of a mutex creation can be done in the following ways:

- · Calling the pthread_mutex_init subroutine prior to the creation of other threads that will use this mutex; for example, in the initial thread.
- Calling the pthread_mutex_init subroutine within a one time initialization routine. For more information, see "One-Time Initializations" on page 234.
- Using a static mutex initialized by the PTHREAD_MUTEX_INITIALIZER static initialization macro; the mutex will have default attributes.

After the mutex is no longer needed, destroy it 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 to create another mutex. For example, the following code fragment is valid, although not very practical:

```
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.

Types of Mutexes

The type of mutex determines how the mutex behaves when it is operated on. The following types of mutexes exist:

PTHREAD_MUTEX_DEFAULT or PTHREAD_MUTEX_NORMAL

Results in a deadlock if the same pthread tries to lock it a second time using the **pthread_mutex_lock** subroutine without first unlocking it. This is the default type.

PTHREAD MUTEX ERRORCHECK

Avoids deadlocks by returning a non-zero value if the same thread attempts to lock the same mutex more than once without first unlocking the mutex.

PTHREAD MUTEX RECURSIVE

Allows the same pthread to recursively lock the mutex using the **pthread_mutex_lock** subroutine without resulting in a deadlock or getting a non-zero return value from **pthread_mutex_lock**. The same pthread has to call the **pthread_mutex_unlock** subroutine the same number of times as it called **pthread_mutex_lock** subroutine in order to unlock the mutex for other pthreads to use.

When a mutex attribute is first created, it has a default type of **PTHREAD_MUTEX_NORMAL**. After creating the mutex, the type can be changed using the **pthread_mutexattr_settype** API library call.

The following is an example of creating and using a recursive mutex type:

```
pthread mutex attr t
                        attr;
pthread_mutex_t
                        mutex:
pthread mutexattr settype(&attr, PTHREAD MUTEX RECURSIVE);
pthread mutex init(&mutex, &attr);
struct {
        int a;
        int b:
        int c;
} A;
f()
        pthread mutex lock(&mutex);
        A.a++;
        g();
        A.c = 0;
        pthread mutex unlock(&mutex);
}
g()
        pthread_mutex_lock(&mutex);
        A.b += A.a:
        pthread mutex unlock(&mutex);
}
```

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 under the following conditions:

- · 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 might block forever or return an error depending on the type of mutex.

The **pthread_mutex_trylock** subroutine acts like the **pthread_mutex_lock** subroutine without blocking the calling thread under the following conditions:

- · 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 under the following conditions:

- · 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 might return an error or unlock the mutex depending on the type of mutex. Unlocking the mutex is not recommended because mutexes are usually locked and unlocked by the same pthread.

Because locking does not provide a cancelation point, a thread blocked while waiting for a mutex cannot be canceled. Therefore, it is recommended that you use mutexes only for short periods of time, as in instances where you are protecting data from concurrent access. For more information, see "Cancelation Points" on page 201 and "Canceling a Thread" on page 200.

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. For more information about implementing long locks and writer-priority readers/writers locks with mutexes, see "Creating Complex Synchronization Objects" on page 239.

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**. For example:

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

There are a number of ways that a multi-threaded application can deadlock. Following are some examples:

- A mutex created with the default type, PTHREAD_MUTEX_NORMAL, cannot be relocked by the same pthread without resulting in a deadlock.
- An application can deadlock when locking mutexes in reverse order. For example, the following code fragment can 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);
```

• An application can deadlock in what is called *resource* deadlock. For example:

```
struct {
                pthread mutex t mutex;
                char *buf;
        } A;
struct {
                pthread mutex t mutex;
                char *buf;
        } B;
struct {
                pthread mutex t mutex;
                char *buf;
        } C;
use all buffers()
        pthread mutex lock(&A.mutex);
        /* use buffer A */
        pthread mutex lock(&B.mutex);
        /* use buffers B */
        pthread mutex lock(&C.mutex);
        /* use buffer C */
        /* All done */
        pthread mutex unlock(&C.mutex);
        pthread mutex unlock(&B.mutex);
        pthread mutex unlock(&A.mutex);
use_buffer_a()
        pthread mutex lock(&A.mutex);
        /* use buffer A */
        pthread_mutex_unlock(&A.mutex);
}
```

```
functionB()
{
        pthread_mutex_lock(&B.mutex);
        /* use buffer B */
        if (..some condition)
        {
            use_buffer_a();
        }
        pthread_mutex_unlock(&B.mutex);
}

/* Thread A */
use_all_buffers();

/* Thread B */
functionB();
```

This application has two threads, thread A and thread B. Thread B starts to run first, then thread A starts shortly thereafter. If thread A executes **use_all_buffers()** and successfully locks **A.mutex**, it will then block when it tries to lock **B.mutex**, because thread B has already locked it. While thread B executes **functionB** and some_condition occurs while thread A is blocked, thread B will now also block trying to acquire **A.mutex**, which is already locked by thread A. This results in a deadlock.

The solution to this deadlock is for each thread to acquire all the resource locks that it needs before using the resources. If it cannot acquire the locks, it must release them and start again.

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 must 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 in the previous example 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. 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. 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
```

In most cases, thread A and thread B 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. After 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.

Using Condition Variables

Condition variables allow threads to wait until some event or condition has occurred. A condition variable has attributes that specify the characteristics of the condition. Typically, a program uses the following 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. For more information about implementing long locks and semaphores with condition variables, see "Creating Complex Synchronization Objects" on page 239.

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. For more information about joins, see "Joining Threads" on page 225.

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.

Creating and Destroying the Condition Attributes Object

The condition 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 can release storage dynamically allocated by the pthread condattr_init subroutine, depending on the implementation of the threads library.

In the following example, a condition attributes object is created and initialized with default values, then used and finally destroyed:

```
pthread condattr t attributes;
                \overline{/*} 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

The following condition attribute is supported:

Process-shared

Specifies the process sharing of a condition variable. This attribute depends on the process sharing POSIX option.

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 following code fragment:

```
pthread cond t cond;
pthread condattr t attr;
pthread condattr init(&attr);
pthread cond ini\overline{t}(&cond, &attr);
pthread condattr destroy(&attr);
is equivalent to the following:
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. Avoid calling the pthread_cond_init subroutine more than once with the same condition parameter (for example, in two threads concurrently executing the same code). Ensuring the uniqueness of a newly created condition variable can be done in the following ways:

- · Calling the pthread cond init subroutine prior to the creation of other threads that will use this variable: for example, in the initial thread.
- · Calling the pthread_cond_init subroutine within a one-time initialization routine. For more information, see "One-Time Initializations" on page 234.
- Using a static condition variable initialized by the PTHREAD COND INITIALIZER static initialization macro; the condition variable will have default attributes.

After the condition variable is no longer needed, destroy it 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 to create another condition. For example, the following code fragment is valid, although not very practical:

```
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. A given condition variable can have only one mutex associated with it, but a mutex can be used for more than one condition variable. It is possible to bundle into a structure the condition, the mutex, and the condition variable, as shown in the following code fragment:

```
struct condition_bundle_t {
   };
```

For more information about using the condition predicate, see "Synchronizing Threads with Condition Variables" on page 216.

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 cancelation point, the only way to exit this deadlock is to cancel the blocked thread, if cancelability is enabled. For more information, see "Canceling a Thread" on page 200.

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 the following fields:

```
tv sec
              A long unsigned integer, specifying seconds
              A long integer, specifying nanoseconds
tv nsec
```

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 the pthread_cond_timedwait subroutine 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 waits 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 wday = 1;
                    /* this field can be omitted -
                        but it will really be a Monday! */
                     /* first day of the year */
date.tm yday = 0;
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 cancelation point, although the sleep is not indefinite. Thus, a sleeping thread can be canceled, whether or not the sleep has a timeout.

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 229) . It may happen on multiprocessor systems, or some non-AIX systems, that more than one thread is awakened. 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 awakened. 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.

You can avoid this kind of deadlock by checking the EBUSY error code returned by the pthread_cond_destroy subroutine when destroying the condition variable, as in the following code

```
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; for example, one of the awoken threads. For more information about the pthread_yield subroutine, see "Scheduling Threads" on page 228.

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. For more information about the sigwait subroutine, see "Signal Management" on page 243.

Synchronizing Threads with Condition Variables

Condition variables are used to wait until a particular condition predicate becomes true. This condition predicate is set by another thread, usually the one that signals the condition.

Condition Wait Semantics

A condition predicate must be protected by a mutex. When waiting for a condition, the wait subroutine (either the pthread cond wait or pthread cond timedwait subroutine) 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 reevaluate 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. Basic implementation of a condition wait is shown in the following code fragment:

```
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, 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 reevaluate the predicate when a timeout occurred to determine whether it should declare a timeout or should proceed anyway. It is recommended that you 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 done as soon 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 need not 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 that calls 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)</pre>
             /* 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. For more complex usage examples, see "Creating Complex Synchronization Objects" on page 239.

Using Read-Write Locks

In many situations, data is read more often than it is modified or written. In these cases, you can allow threads to read concurrently while holding the lock and allow only one thread to hold the lock when data is modified. A multiple-reader single-writer lock (or read-write lock) does this. A read-write lock is acquired either for reading or writing, and then is released. The thread that acquires the read-write lock must be the one that releases it.

Read-Write Attributes Object

The pthread rwlockattr init subroutine initializes a read-write lock attributes object (attr). The default value for all attributes is defined by the implementation. Unexpected results can occur if the pthread_rwlockattr_init subroutine specifies an already-initialized read-write lock attributes object.

The following examples illustrate how to call the **pthread rwlockattr init** subroutine with the **attr** object: pthread rwlockattr t attr;

and:

```
pthread rwlockattr init(&attr);
```

After a read-write lock attributes object has been used to initialize one or more read-write locks, any function affecting the attributes object (including destruction) does not affect any previously initialized read-write locks.

The pthread_rwlockattr_destroy subroutine destroys a read-write lock attributes object. Unexpected results can occur if the object is used before it is reinitialized by another call to the pthread rwlockattr init subroutine. An implementation can cause the pthread rwlockattr destroy subroutine to set the object referenced by the attr object to an invalid value.

Creating and Destroying Read-Write Locks

The pthread rwlock init subroutine initializes the read-write lock referenced by the rwlock object with the attributes referenced by the attr object. If the attr object is NULL, the default read-write lock attributes are used; the effect is the same as passing the address of a default read-write lock attributes object. Upon successful initialization, the state of the read-write lock becomes initialized and unlocked. After initialized, the lock can be used any number of times without being reinitialized. Unexpected results can occur if the call to the pthread rwlock init subroutine is called specifying an already initialized read-write lock, or if a read-write lock is used without first being initialized.

If the pthread_rwlock_init subroutine fails, the rwlock object is not initialized and the contents are undefined.

The pthread_rwlock_destroy subroutine destroys the read-write lock object referenced by the rwlock object and releases any resources used by the lock. Unexpected results can occur in any of the following situations:

- If the lock is used before it is reinitialized by another call to the pthread_rwlock_init subroutine.
- An implementation can cause the pthread_rwlock_destroy subroutine to set the object referenced by the rwlock object to an invalid value. Unexpected results can occur if pthread_rwlock_destroy is called when any thread holds the rwlock object.
- Attempting to destroy an uninitialized read-write lock results in unexpected results. A destroyed read-write lock object can be reinitialized using the pthread_rwlock_init subroutine. Unexpected results can occur if the read-write lock object is referenced after it has been destroyed.

In cases where default read-write lock attributes are appropriate, use the PTHREAD RWLOCK INITIALIZER macro to initialize read-write locks that are statically allocated. For example:

The effect is similar to dynamic initialization using a call to the pthread_rwlock_init subroutine with the attr parameter specified as NULL, except that no error checks are performed. For example: pthread rwlock init(&rwlock2, NULL);

The following example illustrates how to use the pthread_rwlock_init subroutine with the attr parameter initialized. For an example of how to initialize the attr parameter, see "Read-Write Attributes Object."

Locking a Read-Write Lock Object for Reading

The pthread_rwlock_rdlock subroutine applies a read lock to the read-write lock referenced by the rwlock object. The calling thread acquires the read lock if a writer does not hold the lock and there are no writers blocked on the lock. It is unspecified whether the calling thread acquires the lock when a writer does not hold the lock and there are writers waiting for the lock. If a writer holds the lock, the calling thread will not acquire the read lock. If the read lock is not acquired, the calling thread does not return from the pthread_rwlock_rdlock call until it can acquire the lock. Results are undefined if the calling thread holds a write lock on the **rwlock** object at the time the call is made.

A thread may hold multiple concurrent read locks on the rwlock object (that is, successfully call the pthread rwlock rdlock subroutine n times). If so, the thread must perform matching unlocks (that is, it must call the **pthread rwlock unlock** subroutine *n* times).

The pthread rwlock tryrdlock subroutine applies a read lock similar to the pthread rwlock rdlock subroutine with the exception that the subroutine fails if any thread holds a write lock on the rwlock object or there are writers blocked on the rwlock object. Results are undefined if any of these functions are called with an uninitialized read-write lock.

If a signal is delivered to a thread waiting for a read-write lock for reading, upon return from the signal handler, the thread resumes waiting for the read-write lock for reading as if it was not interrupted.

Locking a Read-Write Lock Object for Writing

The pthread_rwlock_wrlock subroutine applies a write lock to the read-write lock referenced by the rwlock object. The calling thread acquires the write lock if no other thread (reader or writer) holds the read-write lock on the rwlock object. Otherwise, the thread does not return from the pthread rwlock wrlock call until it can acquire the lock. Results are undefined if the calling thread holds the read-write lock (whether a read or write lock) at the time the call is made.

The pthread_rwlock_trywrlock subroutine applies a write lock similar to the pthread_rwlock_wrlock subroutine, with the exception that the function fails if any thread currently holds rwlock for reading or writing. Results are undefined if any of these functions are called with an uninitialized read-write lock.

If a signal is delivered to a thread waiting for a read-write lock for writing, upon return from the signal handler, the thread resumes waiting for the read-write lock for writing as if it was not interrupted.

Sample Read-Write Lock Programs

The following sample programs demonstrate how to use locking subroutines. To run these programs, you need the check.h file and makefile.

check.h file:

```
#include stdio.h
#include stdio.h
#include stdio.h
#include stdio.h
/* Simple function to check the return code and exit the program
   if the function call failed
static void compResults(char *string, int rc) {
  if (rc) {
    printf("Error on : %s, rc=%d",
           string, rc);
```

Single-Thread Example

The following example uses the **pthread_rwlock_tryrdlock** subroutine with a single thread. For an example of using the **pthread_rwlock_tryrdlock** subroutine with multiple threads, see "Multiple-Thread Example" on page 221.

```
Example: test01.c
#define MULTI THREADED
#include pthread.h
#include stdio.h
#include "check.h"
                       rwlock = PTHREAD_RWLOCK_INITIALIZER;
pthread_rwlock_t
void *rdlockThread(void *arg)
 int
                  rc;
 int
                  count=0;
 printf("Entered thread, getting read lock with mp wait\n");
 Retry:
  rc = pthread_rwlock_tryrdlock(&rwlock);
  if (rc == EB\overline{U}SY) {
   if (count >= 10) {
      printf("Retried too many times, failure!\n");
      exit(EXIT_FAILURE);
    }
   ++count;
    printf("Could not get lock, do other work, then RETRY...\n");
    sleep(1);
   goto Retry;
 compResults("pthread_rwlock_tryrdlock() 1\n", rc);
 sleep(2);
 printf("unlock the read lock\n");
 rc = pthread_rwlock_unlock(&rwlock);
 compResults("pthread_rwlock_unlock()\n", rc);
```

```
printf("Secondary thread complete\n");
  return NULL;
int main(int argc, char **argv)
  int
                        rc=0;
  pthread t
                        thread;
  printf("Enter Testcase - %s\n", argv[0]);
  printf("Main, get the write lock\n");
  rc = pthread rwlock wrlock(&rwlock);
  compResults("pthread_rwlock_wrlock()\n", rc);
  printf("Main, create the try read lock thread\n");
  rc = pthread create(&thread, NULL, rdlockThread, NULL);
  compResults("pthread create\n", rc);
  printf("Main, wait a bit holding the write lock\n");
  sleep(5);
  printf("Main, Now unlock the write lock\n");
  rc = pthread rwlock unlock(&rwlock);
  compResults("pthread rwlock unlock()\n", rc);
  printf("Main, wait for the thread to end\n");
  rc = pthread_join(thread, NULL);
  compResults("pthread_join\n", rc);
  rc = pthread rwlock destroy(&rwlock);
  compResults("pthread_rwlock_destroy()\n", rc);
  printf("Main completed\n");
  return 0;
The output for this sample program will be similar to the following:
Enter Testcase - ./test01
Main, get the write lock
Main, create the try read lock thread
Main, wait a bit holding the write lock
Entered thread, getting read lock with mp wait
Could not get lock, do other work, then RETRY...
Could not get lock, do other work, then RETRY...
Could not get lock, do other work, then RETRY...
Could not get lock, do other work, then RETRY...
Could not get lock, do other work, then RETRY...
Main, Now unlock the write lock
Main, wait for the thread to end
unlock the read lock
Secondary thread complete
Main completed
```

Multiple-Thread Example

The following example uses the **pthread_rwlock_tryrdlock** subroutine with multiple threads. For an example of using the **pthread_rwlock_tryrdlock** subroutine with a single thread, see "Single-Thread Example" on page 220.

```
Example: test02.c

#define _MULTI_THREADED
#include pthread.h
#include stdio.h
#include "check.h"
```

```
pthread rwlock t
                       rwlock = PTHREAD RWLOCK INITIALIZER;
void *wrlockThread(void *arg)
 int
                  rc;
  int
                  count=0;
 printf("%.8x: Entered thread, getting write lock\n",
         pthread_self());
 Retry:
  rc = pthread_rwlock_trywrlock(&rwlock);
  if (rc == EBUSY) {
   if (count >= 10) {
     printf("%.8x: Retried too many times, failure!\n",
             pthread self());
      exit(EXIT FAILURE);
   ++count;
    printf("%.8x: Go off an do other work, then RETRY...\n",
           pthread self());
   sleep(1);
   goto Retry;
 compResults("pthread rwlock trywrlock() 1\n", rc);
 printf("%.8x: Got the write lock\n", pthread_self());
 sleep(2);
 printf("%.8x: Unlock the write lock\n",
         pthread self());
 rc = pthread rwlock unlock(&rwlock);
 compResults("pthread_rwlock_unlock()\n", rc);
  printf("%.8x: Secondary thread complete\n",
         pthread_self());
  return NULL;
int main(int argc, char **argv)
 int
 pthread t
                        thread, thread2;
 printf("Enter Testcase - %s\n", argv[0]);
 printf("Main, get the write lock\n");
  rc = pthread rwlock wrlock(&rwlock);
 compResults("pthread_rwlock_wrlock()\n", rc);
 printf("Main, create the timed write lock threads\n");
  rc = pthread create(&thread, NULL, wrlockThread, NULL);
  compResults("pthread_create\n", rc);
  rc = pthread create(&thread2, NULL, wrlockThread, NULL);
  compResults("pthread create\n", rc);
  printf("Main, wait a bit holding this write lock\n");
 sleep(1);
 printf("Main, Now unlock the write lock\n");
  rc = pthread rwlock unlock(&rwlock);
 compResults("pthread_rwlock_unlock()\n", rc);
 printf("Main, wait for the threads to end\n");
  rc = pthread join(thread, NULL);
 compResults("pthread join\n", rc);
```

```
rc = pthread join(thread2, NULL);
 compResults("pthread join\n", rc);
  rc = pthread rwlock destroy(&rwlock);
  compResults("pthread rwlock destroy()\n", rc);
 printf("Main completed\n");
 return 0;
The output for this sample program will be similar to the following:
Enter Testcase - ./test02
Main, get the write lock
Main, create the timed write lock threads
Main, wait a bit holding this write lock
00000102: Entered thread, getting write lock
00000102: Go off an do other work, then RETRY...
00000203: Entered thread, getting write lock
00000203: Go off an do other work, then RETRY...
Main, Now unlock the write lock
Main, wait for the threads to end
00000102: Got the write lock
00000203: Go off an do other work, then RETRY...
00000203: Go off an do other work, then RETRY...
00000102: Unlock the write lock
00000102: Secondary thread complete
00000203: Got the write lock
00000203: Unlock the write lock
00000203: Secondary thread complete
Main completed
```

Read-Write Read-Lock Example

The following example uses the **pthread_rwlock_rdlock** subroutine to implement read-write read locks:

```
Example: test03.c
#define MULTI THREADED
#include pthread.h
#include stdio.h
#include "check.h"
pthread rwlock t
                       rwlock;
void *rdlockThread(void *arg)
 int rc;
 printf("Entered thread, getting read lock\n");
  rc = pthread_rwlock_rdlock(&rwlock);
  compResults("pthread rwlock rdlock()\n", rc);
  printf("got the rwlock read lock\n");
 sleep(5);
 printf("unlock the read lock\n");
  rc = pthread_rwlock_unlock(&rwlock);
  compResults("pthread rwlock unlock()\n", rc);
  printf("Secondary thread unlocked\n");
  return NULL;
void *wrlockThread(void *arg)
  int rc;
 printf("Entered thread, getting write lock\n");
```

```
rc = pthread rwlock wrlock(&rwlock);
 compResults("pthread rwlock wrlock()\n", rc);
 printf("Got the rwlock write lock, now unlock\n");
 rc = pthread rwlock unlock(&rwlock);
 compResults("pthread rwlock unlock()\n", rc);
 printf("Secondary thread unlocked\n");
 return NULL;
int main(int argc, char **argv)
 pthread t
                        thread, thread1;
 printf("Enter Testcase - %s\n", argv[0]);
 printf("Main, initialize the read write lock\n");
  rc = pthread rwlock init(&rwlock, NULL);
 compResults("pthread rwlock init()\n", rc);
 printf("Main, grab a read lock\n");
 rc = pthread rwlock rdlock(&rwlock);
 compResults("pthread_rwlock_rdlock()\n",rc);
  printf("Main, grab the same read lock again\n");
  rc = pthread rwlock rdlock(&rwlock);
 compResults("pthread_rwlock_rdlock() second\n", rc);
 printf("Main, create the read lock thread\n");
  rc = pthread create(&thread, NULL, rdlockThread, NULL);
  compResults("pthread create\n", rc);
 printf("Main - unlock the first read lock\n");
  rc = pthread_rwlock_unlock(&rwlock);
  compResults("pthread rwlock unlock()\n", rc);
 printf("Main, create the write lock thread\n");
 rc = pthread create(&thread1, NULL, wrlockThread, NULL);
 compResults("pthread create\n", rc);
  sleep(5);
 printf("Main - unlock the second read lock\n");
  rc = pthread rwlock unlock(&rwlock);
 compResults("pthread_rwlock_unlock()\n", rc);
  printf("Main, wait for the threads\n");
  rc = pthread join(thread, NULL);
 compResults("pthread_join\n", rc);
  rc = pthread_join(thread1, NULL);
 compResults("pthread join\n", rc);
  rc = pthread rwlock destroy(&rwlock);
  compResults("pthread rwlock destroy()\n", rc);
 printf("Main completed\n");
 return 0;
The output for this sample program will be similar to the following:
$ ./test03
Enter Testcase - ./test03
Main, initialize the read write lock
```

Main, grab a read lock Main, grab the same read lock again Main, create the read lock thread Main - unlock the first read lock Main, create the write lock thread Entered thread, getting read lock got the rwlock read lock Entered thread, getting write lock Main - unlock the second read lock Main, wait for the threads unlock the read lock Secondary thread unlocked Got the rwlock write lock, now unlock Secondary thread unlocked Main completed

Joining Threads

Joining a thread means waiting for it to terminate, which can be seen as a specific usage of condition variables.

Waiting for a Thread

Using the pthread_join subroutine allows a thread to wait for another thread to terminate. More complex conditions, such as waiting for multiple threads to terminate, can be implemented by using condition variables. For more information, see "Synchronizing Threads with Condition Variables" on page 216.

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. After a target thread has been joined, it is automatically detached, and its storage can be reclaimed.

The following table indicates the 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 because a deadlock would occur and it is detected by the library. However, two threads may try to join each other. They will deadlock, but this situation is not detected by the library.

Multiple Joins

Several threads can 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

In the following example, the program ends after exactly five messages display in each language. 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)
        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 199). 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 implementation in the following example. In this example, the initial thread creates one thread per file to scan, each thread having the same entry point routine. The initial thread 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 each buffer and releases it.

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 200). 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. The pointer must still be valid after the thread was terminated and its storage reclaimed. Therefore, avoid returning a value, because the destructor routine is called when the thread's storage is reclaimed. For more information, see "Thread-Specific Data" on page 235.

Returning a pointer to dynamically allocated storage to several threads needs special consideration. Consider the following code fragment:

```
void *returned_data;
...
pthread_join(target_thread, &returned_data);
/* retrieves information from returned_data */
free(returned data);
```

The **returned_data** pointer is freed when it is executed by only one thread. If several threads execute the above code fragment concurrently, the **returned_data** pointer is freed several times; a situation that must be avoided. To prevent this, use a mutex-protected flag to signal that the **returned_data** pointer was freed. The following line from the previous example:

```
free(returned data);
```

would be replaced by the following lines, where a mutex can be used for locking the access to the critical region (assuming the *flag* variable is initially 0):

Locking access to the critical region ensures that the **returned_data** pointer is freed only once. For more information, see "Using Mutexes" on page 206.

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.

Scheduling Threads

Threads can be scheduled, and 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. Each thread has its own set of scheduling parameters. These parameters can be set using the thread attributes object before the thread is created. The parameters can also be dynamically set during the thread's execution.

Controlling the scheduling of a thread can be 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.

The threads library allows the programmer to control the execution scheduling of the threads in the following 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 (known as synchronization scheduling)
- · By dynamically changing the scheduling of a thread during synchronization operations (known as synchronization scheduling)

For more information about synchronization scheduling, see "Synchronization Scheduling" on page 232.

Scheduling Parameters

A thread has the following scheduling parameters:

The contention scope of a thread is defined by the thread model used in the threads library. For more scope information about contention scope, see "Contention Scope and Concurrency Level" on page 231. policy The scheduling policy of a thread defines how the scheduler treats the thread after 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 CPU-intensive. Thus, the threads library provides default values that are sufficient for most cases.

Using the inheritsched Attribute

The **inheritsched** attribute of the thread attributes object specifies how the thread's scheduling attributes will be defined. The following values are valid:

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** attribute must first be set to PTHREAD EXPLICIT SCHED. Otherwise, the attributes-object scheduling attributes are ignored.

Scheduling Policy and Priority

The threads library provides the following 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.

Round-robin (RR) scheduling. Each thread has a fixed priority; when multiple threads have the SCHED_RR

same priority level, they run for a fixed time slice in FIFO order.

Default AIX scheduling. Each thread has an initial priority that is dynamically modified by the SCHED OTHER

scheduler, according to the thread's activity; thread execution is time-sliced. On other systems,

this scheduling policy may be different.

The priority is an integer value, in the range from 1 to 127, where 1 is the least-favored priority, and 127 is the most-favored priority. Priority level 0 cannot be used: it is reserved for the system.

Note: 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 priority. 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. This structure contains the following fields:

sched priority Specifies the priority.

sched policy This field is ignored by the threads library. Do not use.

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 previously defined scheduling policies. The current value of the schedpolicy attribute of a thread attributes object can be obtained by using 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);
```

For more information about the **inheritsched** attribute, see "Using the inheritsched Attribute" on page 228.

Setting the Scheduling Attributes at Execution Time

The pthread_getschedparam subroutine returns the schedpolicy and schedparam attributes of a thread. 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 can be scheduled immediately at the end of the subroutine call.

For example, consider a thread T that is currently running with round-robin 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 the pthread setschedparam subroutine, the target thread will be scheduled immediately if it is the highest priority runnable thread.

Note: Both subroutines use 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 the subroutines then ignore the sched policy field.

Scheduling-Policy Considerations

Applications should use the default scheduling policy, unless a specific application requires the use of a fixed-priority scheduling policy. Consider the following points about using the nondefault policies:

- · Using the round-robin 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 must 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 round-robin policy may not always be allowed.

sched_yield Subroutine

The sched yield subroutine is the equivalent for threads of the yield subroutine. The sched yield subroutine forces the calling thread to relinquish the use of its processor and gives other threads an opportunity to be scheduled. The next scheduled thread may belong to the same process as the calling thread or to another process. Do not use the **yield** subroutine in a multi-threaded program.

The interface **pthread yield** subroutine is not available in Single UNIX Specification, Version 2.

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 pthread_getschedparam Returns the value of the schedpolicy and schedparam attributes of a thread.

Forces the calling thread to relinquish use of its processor.

sched_yield

Contention Scope and Concurrency Level

The *contention scope* of a user thread defines how it is mapped to a kernel thread. The threads library defines the following contention scopes:

PTHREAD SCOPE PROCESS

Process contention scope, sometimes called *local contention scope*. Specifies that the thread will be scheduled against all other local contention scope threads in the process. 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 an M:1 thread model have process contention scope.

PTHREAD SCOPE SYSTEM

System contention scope, sometimes called *global contention scope*. Specifies that the thread will be scheduled against all other threads in the system and is directly mapped to one kernel thread. All user threads in a 1:1 thread model have system 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 virtual processors 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.

For more information about thread models, see "Thread Models and Virtual Processors" on page 194.

Setting the Contention Scope

The contention scope can only be set before a thread is created 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 **TestImplementation** routine could be written as follows:

Impacts of Contention Scope on Scheduling

The contention scope of a thread influences its scheduling. Each 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 round-robin). The following code will always return the EPERM error code if the calling thread has system contention scope but does not have root authority. This code would not fail, if the calling thread had process contention scope.

```
schedparam.sched priority = 3;
pthread setschedparam(pthread self(), SCHED FIFO, schedparam);
```

Note: Root authority is not required to control the scheduling parameters of user threads having process contention scope.

Local user threads 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.

Synchronization Scheduling

Programmers can 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 the mutex protocols facility to avoid priority inversions.

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. Some implementations of the threads library do not provide synchronization scheduling.

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. In this example, the case of a uniprocessor system is considered. Priority inversions also occur on multiprocessor systems in a similar way.

In our example, a mutex M is used to protect some common data. Thread A has a priority level of 100 and is scheduled very often. Thread B has a priority level of 20 and is a background thread. Other threads in the process have priority levels near 60. A code fragment from thread A is as follows:

```
/* 1 */
pthread mutex lock(&M);
pthread_mutex_unlock(&M);
```

A code fragment from thread *B* is as follows:

```
pthread mutex lock(&M);
                               /* 2 */
fprintf(...);
                                /* 3 */
pthread mutex unlock(&M);
```

Consider the following execution chronology. Thread B is scheduled and executes line 2. While 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, the following mutex protocols are provided by the threads library:

Priority inheritance protocol

Sometimes called basic 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

Sometimes called priority ceiling protocol emulation. 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.

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.

Choosing a Mutex Protocol

The choice of a mutex protocol is made by setting attributes when creating a mutex. The mutex protocol is controlled through the protocol attribute. This attribute can be set in the mutex attributes object by 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, by using the pthread mutexattr getprioceiling and pthread mutexattr setprioceiling subroutines.

The prioceiling attribute of a mutex can also be dynamically controlled by using the pthread mutex getprioceiling and pthread mutex setprioceiling subroutines. 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. For more information about the priority inheritance and the priority protection POSIX options, see "Threads Library Options" on page 247.

Inheritance or Protection

Both protocols are similar and result in promoting the priority of the thread holding the mutex. If both protocols are available, programmers must choose a protocol. The choice depends on whether the priorities of the threads that will lock the mutex are available to the programmer who is 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, as follows:

- 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 Synchronization Subroutines

pthread mutex destroy pthread mutex init PTHREAD MUTEX INITIALIZER pthread_mutex_lock or pthread_mutex_trylock pthread_mutex_unlock pthread_mutexattr_destroy pthread mutexattr init pthread_cond_destroy pthread_cond_init PTHREAD_COND_INITIALIZER pthread_cond_signal or pthread_cond_broadcast pthread_cond_wait or pthread_cond_timedwait

pthread_condattr_destroy pthread_condattr_init

Deletes a mutex.

Initializes a mutex and sets its attributes. Initializes a static mutex with default attributes.

Locks a mutex. Unlocks a mutex.

Deletes a mutex attributes object.

Creates a mutex attributes object and initializes it with default values.

Deletes a condition variable.

Initializes a condition variable and sets its attributes. Initializes a static condition variable with default attributes.

Unblocks one or more threads blocked on a condition.

Blocks the calling thread on a condition. Deletes a condition attributes object.

Creates a condition attributes object and initializes it with default values.

One-Time Initializations

Some C libraries are designed for dynamic initialization, in which 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, use the **pthread once** subroutine. Otherwise, library initialization must be accomplished by an explicit call to a library exported initialization function prior to any use of the library. The pthread_once subroutine also provides an alternative for initializing mutexes and condition variables.

One-Time Initialization Object

The uniqueness of the initialization is ensured by the one-time initialization object. 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 cancelation point. However, the initialization routine may provide cancelation 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 204.

One-time initialization routines can be used for initializing mutexes or condition variables or to perform dynamic initialization. In a multi-threaded library, the code fragment shown above (void init routine();) would be written 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 must 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 the

pthread key t data type. 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, which 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. Thread T4 has the value of 2 associated with the same key.

		Threads				
		T1	T2	Т3	T4	
Keys	K1	6	56	4	1	
	K2	87	21	0	9	
	К3	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

- 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 450 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 */
```

In our example, 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 the following 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 not have enough 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 that keys are created uniquely can be done in the following ways:

- · Using the one-time initialization facility. See "One-Time Initializations" on page 234.
- 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 when the thread terminates the buffer is freed, the free subroutine can be used as follows:

```
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 as follows:

```
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;
```

Destructor calls can be repeated up to four times.

Key Destruction

A thread-specific data key can be destroyed by calling the pthread key delete subroutine. The pthread_key_delete subroutine does not actually call the destructor routine for each thread having data. After a data key is destroyed, it can be reused by another call to the pthread key create subroutine. Thus, the pthread_key_delete subroutine 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 **pthread_getspecific** subroutine reads the value bound to the specified key and is specific to the calling thread; the **pthread_setspecific** subroutine sets the value.

Setting Successive Values

The value bound to a specific key should be a pointer, which can 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.

Using 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 releasing the storage allocated for thread-specific data, the pointer is set to **NULL**. Otherwise, the destructor routine might 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 released memory, an error can 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

Although it is possible to store values that are not pointers, it is not recommended for the following 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.

Creating Complex Synchronization Objects

The subroutines provided in the threads library can be used as primitives to build more complex synchronization objects.

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 can be held for a long time without affecting the performance of the program. Long locks should not be used if cancelability is enabled.

A long lock has the <code>long_lock_t</code> data type. It must be initialized by the <code>long_lock_init</code> routine. The <code>long_lock</code>, <code>long_trylock</code>, and <code>long_unlock</code> subroutine performs similar operations to the <code>pthread_mutex_lock</code>, <code>pthread_mutex_trylock</code>, and <code>pthread_mutex_unlock</code> subroutine.

The following example shows a typical use of condition variables. In this example, the lock owner is not checked. As a result, any thread can unlock any lock. Error handling and cancelation handling are not performed.

```
{
        pthread mutex lock(&ll->lock);
        11->wanted++;
        while(!11->free)
                pthread_cond_wait(&11->cond);
        11->wanted--;
        11->free = 0;
        pthread mutex unlock(&11->lock);
}
int long trylock(long lock t *11)
        int got_the_lock;
        pthread mutex lock(&ll->lock);
        got the lock = 11->free;
        if (got_the_lock)
                11-> free = 0;
        pthread_mutex_unlock(&ll->lock);
        return got_the_lock;
}
void long_unlock(long_lock_t *11)
        pthread mutex lock(&ll->lock);
        11->free = 1;
        if (11->wanted)
                pthread cond signal(&11->cond);
        pthread mutex unlock(&11->lock);
```

Semaphores

Traditional semaphores in UNIX systems are interprocess-synchronization facilities. For specific usage, you can implement interthread semaphores.

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 semaphore wait and semaphore post operations are respectively performed by the **sema_p** and **sema_v** routines.

In the following basic implementation, error handling is not performed, but cancelations are properly handled with cleanup handlers whenever required:

```
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 (\overline{sem} - \overline{sem} - \overline{sem})
                 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)</pre>
                 pthread cond signal(&sem->cond);
        pthread mutex unlock(&sem->lock);
}
```

The counter specifies the number of users that are allowed to use 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. The semaphore wait operation is cancelable, because the **pthread_cond_wait** subroutine provides a cancelation point.

Write-Priority Read/Write Locks

A write-priority read/write lock provides multiple threads with simultaneous read-only access to a protected resource, and a single thread with 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.

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.

In the following example, the lock owner is not checked. As a result, any thread can unlock any lock. Routines, such as the **pthread_mutex_trylock** subroutine, are missing and error handling is not performed, but cancelations are properly handled with cleanup handlers whenever required.

```
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);
1->lock_count = 0;
if (!rwl->wating writers)
        pthread cond broadcast(&rwl->rcond);
        pthread cond signal(&rwl->wcond);
pthread mutex unlock(&rwl->lock);
```

Readers are counted only. 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.

The locking routines are cancelable, because they call the pthread_cond_wait subroutine. Cleanup handlers are therefore registered before calling the subroutine.

Signal Management

Signals in multi-threaded processes are an extension of signals in traditional single-threaded programs. Signal management in multi-threaded processes is shared by the process and thread levels, and consists of the following:

- Per-process signal handlers
- · Per-thread signal masks
- Single delivery of each signal

Signal Handlers and Signal Masks

Signal handlers are maintained at process level. It is strongly recommended to use the sigwait subroutine when waiting for signals. The **sigaction** subroutine is not recommended because the list of signal handlers is maintained at process level and 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 overrides the setting of the previous thread call; and in most cases, the order in which threads are scheduled cannot be predicted.

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; because unexpected behavior may result.

The pthread_sigmask subroutine is very similar to the sigprocmask subroutine. The parameters and usage of both subroutines are identical. When porting existing code to support the threads library, you can replace the **sigprocmask** subroutine with the **pthread sigmask** subroutine.

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. The following 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.

Note: Using **longjmp** from a signal handler can result in undefined behavior.

• No **pthread** routines can be called from a signal handler. Calling a **pthread** routine from a signal handler can lead to an application deadlock.

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 the awaited signal using the **sigwait** subroutine.

Typically, programs may create a dedicated thread to wait for asynchronously generated signals. Such a thread loops on a **sigwait** subroutine call and handles the signals. The following code fragment gives an example of such a signal-waiter thread:

```
#include <pthread>
static pthread mutex t mutex;
sigset_t set;
static int sig cond = 0;
void *run me(void *id)
       int sig;
       int err;
       err = sigwait(&set, &sig);
       if(err)
       {
              /* do error code */
       else
              printf("SIGINT caught\n");
              pthread mutex lock(&mutex);
              sig cond = 1;
              pthread_mutex_unlock(&mutex);
       }
       return;
```

```
main()
       pthread_t tid;
       sigemptyset(&set);
       sigaddset(&set, SIGINT);
       pthread sigmask(SIG BLOCK, &set, 0);
       pthread mutex init(&mutex, NULL);
       pthread create(&tid, NULL, run me, (void *)1);
       while(1)
              sleep(1);
              /* or so something here */
              pthread mutex lock(&mutex);
              if(sig cond)
                     /* do exit stuff */
                     return:
              pthread mutex unlock(&mutex);
      }
```

If more than one thread called the **sigwait** subroutine, exactly one call returns when a matching signal is sent. Which thread will be awakened cannot be predicted. Note that the sigwait subroutine provides a cancelation point.

Because a dedicated thread is not a real signal handler, it may signal a condition 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. For more information on condition variables, see "Using Condition Variables" on page 212.

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). Single-threaded programs may thus be rewritten as multi-threaded programs without changing their externally visible signal behavior.
 - For example, consider 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. 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.

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_killSends a signal to the specified thread.pthread_sigmaskSets the signal mask of a thread.

raise Sends a signal to the executing thread. Specifies the action to take upon delivery of a signal.

sigsuspend or 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.

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 section describes the interactions between threads and processes when duplicating and terminating a process.

Forking

Programmers call the fork subroutine in the following cases:

- To create a new flow of control within the same program. AIX creates a new process.
- 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 that the **fork** subroutine be used 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

The preceding forking rule 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 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 that 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. The following fork handlers exist:

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

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.

Threads Library Options

This section describes special attributes of threads, mutexes, and condition variables. 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.

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. The following information does not apply to the initial thread, which is created by the system.

Stack Size

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.

The stacksize attribute is defined in AIX. The following attribute and subroutines are available when the option is implemented:

- The **stacksize** attribute of the thread attributes object
- · The pthread attr getstacksize returns the value of the attribute
- · and pthread attr setstacksize subroutines sets the value

The default value of the **stacksize** attribute is 96 KB. The minimum value of the **stacksize** attribute for the 32-bit kernel is 8 KB and 16 KB for the 64-bit kernel. If the assigned value is less than the minimum value, the minimum value is allocated.

In the AIX implementation of the threads library, a chunk of data, called user thread area, is allocated for each created thread. The area is divided into the following sections:

- · A red zone, which is both read-protected and write-protected for stack overflow detection. There is no red zone in programs that use large pages.
- · A default stack.
- A pthread structure.
- A thread structure.
- · A thread attribute structure.

Note: The user thread area described here has no relationship to 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 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.

The following attribute and subroutines are available when the option is implemented:

- The **stackaddr** attribute of the thread attributes object 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 pthread attr setstackaddr subroutine sets the value.

If no stack address is specified, the stack is allocated by the system at an arbitrary address. If you must have the stack at a known location, you can use the stackaddr attribute. For example, if you need a very large stack, you can 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 attempts to allocate the stack at the given address. If it fails, the pthread_create subroutine returns EINVAL. Because the pthread attr setstackaddr subroutine does not actually allocate the stack, it only returns an error if the specified stack address exceeds the addressing space.

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. For more information about the scheduling properties of a thread, see "Scheduling Threads" on page 228.

The following attributes and subroutines 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 or at run time. Portable programs should check the availability of options before using them, so that they need not 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. Check option symbols in each program that might be ported to another system.

The following list indicates the symbol associated with each option:

_POSIX_THREAD_ATTR_STACKADDR Stack address _POSIX_THREAD_ATTR_STACKSIZE Stack size _POSIX_THREAD_PRIORITY_SCHEDULING Priority scheduling

_POSIX_THREAD_PRIO_INHERIT Priority inheritance _POSIX_THREAD_PRIO_PROTECT Priority protection Process sharing POSIX THREAD PROCESS SHARED

When an option is not available, you can 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 symbols that are 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

_SC_THREAD_PRIORITY_SCHEDULING Priority scheduling

_SC_THREAD_PRIO_INHERIT Priority inheritance _SC_THREAD_PRIO_PROTECT Priority protection SC THREAD PROCESS SHARED Process sharing

To check the general options, use the **sysconf** subroutine with the following *Name* parameter values:

_SC_REENTRANT_FUNCTIONS
SC THREADS

Denotes that reentrant functions are required.

Denotes the implementation of the threads library.

Process Sharing

AIX and most UNIX systems allow several processes to share a common data space, known as *shared memory*. For more information about the AIX shared memory facility, see "Understanding Memory Mapping" on page 379. 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, because there is no industry-standard interface for shared memory management, the process-sharing POSIX option is not implemented in the AIX threads library.

Threads Data Types

The following data types are defined for the threads library. The definition of these data types can vary between systems:

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

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:

- 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. If an application is compiled with the -D_LARGE_THREADS flag, the maximum number of
 threads per process is 32767.
- The minimum stack size for a thread is 8 K. The default stack size is 96 KB. This number can be retrieved at compilation time using the PTHREAD_STACK_MIN symbolic constant defined in the pthread.h header file.

Note: The maximum stack size is 256 MB, the size of a segment. This limit is indicated by the **PTHREAD_STACK_MAX** symbolic constant in the **pthread.h** header file.

 The maximum 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:

- The default value for the **DEFAULT_DETACHSTATE** symbolic constant is PTHREAD CREATE DETACHED, which specifies the default value for the detachstate attribute.
- The default value for the **DEFAULT_JOINABLE** symbolic constant is **PTHREAD_CREATE_JOINABLE**, which specifies the default value for the joinable state.
- The default value for the **DEFAULT INHERIT** symbolic constant is **PTHREAD INHERIT SCHED**, which specifies the default value for the inheritsched attribute.
- The default value for the **DEFAULT PRIO** symbolic constant is 1, which specifies the default value for the sched prio field of the **schedparam** attribute.
- · The default value for the **DEFAULT SCHED** symbolic constant is **SCHED OTHER**, which specifies the default value for the **schedpolicy** attribute of a thread attributes object.
- The default value for the **DEFAULT SCOPE** symbolic constant is **PTHREAD SCOPE LOCAL**, which specifies the default value for the **contention-scope** attribute.

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 one-time synchronization control structure.
pthread_setspecific	Sets the thread-specific data associated with the specified key.

Supported Interfaces

On AIX systems, the _POSIX_THREADS, _POSIX_THREAD_ATTR_STACKADDR, _POSIX_THREAD_ATTR_STACKSIZE and _POSIX_THREAD_PROCESS_SHARED symbols 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_init
- pthread_attr_setdetachstate
- 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
- 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
- sigwait

Single UNIX Specification, Version 2 Interfaces

The following is a list of Single UNIX Specification, Version 2 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 symbol 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_mutex_getprioceiling
- · pthread_mutex_setprioceiling
- · pthread_mutexattr_getprioceiling
- pthread_mutexattr_getprotocol
- pthread_mutexattr_setprioceiling
- pthread_mutexattr_setprotocol

Non-Thread-Safe Interfaces

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
- I64a
- Igamma
- localtime
- Irand48
- mrand48
- nl_langinfo
- ptsname
- putc_unlocked
- putchar_unlocked
- 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
- Igamma

None of the functions in the following libraries are thread-safe:

- · libPW.a
- · libblas.a
- · libcur.a
- · libcurses.a
- · libplot.a
- · libprint.a

The **ctermid** and **tmpnam** interfaces are not thread-safe if they are passed a NULL argument.

Note: Certain subroutines may be implemented as macros on some systems. Avoid using the address of threads subroutines.

Writing Reentrant and Thread-Safe Code

In single-threaded processes, only one flow of control exists. The code executed by these processes thus need not 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.

Reentrance and thread safety are both related to the way that functions handle resources. Reentrance and thread safety are separate concepts: a function can be either reentrant, thread-safe, both, or neither.

This section provides information about writing reentrant and thread-safe programs. It does not cover the topic of writing thread-efficient programs. Thread-efficient programs are efficiently parallelized programs. You must consider thread effiency 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.

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 language, 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, as in the following example:

```
/* 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. Global data 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 the following 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 must 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;
```

This function is not reentrant (nor thread-safe). To make the function reentrant by returning dynamically allocated data, the function would be similar to the following code fragment:

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 caller-provided storage. This is discussed in "Reentrant and Thread-Safe Libraries" on page 260.

Keeping Data over Successive Calls

No data should be kept over successive calls, because different threads may successively call the function. If a function must maintain some data over successive calls, such as a working buffer or a pointer, the caller should provide this data.

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;
    }
}
```

This function is not reentrant. To make it reentrant, the static data, the **index** variable, must be maintained by the caller. The reentrant version of the function could be implemented as in the following code fragment:

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:

Making a Function Thread-Safe

In multi-threaded programs, all functions called by multiple threads must be thread-safe. However, a workaround exists for using thread-unsafe subroutines in multi-threaded programs. 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:

To be thread-safe, the static variable **counter** must be protected by a static lock, as in the following example:

```
/* pseudo-code thread-safe function */
int increment counter();
        static int counter = 0;
        static lock type counter lock = LOCK INITIALIZER;
        pthread mutex lock(counter lock);
        counter++;
        pthread mutex 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.

Workarounds for Thread-Unsafe Functions

It is possible to use a workaround to use thread-unsafe functions called by multiple threads. This can 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. The following are possible workarounds:

 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). This solution can create performance bottlenecks because only one thread can access any part of the library at any given time. The solution in the following pseudocode is acceptable only if the library is seldom accessed, or as an initial, quickly implemented workaround.

```
/* this is pseudo code! */
lock(library lock);
library call();
unlock(library lock);
lock(library lock);
x = library var;
unlock(library lock);
```

Use a lock for each library component (routine or global variable) or group of components. This solution is somewhat more complicated to implement than the previous example, 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.

```
/* this is pseudo-code! */
lock(library moduleA lock);
library_moduleA_call();
unlock(Tibrary moduleA lock);
lock(library moduleB lock);
x = library_moduleB var;
unlock(library moduleB lock);
```

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. 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 $_{\bf r}$ (underscore followed by the letter $_{\bf r}$).

When writing multi-threaded programs, use the reentrant versions of subroutines 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. Ensure the uniqueness of the thread using the library; otherwise, the program will have unexpected behavior, or may even stop.

Converting Libraries

Consider the following when converting an existing library to a reentrant and thread-safe library. This information applies only to C language libraries.

- Identify 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), and access (read and write) subroutines
 should be created.
- Identify 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 facility may be used. For more information, see "One-Time Initializations" on page 234.
- Identify non-reentrant functions and make them reentrant. For more information, see "Making a Function Reentrant" on page 257.
- Identify thread-unsafe functions and make them thread-safe. For more information, see "Making a Function Thread-Safe" on page 259.

Developing Multi-Threaded Programs

Developing multi-threaded programs is similar to developing programs with multiple processes. Developing programs also consists of compiling and debugging the code.

Compiling a Multi-Threaded Program

This section explains how to generate a multi-threaded program. It describes the following:

- · The required header file
- Invoking the compiler, which is used to generate multi-threaded programs.

Header File

All subroutine prototypes, macros, and other definitions for using the threads library are in the pthread.h header file, which is 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 that thread-safe subroutines are used. 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 I-value in a multi-threaded program.

Invoking the Compiler

When compiling a multi-threaded program, 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 Single UNIX Specification, Version 2. The POSIX Threads Specification 1003.1c is a subset of the Single UNIX Specification, Version 2.

The following libraries are automatically linked with your program when using the xlc r and cc r 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

Invoking the Compiler 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.

When compiling a multi-threaded program for Draft 7 support of threads, invoke the C compiler using one of the following commands:

Invokes the compiler with default language level of ansi xlc r7 cc_r7 Invokes the compiler with default language level of extended

The following libraries are automatically linked with your program when using the xlc_r7 and cc_r7 commands:

libpthreads_compat.a Draft 7 Compatibility Threads library To achieve source code compatibility, use 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, because the commands provide the necessary options. These options are provided for those who do not have the latest AIX compiler.

Porting Draft 7 Applications to the Single UNIX Specification, Version 2

Differences exist between Draft 7 and the final standard include:

- · 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.
- The default state when a pthread is created is joinable. This is a significant change because it can result in a memory leak if ignored. For more information about thread creation, see "Creating Threads" on page 196.
- · The default pthread scheduling parameter is scope. For more information about scheduling, see "Scheduling Threads" on page 228.
- The pthread_yield subroutine has been replaced by the sched_yield subroutine.
- 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 that a process can have depends on the memory model and the use of process address space, for other purposes. The amount of memory that 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 the pthread_attr_setstacksize subroutine and the size of the guard region with the pthread attr setguardsize subroutine.

Note: The soft limit on stack size imposed by the command ulimit -s applies only to the stack of the main thread of the application.

The following table indicates the maximum number of pthreads that 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 depend on other memory usage in the program. For a 64-bit process, the **ulimit** subroutine 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.

Data Model	-bmaxdata	Maximum Pthreads	
Small Data	n/a	1084	
Big Data	0x10000000	2169	
Big Data	0x20000000	4340	
Big Data	0x30000000	6510	
Big Data	0x4000000	8681	
Big Data	0x5000000	10852	
Big Data	0x6000000	13022	
Big Data	0x70000000	15193	
Big Data	0x80000000	17364	

The NUM_SPAREVP environment variable can be set to control the number of spare virtual processors that are maintained by the library. It is not necessary to modify this variable. In some circumstances,

applications that use only a few megabytes of memory can reduce memory overhead by setting the **NUM_SPAREVP** environment variable to a lower value. Typical settings include the number of CPUs on the system or the peak number of process threads. Setting this variable does not affect process performance. The default setting is 256.

Note: The NUM_SPAREVP environment variable is available only in AIX 5.1.

Example of a Multi-Threaded Program

The following short multi-threaded program displays "Hello!" in both English and French for five seconds. Compile with **cc_r** or **xlc_r**. For more information about compiling thread programs, see "Developing Multi-Threaded Programs" on page 261.

```
#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.

Debugging a Multi-Threaded Program

The following tools are available to debug multi-threaded programs:

- Application programmers can use the dbx command to perform debugging. Several subcommands are available for displaying thread-related objects, including attribute, condition, mutex, and thread.
- Kernel programmers can use the kernel debug program to perform debugging on kernel extensions and device drivers. The kernel debug program provides limited access to user threads, and primarily handles kernel threads. Several subcommands support multiple kernel threads and processors, including:
 - The cpu subcommand, which changes the current processor
 - The ppd subcommand, which displays per-processor data structures

- The **thread** subcommand, which displays thread table entries
- The uthread subcommand, which displays the uthread structure of a thread

For more information on the kernel debug program, see the AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts.

Core File Requirements of a Multi-Threaded Program

By default, processes do not generate a full core file. If an application must debug data in shared memory regions, particularly thread stacks, it is necessary to generate a full core dump. To generate full core file information, run the following command as root user:

chdev -1 sys0 -a fullcore=true

Each individual pthread adds to the size of the generated core file. The amount of core file space that a pthread needs includes the stack size, which the user can control with the pthread attr setstacksize subroutine. For pthreads created with the NULL pthread attribute, each pthread in a 32-bit process adds 128 KB to the size of the core file, and each pthread in a 64-bit process adds 256 KB to the size of the core file.

Developing Multi-Threaded Programs to Examine and Modify pthread Library Objects

The pthread debug library (libpthdebug.a) provides a set of functions that enable application developers 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 applications with 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 is in 64-bit format both for 64-bit and 32-bit applications. It is the application's 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 is ignored.

The pthread debug library does not report information on mutexes, mutex attributes, condition variables, condition variable attributes, read/write locks, and read/write lock attibutes 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 assigns a unique session identifier that must be passed as the first parameter to all other pthread debug library functions, except pthdb session pthreaded function, 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 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 Function. 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 obtain and write data, as well as to give storage management to the application. Required call back functions for an application are as follows:

read_data

Retrieves **pthread library** object information

alloc Allocates memory in the pthread debug library

realloc

Reallocates memory in the pthread debug library

dealloc

Frees allocated memory in the pthread debug library

Optional call back functions for an application are as follows:

read_regs

Necessary only for the pthdb_pthread_context and pthdb_pthread_setcontext subroutines write data

Necessary only for the pthdb_pthread_setcontext subroutine

write_regs

Necessary only for pthdb_pthread_setcontext subroutine

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 resets 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 obtains the context information, and the pthdb_pthread_setcontext function sets 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, the context information saved by the 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. The application must determine if the kernel thread is in kernel mode or user mode and then to provide the correct information for that mode.

When a pthread with kernel thread is in kernel mode, you cannot get the full user mode context because the kernel does not save it in one place. The **getthrds** function can be used to obtain part of this information, because it always saves the user mode stack. The application can discover this by checking the **thrdsinfo64.ti_scount** structure. If this is non-zero, the user mode stack is available in the **thrdsinfo64.ti_ustk** structure. From the user mode stack, it is possible to determine the instruction address register (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 that customize the session. These flags control the number of registers that are read or written during context operations.

The **pthdb_session_flags** function obtains the current flags for the session.

Terminating the Session

At the end of the session, the session data structures must be deallocated, and the session data must 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 of Connecting to the pthread Debug Library

The following example shows 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()
{
```

Developing Multi-Threaded Program Debuggers

The pthread debug library (**libpthdebug.a**) provides a set of functions that allows developers to provide debug capabilities for applications that use the pthread library.

The pthread debug 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 to examine pthread information of its own application. This library can be used by a multi-threaded debugger to debug a multi-threaded application. Multi-threaded debuggers are supported in the **libpthreads.a** library, which 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, because the ptrace facility is not supported 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 with 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 is in 64-bit format both for 64-bit and 32-bit applications. It is the debugger's 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 is ignored.

The pthread debug library does not report mutexes, mutex attributes, condition variables, condition variable attributes, read/write locks, and read/write lock attributes 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 that can be used to set a breakpoint for immediate notification that the pthread library has been initialized. Therefore, the pthdb session pthreaded function provides the following methods for determining when the pthread library has been initialized:

- The debugger calls the function each time the debug process stops, to see if the program that is being debugged is pthreaded.
- The debugger calls the function once and if the program that is being debugged is not pthreaded, sets a breakpoint to notify the debugger when the debug process is pthreaded.

After 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 call back function, 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 in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 1.

Call Back Functions

The pthread debug library uses call back functions to do the following:

- Obtain addresses and data
- Write data
- · Give storage management to the debugger
- · Aid debugging of the pthread debug library

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 resets 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 must support hold and unhold of threads for the following reasons:

- 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 through a subset of available threads, it must be possible to hold threads not in the set.

The following list of functions perform hold and unhold tasks:

- 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 or not a pthread has a kernel thread.

- 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 breakpoint 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 thread ID (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 the following 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 ensures that the same TID is stopped as before the commit operation.

To hold or unhold pthreads, use the 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. Select 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 that 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 must continue these kernel threads along with the others that 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 obtains the context information and the pthdb pthread setcontext function sets 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, the context information saved by the 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 debugger's responsibility to determine if the kernel thread is in kernel mode or user mode and then to provide the correct information for that mode.

When a pthread with kernel thread is in kernel mode, you cannot get the full user mode context because the kernel does not save it in one place. The **getthrds** function can be used to obtain part of this information, because it always saves the user mode stack. The debugger can discover this by checking the thrdsinfo64.ti scount structure. If this is non-zero, the user mode stack is available in the thrdsinfo64.ti_ustk structure. From user mode stack, it is possible to determine the instruction address register (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 debugger to change the flags that customize the session. These flags control the number of registers that are read or written to during context operations, and to control the printing of debug information.

The **pthdb** session flags function obtains the current flags for the session.

Terminating the Session

At the end of the debug session, the session data structures must be deallocated, and the session data must 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 of Hold/Unhold Functions

The following pseudocode example shows how the debugger uses 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 the program that is being debugged */
   /* the program that is being debugged uses ptrace()/ptracex() with PT TRACE ME */
  while (/* waiting on an event */)
     /* debugger waits on the program that is being debugged */
     if (pthreaded symbol==NULL) {
       rc = pthdb session pthreaded(user, &callbacks, pthreaded symbol);
       if (rc == \overline{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);
                          /* 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);
                        /* handle error and exit */
           case DB EXIT:
rc = pthdb_session_destroy(session);
/* other clean up code */
exit(0);
   }
  exit(0);
```

Benefits of Threads

Multi-threaded programs can improve performance compared to traditional parallel programs that use multiple processes. Furthermore, improved 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 64 KB, 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 library for managing processes. Thread creation requires only 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. Protect shared data from concurrent access by using mutexes or other synchronization tools.

Synchronization facilities provided by the threads library ease implementation of flexible and powerful synchronization tools. These tools can replace traditional inter-process communication facilities, such as message queues. 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 can 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.

Related Information

 AIX 5L Version 5.2 Performance Management Guide provides a performance overview of the CPU Scheduler.

Subroutine References

- · Session Functions in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 1
 - pthdb_session_commit_tid
 - pthdb_session_committed
 - pthdb session concurrency

- pthdb_session_continue_tid
- pthdb_session_destroy
- pthdb_session_flags
- pthdb_session_init
- pthdb_session_pthreaded
- pthdb_session_setflags
- pthdb_session_stop_tid
- pthdb_session_update
- List Functions in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 1
 - pthdb attr
 - pthdb_cond
 - pthdb_condattr
 - pthdb key
 - pthdb_mutex
 - pthdb_mutexattr
 - pthdb pthread
 - pthdb_pthread_key
 - pthdb_rwlock
 - pthdb rwlockattr
- Pthread Functions in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 1
 - 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 in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 1
 - pthdb_pthread_context
 - pthdb_pthread_setcontext
- Pthread Signal Functions in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 1
 - pthdb_pthread_sigmask
 - pthdb_pthread_sigpend

- pthdb_pthread_sigwait
- Pthread Specific Data Functions in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 1
 - pthdb_pthread_specific
- Pthread Mapping to Kernel Thread Functions in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 1
 - pthdb_pthread_tid
 - pthdb_tid_pthread_tid
- · Pthread Hold/Unhold Functions
 - pthdb_pthread_hold
 - pthdb_pthread_holdstate
 - pthdb_pthread_unhold
- Attribute Functions in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 1
 - 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 in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 1
 - 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 in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 1
 - pthdb_mutexattr_addr
 - pthdb mutexattr prioceiling
 - pthdb_mutexattr_protocol
 - pthdb_mutexattr_pshared
 - pthdb_mutexattr_type
- Condition Variable Functions in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 1
 - pthdb_cond_addr
 - pthdb_cond_mutex

- pthdb_cond_pshared
- Condition Variable Attribute Functions in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 1
 - pthdb_condattr_addr
 - pthdb_condattr_pshared
- · Read/Write Lock Functions in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 1
 - pthdb_rwlock_addr
 - pthdb_rwlock_lock_count
 - pthdb_rwlock_owner
 - pthdb_rwlock_pshared
 - pthdb_rwlock_state
- Read/Write Lock Attribute Functions in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 1
 - pthdb_rwlockattr_addr
 - pthdb_rwlockattr_pshared
- Waiter Functions in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 1
 - pthdb mutex waiter
 - pthdb_cond_waiter
 - pthdb_rwlock_read_waiter
 - pthdb_rwlock_write_waiter

Files References

The pthread.h file.

Chapter 11. lex and yacc Program Information

For a program to receive input, either interactively or in a batch environment, you must provide another program or a routine to receive the input. Complicated input requires additional code to break the input into pieces that mean something to the program. You can use the **lex** and **yacc** commands to develop this type of input program.

The **lex** command 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. The **yacc** command 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. Together these commands generate a lexical analyzer and parser program for interpreting input and output handling.

The following topics are covered in this chapter:

- "Generating a Lexical Analyzer with the lex Command"
- "Using the lex Program with the yacc Program" on page 280
- "Extended Regular Expressions in the lex Command" on page 281
- · "Passing Code to the Generated lex Program" on page 284
- "Defining lex Substitution Strings" on page 285
- "lex Library" on page 285
- "Actions Taken by the Lexical Analyzer" on page 286
- "lex Program Start Conditions" on page 290
- "Creating a Parser with the yacc Program" on page 290
- "The vacc Grammar File" on page 291
- "Using the yacc Grammar File" on page 292
- "yacc Grammar File Declarations" on page 294
- "yacc Rules" on page 296
- "yacc Actions" on page 297
- "yacc Program Error Handling" on page 299
- "Parser Operation Generated by the vacc Command" on page 300
- "Using Ambiguous Rules in the yacc Program" on page 302
- "Turning on Debug Mode for a Parser Generated by the yacc Command" on page 303
- "Example Program for the lex and yacc Programs" on page 303
- "Related Information" on page 307

Generating a Lexical Analyzer 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 Action statements

Character patterns that the generated lexical analyzer recognizes. C language program fragments that define how the generated lexical analyzer reacts to extended regular expressions it recognizes.

For information about the format and logic allowed in this file, see the **lex** command in AIX 5L Version 5.2 Commands Reference. Volume 3.

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 produce an executable lexical analyzer:

```
cc lex.yy.c -11
```

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. For more information, see "Creating a Parser with the yacc Program" on page 290.

You can move a lex.yy.c output file to another system if it has a 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.

The lexical analyzer generated by the lex command uses an analysis method called a deterministic finite-state automaton. This method provides for a limited number of conditions in which the lexical analyzer can exist, along with the rules that determine the state of the lexical analyzer.

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 the 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.

Compiling the Lexical Analyzer

To compile a lex program, do the following:

- 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 -II 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 -11

Using the lex Program with the yacc Program

When used alone, the lex program generator produces 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 vacc command. The vacc 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 **vacc** 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 installed and its main program is used, the yacc program calls the yylex subroutine. In this case, where the appropriate token value is returned, each lex program rule should end with the following:

```
return(token);
```

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.vv.c"
```

The yacc library should be loaded before the lex library to obtain a main program that invokes the yacc parser. You can generate lex and yacc programs in either order.

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:

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

New line character (Do not use the actual new line character in an expression.) \n

\r Return

\t Tab

\v Vertical tab

\\ Backslash

\Digits The character whose encoding is represented by the one-digit, two-digit, or three-digit octal integer specified by the *Digits* string.

\xDigits

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 the **lex** command.

[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), ^ (caret), 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:

```
cntrl lower
alnum
                     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.

Period Character (.)

Matches any character except the new line character. In order for the period character (.) 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, the period character (.) 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 I (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 Expression is at the beginning of the line and the ^ (caret) operator is the first character in an expression.

Example: ^h matches an h at the beginning of a line.

Expression\$

Indicates a match only when Expression is at the end of the line and the \$ (dollar sign) operator is the last character in an expression.

Example: h\$ matches an h at the end of a line.

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 ^ (caret) 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 (see "lex Program Start Conditions" on page 290).

Example: If being at the beginning of a line is start condition ONE, then the ^ (caret) operator equals the expression <0NE>.

To use the operator characters as text characters, use one of the escape sequences: " " (double quotation marks) or \ (backslash). The " " operator indicates that what is enclosed is text. Thus, the following example matches the string xyz++:

xyz"++"

A portion of a string can be quoted. Quoting an ordinary text character has no effect. For example, the following expression is equivalent to the previous example:

"xyz++"

To ensure that text is interpreted as text, quote all characters that are not letters or numbers.

Another way to convert an operator character to 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 + +

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 following general format:

```
translation
name
```

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:

```
[0-9]
              [DEde] [-+] {D}+
F
```

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 Library

The **lex** library contains the following subroutines:

main() Invokes the lexical analyzer by calling the **yylex** subroutine.

Returns the value 1 when the end of input occurs. yywrap()

Appends the next matched string to the current value of the vytext array rather than yymore()

replacing the contents of the **yytext** array.

Retains *n* initial characters in the **yytext** array and returns the remaining characters to the yyless(int n)

input stream.

Allows the lexical analyzer to match multiple rules for the same input string. (The yyreject yyreject()

subroutine is called when the special action **REJECT** is used.)

Some of the **lex** subroutines can be substituted by user-supplied routines. For example, the **lex** command supports user-supplied versions of the **main** and **yywrap** subroutines. The library versions of these routines, provided as a base, are as follows:

main Subroutine

```
#include <stdio.h>
#include <locale.h>
main() {
    setlocale(LC_ALL, "");
    yylex();
    exit(0);
}

yywrap Subroutine

yywrap() {
    return(1);
}
```

The **yymore**, **yyless**, and **yyreject** subroutines are available only through the **lex** library. However, these subroutines are required only when used in **lex** command actions.

Actions Taken by the Lexical Analyzer

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

Same As Next Action

To avoid repeatedly writing the same action, use the I (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 that ignores blank, tab, and new line characters can also be written as follows:

```
"\t"
"\n" ;
```

The quotation marks that surround \n and \t are not required.

Printing a Matched String

To determine 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

The name of the array containing the data to be printed yytext

The name of the array containing the multibyte type (wchar_t) data to be printed yywtext

The **lex** command defines **ECHO**; as a special action to print the contents of **yytext**. For example, the following two rules are equivalent:

```
[a-z]+
             ECHO;
             printf("%s",yytext);
[a-z]+
```

You can change the representation of yytext by using either %array or %pointer in the definitions section of the **lex** specification file, as follows:

Defines yytext as a null-terminated character array. This is the default action. %array

%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.

Tracks the number of wide characters in the matched string. Multibyte characters have a length greater yywleng

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 action. For example, to count all instances of she and he, including the instances of he that are included in she, use the following action:

```
she
                   {s++; REJECT;}
                   {h++}
he
\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.

Adding Results to the yytext Array

Typically, 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 analyzer looks for strings:

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.

Returning Characters to the Input Stream

To return characters to the input stream, use the following 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 function 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 as x=-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 the following:

```
=-[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

The lex program 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. These subroutines 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 vytext 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 following entries:

The translation table contains additional lines that indicate the value associated with each character. For example:

```
{integer}
            {character string}
{integer}
             {character string}
{integer}
               {character string}
```

End-of-File Processing

When the lexical analyzer reaches the end of a file, it calls the yywrap library subroutine, which 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 that 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.

lex Program 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.

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.

A C language subroutine to perform lexical analysis on the input stream and pass tokens to yylex

the parser. You can use the lex command to generate this lexical analyzer subroutine.

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 vyparse subroutine cannot recover, it returns a value of 1 to the **main** subroutine. If it finds the end-marker token, the **yyparse** subroutine returns a value of 0 to the **main** subroutine.

The yacc Grammar File

To use the yacc command to generate a parser, provide it with 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. 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. 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 Id command (or to the cc command). The source code for the **main** library program is as follows:

```
#include <locale.h>
main()
     setlocale(LC ALL, "");
     yyparse();
```

The source code for the **yyerror** library program is as follows:

```
#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.

Because these programs are limited, 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.

vylex Subroutine

The input subroutine that you supply to the grammar file must be able to do the following:

- · 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.

For example, 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.

Note: To avoid conflict with the parser, do not use subroutine 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 vacc Grammar File

A yacc grammar file consists of the following sections:

- Declarations
- Rules
- Programs

Two adjacent % (percent sign, percent sign) 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 the following:

```
declarations
rules
2%
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 as follows:

```
%%
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 line characters in names or reserved symbols.

Using Comments

To explain what the program is doing, put comments in the grammar file. You can put comments anywhere in the grammar file that 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
' ∖b '	Backspace
'\ f '	Form-feed
'\ n '	New line
'\ r '	Return
'\t'	Tab
' ∖v '	Vertical tab
`\`,	Single quote (')
`\"`	Double quote (")
`\?`	Question mark (?)
'\\'	Backslash (\)
`\Digits`	The character whose encoding is represented by the one-, two-, or three-digit octal integer specified by the <i>Digits</i> string.
`\ x Digits`	The character whose encoding is represented by the sequence of hexadecimal characters specified by the <i>Digits</i> string.

Because its ASCII code is zero, the null character (\0 or 0) must not be used in grammar rules. The vylex subroutine returns 0 if the null character is used, signifying end of input.

Formatting the Grammar File

To help make the **yacc** grammar file more readable, use the following guidelines:

- Use uppercase letters for token names, and use lowercase letters for nonterminal symbol names.
- Put grammar rules and actions on separate lines to allow changing either one without changing the
- 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 Grammar File Declarations

The declarations section of the yacc grammar file contains the following:

- · 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

You can keep semantic information associated with the tokens that are 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 following 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), which is followed by a token or nonterminal name. These keywords are as follows:

%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 following declaration declares the Name parameter to be a token:

```
%token [<Tag>] Name [Number] [Name [Number]]...
```

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, the following describes the precedence and associativity of the four arithmetic operators:

```
%left '+' '-'
%left '*' '/'
```

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 structure, the parser starts in its top-down analysis of the input stream at this point. 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 as follows:

```
main()
        code segment
```

The start symbol points 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:

- A literal character is the numerical value of the character in the ASCII character set.
- 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. For more information on endmarker tokens, see "End-of-Input Marker" on page 297.

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 of the grammar file contains one or more grammar rules. Each rule describes a structure and gives it a name. A grammar rule has the following 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 that 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 following grammar rules:

```
A : B C D ;
A : E F ;
```

can be given to the vacc 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 following 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.

```
EndCase
rule
                EndCase rule
```

The following example defines the line rule as one or more combinations of a string followed by a newline character (\n):

```
1ines
                 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
          х;
OR
empty
          Х
```

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, which 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 grammar 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 can 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 following rule 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;
        B $ACT C
        x = $2:
        y = $3;
```

where \$ACT is an empty action.

vacc Program 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, ignoring 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 uses a special token name, error, 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 three 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 following form:

```
stat : error ';'
```

tells the parser that when there is an error, it should ignore 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. The following example shows one way to do this.

```
input : error '\n'
          printf(" Reenter last line: " );
         input
         $$ = $4;
```

However, in this example, the parser stays in the error state for three input tokens following the error. If the corrected line contains an error in the first three tokens, the parser deletes the tokens and does not produce a message. To allow for this condition, use the following 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;

Parser Operation Generated by the yacc Command

The **yacc** command converts the grammar file to a C language program. That program, when compiled and executed, parses the input according to the grammar specification provided.

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 the following 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.

The parser looks at all input, matches it to the grammar specification, and recognizes the input accept 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 that 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 determine 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 Action

The **shift** action is the most common action that the parser takes. Whenever the parser does a shift, a look-ahead token always exists. 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 Action

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, it is easy to 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 : xyz;

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 that 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 the following:

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 following grammar rule states a rule that forms an arithmetic expression by putting two other expressions together with a minus sign between them.

```
expr : expr '-' expr
```

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 can occur over which of its four actions to perform when processing the input. The following 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 either of 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, which is called a reduce/reduce conflict.

How the Parser Responds to Conflicts

When shift/reduce or reduce/reduce conflicts occur, the vacc 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, the vacc program uses the following rules:

- In a shift/reduce conflict, choose the shift.
- · 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 by using the preceding rules.

Turning on Debug Mode for a Parser Generated by the yacc Command

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 and 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 the dbx program 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 contains 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 vacc programs are as follows:

Content File

calc.lex Specifies the lex command specification file that defines the lexical analysis rules. For more

information, see "Lexical Analyzer Source Code" on page 307.

Specifies the yacc command grammar file that defines the parsing rules, and calls the yylex calc.yacc

subroutine created by the lex command to provide input. For more information, see "Parser Source

Code" on page 305.

The following descriptions assume that the calc.lex and calc.yacc example programs are located in your current directory.

Compiling the Example Program

To create the desk calculator example program, do the following:

1. Process the yacc grammar file using the -d optional flag (which informs 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 **Is** 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 **Is** command to verify that the following file was created:

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 **Is** command to verify that the following files were created:

y.tab.o

The object file for the **v.tab.c** source file

lex.yy.o

The object file for the lex.yy.c source file

a.out The executable program file

To run the program directly from the **a.out** file, type:

\$ a.out

OR

To move the program to a file with a more descriptive name, as in the following example, and run it, type:

\$ 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 as you would on a calculator. When you press the Enter key, the program displays the result of the operation. After you assign a value to a variable, as follows, the cursor moves to the next line.

```
m=4 <enter>
```

When you use the variable in subsequent calculations, it will have the assigned value:

```
m+5 <enter>
```

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 '*' '/' '%'
{\rm 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 DİGIT
          $$ = base * $1 + $2;
%%
main()
return(yyparse());
yyerror(s)
char *s;
 fprintf(stderr, "%s\n",s);
yywrap()
 return(1);
```

The file contains the following sections:

- · 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

This file contains include statements for standard input and output, as well as for the **y.tab.h** file. If you use the **-d** flag with the **yacc** command, the **yacc** program generates that file from the **yacc** grammar file information. The **y.tab.h** file contains definitions for the tokens that the parser program uses. In addition, the **calc.lex** file contains the rules to generate these tokens from the input stream. The following are the contents of the **calc.lex** file.

```
#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);
```

Related Information

Subroutine References

The **printf** subroutine.

Command References

The ed command in AIX 5L Version 5.2 Commands Reference, Volume 2.

The lex command in AIX 5L Version 5.2 Commands Reference, Volume 3.

The **sed** command in AIX 5L Version 5.2 Commands Reference, Volume 5.

The yacc command in AIX 5L Version 5.2 Commands Reference, Volume 6.

Chapter 12. make Command

This chapter provides information about simplifying the recompiling and relinking processes using the **make** command. It allows you to record, once only, specific relationships among files. You can then use the **make** command to automatically perform all updating tasks.

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 assists in maintaining a set of programs, usually pertaining to a particular software project, by building up-to-date versions of programs. The **make** command 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).

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 command to create many of the basic types of files.
- · Create libraries.

The **make** command requires a description file, file names, specified rules to inform the **make** command how to build many standard types of files, and time stamps of all system files.

The following topics are covered in this chapter:

- · "Creating a Description File"
- "Internal Rules for the make Program" on page 312
- · "Defining and Using Macros in a Description File" on page 315
- "Creating a Target File with the Make Command" on page 319
- "Using the make Command with Source Code Control System (SCCS) Files" on page 320
- "Understanding makefile considerations in Source Code Control Systems (SCCS)" on page 321
- "Using the make Command with Non-Source Code Control System (SCCS) Files" on page 321
- "Understanding How the make Command Uses the Environment Variables" on page 321
- "Example of a Description File" on page 322

Creating a Description File

The **make** command uses information from a description file, which you create, to build a file containing the completed program, which is then called a *target* file. The description file informs 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** command 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 either makefile or Makefile and are working in the directory containing that description file, type the following 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

where Desc-File is the name of the description file.

Format of an Entry in the make Description File

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 character 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 command ignores the # and all characters that follow it. The make command 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 quotation marks. 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 the 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...
             dependency list2...
test:
```

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 command discards 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 command 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 command from a Description File

To nest calls to the make command 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. To use this characteristic to test a set of description files that describe a program, enter:

The make command 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 command from Writing Commands

To prevent the make command 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 target name .SILENT on a dependency line by itself in the description file. Because .SILENT is not an actual 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 command from Stopping on Errors

The make command 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.0: Z.C
# Use the cc program to make z.o
    CC -C Z.C
```

If this file is called **makefile**, enter the **make** command to update the **prog** program after making changes to any of the source files: **x.c**, **y.c**, **z.c**, or **defs**.

Simplifying the Description File

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:

Internal Rules for the make Program

The internal rules for the **make** command are located in a file that looks like a description file. When the **-r** flag is specified, the **make** command 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, and rules that inform 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\~ Source Code Control System (SCCS) file containing C++ source file
- .c C source file
- .c~ SCCS file containing C source file
- .f FORTRAN source file
- .f~ SCCS file containing FORTRAN source file

```
.h
         C language header file.
         SCCS file containing C language header file
.h∼
J.
         lex source grammar
~ا.
         SCCS file containing lex source grammar.
         Object file.
٥.
         Assembler source file.
.s
         SCCS file containing assembler source file.
.s~
         Shell-command source file.
.sh
.sh~
         SCCS file containing shell-command source file.
         vacc-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 of .SUFFIXES. Because the make command looks at the suffixes list in left-to-right order, the order of the entries is important.

The **make** command uses the first entry in the list that satisfies the following equirements:

- The entry matches input and output suffix requirements for the current target and dependency files.
- The entry has a rule assigned to it.

.y∼

The **make** command 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 of .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=x1C
AS=as
CFLAGS=
CCFLAGS=
# End macro definitions for
# internal macros
# Create a .o file from a .c
# file with the cc program.
         $(CC) $(CFLAGS) -c $<
# Create a .o file from
# a .s file with the assembler.
S.0:
         $(AS)$(ASFLAGS) -o $@ $<
.y.o:
# Use yacc to create an intermediate file
         $(YACC) $(YFLAGS) $<
# Use cc compiler
```

Single-Suffix Rules

The **make** command has a set of single-suffix rules to build source files directly into a target file name that does not have a suffix (for example, command files). The **make** command also has rules to change the following source files with a 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, if all of the needed source files are in the current directory, to maintain the cat program, enter:

make cat

Using the Make Command with Archive Libraries

Use the **make** command to build libraries and library files. The **make** command recognizes the **.a** suffix 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** command 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** command 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** command 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 command in place of the usual C language compiler.

Similarly, the command:

make "CFLAGS=-0"

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, see 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 as follows:

.DEFAULT:

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 command 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 for each **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.

When you are using include files, do not use more than 16 levels of nesting.

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 by using just one shorter name. 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 "-11 -1y"
abc = -11 -1y
# 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 the macro name. 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:

The **make** command that uses 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 following command loads the files with the **lex** (-11) library:

make "LIBES= -11"

Note: When macros contain blanks and you enter 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, the expression \$(\$(macro2)) would evaluate to value1:

```
macro1=value1 macro2=macro1
```

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 = '1s *.0'
```

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 Is command at the time the OBJS macro is defined.

Internal Macros

The make command has built-in macro definitions for use in the description file. These macros help specify variables in the description file. The **make** command 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 command 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:

```
$$@.c
cat:
```

The make command translates it to:

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 similar to:

```
# Define macro CMDS as a series
# of command names
CMDS = cat dd echo date cc cmp comm ar 1d chown
# Each command depends on a .c file
$(CMDS):
            $$0.c
# Create the new command set by compiling the out of
# date files ($?) to the target file name ($0)
         $(CC) -0 $? -o $@
```

The make command 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 $@
```

The preceding description file creates a file in the /usr/include directory when the corresponding file in the current directory has been changed.

Changed 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 command 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 Out-of-Date file name is the name of the parent file whose timestamp did not match the timestamp of 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** command 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 command 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

```
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.

Creating a Target File with the Make Command

To create a file containing the completed program called a *target* file, 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 on which it depends.

If the target file or one of the parent files is out-of-date, the make command 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** command 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 commands 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. For more information about SCCS, seeChapter 21, "Source Code Control System (SCCS)," on page 471.

All SCCS files use the prefix **s.** to indicate that the files are SCCS files and not regular text files. The **make** command 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** command 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 command 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** command has internal rules for changing the following SCCS files:

```
.C\~.a:
.C\~.c:
.C\~.o:
.c~:
.c~:
.c~.a:
.c~.c:
.c~.o:
.f~:
.f~.a:
.f~.o:
.f~.f:
.h~.h:
```

.s~.a:

.sh~: .s~.o:

.y~.c:

.y~.o:

Understanding makefile considerations in Source Code Control Systems (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 command looks for an SCCS file named either s.makefile or s.Makefile. If either of these files is 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 command 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

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** command 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 contain equal signs) and assigns values to them. If the make command finds a definition for a macro on the command line different from the definition for that macro in the description file, it uses the command-line definition for the macro.

Next, the make command looks at the flags. For more information, see314 for a list of the make command flags.

The make command 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 command creates the target files in left-to-right order. Without a target file name, the make command 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.

Understanding 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 command 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.
- Reads the internal macro definitions.
- 5. Reads the environment. The make command 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 vacc 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 = -0
    # make depends on the files specified
    # in the OBJECTS macro definition
            $(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
            dosys.c doname.c files.c main.c misc.c \
    lint:
             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 command 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 -0 -c version.c
cc -0 -c main.c
cc -0 -c doname.c
cc -0 -c misc.c
cc -0 -c files.c
cc -0 -c dosys.c
yacc gram.y
mv y.tab.c gram.c
cc -0 -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 13. 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.

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** macro processor 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** macro processor 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** macro processor 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** macro processor processes each argument in order. If there are no arguments or if an argument is - (dash), **m4** macro processor reads standard input as its input file. The **m4** macro processor writes its results to standard output. Therefore, to redirect the output to a file for later use, use a command such as the following:

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** macro processor defines the string name as stuff. When the string name occurs in a program file, the **m4** macro processor 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 *I* (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 non-alphanumeric characters. In the following example, the variable NNN is not related to the defined macro N.

```
define(N, 100)
    . . .
if (NNN > 100)
```

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 initially.

```
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 `and' (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** macro processor 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** macro processor ignores this statement because it can define only names, not numbers.

Changing the Quote Characters

Quote characters are normally `and' (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 nth argument. When the macro is used, the m4 macro processor replaces the symbol with the value of the indicated argument. For example, the following 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** macro processor 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 as follows:

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 because corresponding arguments were not provided.

The **m4** macro processor 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 Predefined m4 Macro

The **m4** macro processor provides a set of predefined macros. This section explains 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. After you remove a built-in macro with the **undefine** macro, as follows: undefine(`define')

You cannot use the 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 macro processor 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.evalEvaluates 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 descending order of precedence):

```
unary + and -

** or ^ (exponentiation)

* / % (modulus)

+ -

== != < <= >>=
!(not)

& or && (logical AND)
I or II (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)')
```

Unless the text is very simple, use quote characters around the text that defines a macro

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, the **sinclude** macro (silent include).

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 macro processor can be redirected again to temporary files during processing, and the collected material can be output upon command. The m4 macro processor 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 macro processor 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 macro processor writes all redirected output to the temporary files in numerical order at the end of processing. The m4 macro processor 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 macro processor 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.

You can use the divnum macro to determine which temporary file is currently in use.

divnum

Returns the value of the currently active temporary file.

If you do not change the output file with the divert macro, the m4 macro processor 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** macro processor returns a string that is myfile concatenated with the process ID. Use this string to name a temporary file.

Using Conditional Expressions

Conditional expression evaluation allows process time determination of macro 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

The macros in this section allow you to convert input strings into output 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 ith 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:

ow 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(`little', aeiou, 12345)
```

replaces the vowels by the corresponding digits and returns the following:

13tt12

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(`little', aeiou)
```

deletes vowels from string little and returns the following:

1tt1

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
```

Debugging M4 Macros

The macros in this section allow you to report errors and processing information.

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.

Additional m4 Macros

A list of additional m4 macros, with a brief explanation of each, follows:

changecom (l, r)

defn (MacroName) en (String) m4exit (Code)

m4wrap (MacroName)

popdef (MacroName)

pushdef (MacroName, Replacement)

sysval

traceoff (MacroList)

traceon (MacroName)

Changes the left and right comment characters to the characters represented by the *I* and *r* variables.

Returns the quoted definition of *MacroName*Returns the number of characters in *String*.
Exits **m4** macro processor with a return code of *Code*.

Runs macro *MacroName* at the end of **m4**

macro processor.

Replaces the current definition of *MacroName* with the previous definition saved with the **pushdef** macro.

Saves the current definition of *MacroName* and then defines *MacroName* to be

Replacement.

Gets the return code from the last use of the **syscmd** macro.

Turns off trace for any macro in *MacroList*. If *MacroList* is null, turns off all tracing.
Turns on trace for macro *MacroName*. If *MacroName* is null, turns trace on for all

macros.

Chapter 14. 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 are provided 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"
- "ODM Descriptors" on page 339
- "ODM Object Searches" on page 342
- · "ODM Commands and Subroutines" on page 345
- "ODM Example Code and Output" on page 346

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 (see "ODM Commands and Subroutines" on page 345). 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

object

A group of objects with the same definition. An object class comprises one or more descriptors. It is similar to a table. 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. (see "ODM Descriptors" on page 339).

A member of an object class. It is an entity that requires storage and management of data. It is similar to a logical record in a database.

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.

© Copyright IBM Corp. 1997, 2004 335

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 is an example of manipulating object classes and objects.

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 1. 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
```

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 using one of the following:

1. Store the file in the default directory indicated by **\$ODMDIR**, which is the /usr/lib/objrepos directory.

- 2. Use the **export** 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 stored. Then, run the ODM commands in that directory. The file defining the object classes 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 containing the object classes using one of the following:

- 1. Work with object classes in the default directory indicated by **\$ODMDIR**, which is the /usr/lib/objrepos directory.
- 2. Use the **export** command to set the **ODMDIR** environment variable to specify the directory.
- 3. From the command line, use the export command to set the ODMPATH environment variable to a string containing a colon-separated list of directories to be searched for object classes. For example: \$ export ODMPATH = /usr/lib/objrepos:/tmp/myrepos

The directories in the **\$ODMPATH** are searched only if the directory indicated by **\$ODMDIR** does not contain the object class.

Creating an Object Class

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 346 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.

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

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" for instructions on creating an object class.
- 2. Create the definitions for one or more objects. "ODM Example Code and Output" on page 346 shows an ASCII file containing several object definitions.
- 3. Specify the directory which contains the object class 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.
- 4. Add objects to an 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.

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 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

Defines a character or numerical data type. (See "ODM"

Terminal Descriptors")

link descriptor (See "ODM Link Descriptor")Defines a relationship between object classesmethod descriptor (See "ODM Method Descriptor" onDefines an operation or method for an object

page 341)

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 342. 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:

shortSpecifies a signed 2-byte number.longSpecifies a signed 4-byte number.ulongSpecifies 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/int64Specifies a signed 8-byte number.ulong64/ODM_ULONG_LONG/uint64Specifies 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:

```
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"
                       = "Once upon a time"
       Birthday
                      = 19
       Age
                      = "Cinderella"
       Friends of
Fictional Characters:
       Story Star
                       = "Snow White"
                       = "Once upon a time"
       Birthday
       Age
                       = 18
                       = "Snow White"
       Friends of
Friend Table:
                       = "Cinderella"
       Friend of
       Friend
                       = "Fairy Godmother"
Friend Table:
                       = "Cinderella"
       Friend of
       Friend
                       = "mice"
Friend Table:
                      = "Snow White"
       Friend of
        Friend_
                       = "Sneezy"
Friend Table:
                       = "Snow White"
       Friend of
                       = "Sleepy"
       Friend
Friend_Table:
                       = "Cinderella"
       Friend of
                       = "Prince"
       Friend
Friend Table:
       Friend of
                       = "Snow White"
                       = "Happy"
       Friend
```

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.

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. The following table provides a conceptual picture of a link relationship between two object classes.

Friend_of (char)	Friend (char)
Cinderella	Fairy Godmother
Cinderella	mice
Snow White	Sneezy
Snow White	Sleepy
Cinderella	Prince
Snow White	Нарру

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
                           = 19
         Age
         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:
                       = "Sleepy"
        Others
        Dexterity
                       = 1
                       = 1
        Speed
        Strength
                       = 3
        Do This = "echo Sleepy has speed of 1"
Supporting Cast_Ratings:
                       = "Fairy Godmother"
        Others
                       = 10
        Dexterity
                       = 10
        Speed
        Strength
                      = 10
        Do This = "odmget -q "Others='Fairy Godmother'" Supporting Cast Ratings"
```

The following 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.

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"
```

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 SUPPN0=30, PARTN0>0, and PARTN0<101, and the AND logical operator is used to join the predicates. In the first predicate, SUPPN0

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</pre>

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:

1. 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:

```
2.545 0.5
                   -2e5
                        2.11E0
+4.555e-10 4E0 -10
                   999 +42
```

The E0 exponent can be used to specify no exponent.

2. Character string constants must be enclosed in single quotation marks:

```
'smith'
```

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, which is logically concatenated with predicate3.

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 you are writing a C language program using the subroutines, 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.

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.

odmget Retrieves objects from object classes and puts the object information into odmadd command

format.

Subroutines

ODM subroutines are:

odm_add_objAdds a new object to the object class.odm_change_objChanges the contents of an object.

odm_close_class Closes an object class.

odm_create_classCreates an empty object class.odm_err_msgRetrieves 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_firstRetrieves the first object that matches the specified criteria in an object class.odm_get_listRetrieves a list of objects that match the specified criteria in an object class.odm_get_nextRetrieves the next object that matches the specified criteria in an object class.odm_get_objRetrieves 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_pathSets the default path for locating object classes.odm_unlockUnlocks 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.

Table 2. 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

Table 3. Friend_Table

Friend_of (char)	Friend (char)
Cinderella	Fairy Godmother
Cinderella	mice
Snow White	Sneezy
Snow White	Sleepy
Cinderella	Prince
Snow White	Нарру

Table 4. 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];
                Enemy [20];
        char
};
class
        Fictional Characters {
                Story Star[20];
        char
        char
                Birthday [20];
        short
        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.
- Friends_of and Enemies_of are both from the link class and link to the two previously defined object classes.

Note: The object class link is repeated twice.

· Do This with a descriptor type of method.

To generate the object class files required by ODM, the file containing this code must be processed with the **odmcreate** command

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:

```
* MvObiects.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 {
                         * unique object id within object class *
                reserved; * reserved field *
      long
                 scratch; * extra field for application use *
               Friend of [20];
       char
                Friend[20];
       char
};
#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 {
                _id;
        long
        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 *
                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.
                        "Once upon a time"
Birthday
                        19
Age
                        "Cinderella"
Friends_of
                =
Enemies_of
                        "Cinderella"
Do This =
                        "echo Cleans house"
Fictional Characters:
Story Star
                        "Snow White"
Birthday
                        "Once upon a time"
Age
                =
                        18
                        "Snow White"
Friends of
Enemies_of
                        "Snow White"
Do_This
                         "echo Cleans house"
Friend Table:
Friend of
                        "Cinderella"
                        "Fairy Godmother"
Friend
Friend_Table:
                        "Cinderella"
Friend of
                        "mice"
Friend
Friend Table:
                        "Snow White"
Friend of
Friend
                         "Sneezy"
Friend Table:
Friend of
                         "Snow White"
Friend
                         "Sleepy"
Friend Table:
Friend_of
                         "Cinderella"
                        "Prince"
Friend
Friend Table:
Friend of
                         "Snow White"
                        "Happy"
Friend
```

```
Enemy_Table:
                       "Cinderella"
Enemy of
                       "midnight"
Enemy
Enemy Table:
                       "Cinderella"
Enemy of
                        "Mean Stepmother"
Enemy
Enemy_Table:
                        "Snow White"
Enemy_of
Enemy
                        "Mean Stepmother"
* End of MyObjects.add input file for ODM add utilities. *
```

Note: The * (asterisk) or the # (pound sign) comment in the previous example code will not go into the object file. If the beginning of the line causes the line to be a comment, the command does not go in the object file. The comment will be included in the file and treated as a string if it is included inside the " " (double quotation marks).

Related Information

For further information on this topic, see ODM Error Codes in AIX 5L Version 5.2 Technical Reference.

Subroutine References

The odm add objsubroutine, odm create class subroutine, odm run method subroutine.

Commands References

The odmadd command, odmchange command, odmcreate command, odmdelete command, odmdrop command, odmget command, odmshow command in AIX 5L Version 5.2 Commands Reference, Volume 4.

Chapter 15. Dynamic Logical Partitioning

Partitioning your system is similar to partitioning a hard drive. When you partition a hard drive, you divide a single physical hard drive so that the operating system recognizes it as a number of separate logical hard drives. On each of these divisions, called partitions, you can install an operating system and use each partition as you would a separate physical system.

A *logical partition* (LPAR) is the division of a computer's processors, memory, and hardware resources into multiple environments so that each environment can be operated independently with its own operating system and applications. The number of logical partitions that can be created depends on the system. Typically, partitions are used for different purposes, such as database operation, client/server operations, Web server operations, test environments, and production environments. Each partition can communicate with the other partitions as if each partition is a separate machine.

Dynamic logical partitioning (DLPAR) provides the ability to logically attach and detach a managed system's resources to and from a logical partition's operating system without rebooting. Some of the features of DLPAR include:

- The *Capacity Upgrade on Demand (CUoD*) feature of the pSeries, which allows customer to activate preinstalled but inactive processors as resource requirements change.
- The Dynamic Processor Deallocation feature of the pSeries servers, and on some SMP models.
 Dynamic Processor Deallocation enables a processor to be taken offline dynamically when an internal
 threshold of recoverable errors is exceeded. DLPAR enhances the Dynamic Processor Deallocation
 feature by allowing an inactive processor, if one exists, to be substituted for the processor that is
 suspected of being defective. This online switch does not impact applications and kernel extensions.
- DLPAR enables cross-partition workload management, which is particularly important for server consolidation in that it can be used to manage system resources across partitions.

DLPAR requests are built from simple add and remove requests that are directed to logical partitions. The user can execute these commands as move requests at the Hardware Management Console (HMC), which manages all DLPAR operations. DLPAR operations are enabled by pSeries firmware and AIX.

DLPAR-Safe and -Aware Programs

A DLPAR-safe program is one that does not fail as a result of DLPAR operations. Its performance might suffer when resources are removed and it might not scale with the addition of new resources, but the program still works as expected In fact, a DLPAR-safe program can prevent a DLPAR operation from succeeding because it has a dependency that the operating system is obligated to honor.

A *DLPAR-aware* program is one that has DLPAR code that is designed to adjust its use of system resources as the actual capacity of the system varies over time. This can be accomplished in the following ways:

- · By regularly polling the system in an attempt to discover changes in the system topology
- By registering application specific code that is notified when a change is occurring to the system topography

DLPAR-aware programs must be designed, at minimum, to avoid introducing conditions that might cause DLPAR operations to fail. At maximum, DLPAR-aware programs are concerned with performance and scalability. This is a much more complicated task because buffers might need to be drained and resized to maintain expected levels of performance when memory is added or removed. In addition, the number of threads must be dynamically adjusted to account for changes in the number of online processors. These thread-based adjustments are not necessarily limited to processor-based decisions. For example, the best

way to reduce memory consumption in Java programs might be to reduce the number of threads, because this reduces the number of active objects that need to be processed by the Java Virtual Machine's garbage collector.

Most applications are DLPAR-safe by default.

Making Programs DLPAR-Safe

The following types of errors, which represent binary compatibility exposures, can be introduced by DLPAR:

Note: These errors are a result of processor addition.

· If a program has code that is optimized for uniprocessor systems and the number of processors in the partition is increased from one to two, programs that make runtime checks might take an unexpected path if a processor is added during one of these checks. Potential problems can also occur in programs that implement their own locking primitives, but do so using uniprocessor serialization techniques; that is, the sync and isync instructions are not included. The use of these instructions is also required for self-modifying and generated code, and are thus necessary on DLPAR-enabled systems. Be sure to look for uniprocessor-based logic. Programs that make uniprocessor assertions must include logic that identifies the number of online processors.

Programs can determine the number of online processors by:

- Loading the _system_configuration.ncpus field
- var.v ncpus
- Using the sysconf system call with the SC NPROCESSORS ONLN flag.
- · Programs that index data by processor number typically use the mycpu system call to determine the identity of the currently executing processor, in order to index into their data structures. The problem potentially occurs when a new processor is added because the path to the data might not be properly initialized or allocated. Programs that preallocate processor-based lists using the number of online CPUs are broken, because this value changes with DLPAR.

Avoid this problem by preallocating processor-based data using the maximum possible number of processors that can be brought online at the same time. The operating system can be said to be configured to support a maximum of N processors, not that there are N processors active at any given time. The maximum number of processors is constant, while the number of online processors is incremented and decremented as processors are brought online and taken offline. When a partition is created, the minimum, desired, and maximum numbers of processors are specified. The maximum value is reflected in the following variables:

- _system_configuration.max_ncpus
- _system_configuration.original_ncpus
- var.v_ncpus_cfg
- sysconf (SC NPROCESSORS CONF)

The _system_configuration.original_ncpus and var.v_ncpus_cfg variables are preexisting variables. On DLPAR-enabled systems they represent a potential maximum value. On systems not enabled for DLPAR, the value is dictated by the number of processors that are configured at boot time. Both represent the conceptual maximum value that can be supported, even though a processor might have been taken offline by Dynamic Processor Deallocation. The use of these preexisting fields is recommended for applications that are built on AIX 4.3, because this facilitates the use of the same binary on AIX 4.3 and later. If the application requires runtime initialization of its processor-based data, it can register a DLPAR handler that is called before a processor is added.

Making Programs DLPAR-Aware

A DLPAR-aware program is one that is designed to recognize and dynamically adapt to changes in the system configuration. This code need not subscribe to the DLPAR model of awareness, but can be structured more generally in the form of a system resource monitor that regularly polls the system to

discover changes in the system configuration. This approach can be used to achieve some limited performance-related goals, but because it is not tightly integrated with DLPAR, it cannot be effectively used to manage large-scale changes to the system configuration. For example, the polling model might not be suitable for a system that supports processor hot plug, because the hot-pluggable unit might be composed of several processor and memory boards. Nor can it be used to manage application-specific dependencies, such as processor bindings, that need to be resolved before the DLPAR processor remove event is started.

The following types of applications can exploit DLPAR technology:

- Applications that are designed to scale with the system configuration, including those:
 - That detect the number of online processors or the size of physical memory when the application starts
 - That are externally directed to scale based on an assumed configuration of processors and memory. which usually translates into the use of a maximum number of threads, maximum buffer sizes, or a maximum amount of pinned memory
- Applications that are aware of the number of online processors and the total quantity of system memory, including the following types of applications:
 - Performance monitors
 - Debugging tools
 - System crash tools
 - Workload managers
 - License managers

Note: Not all license managers are candidates for DLPAR, especially user-based license managers.

- Applications that pin their application data, text, or stack using the plock system call
- Applications that use System V Shared Memory Segments with the PinvOption (SHM_PIN)
- Applications that bind to processors using the **bindprocessor** system call

Dynamic logical partitioning of large memory pages is not supported. The amount of memory that is preallocated to the large page pool can have a material impact on the DLPAR capabilities of the partition regarding memory. A memory region that contains a large page cannot be removed. Therefore, application developers might want to provide an option to not use large pages.

Making Programs DLPAR-Aware Using DLPAR APIs

Application interfaces are provided to make programs DLPAR-aware. The SIGRECONFIG signal is sent to the applications at the various phases of dynamic logical partitioning. The DLPAR subsystem defines check, pre and post phases for a typical operation. Applications can watch for this signal and use the DLPAR-supported system calls to learn more about the operation in progress and to take any necessary actions.

Note: When using signals, the application might inadvertently block the signal, or the load on the system might prevent the thread from running in a timely fashion. In the case of signals, the system will wait a short period of time, which is a function of the user-specified timeout, and proceed to the next phase. It is not appropriate to wait indefinitely because a non-privileged rogue thread could prevent all DLPAR operations from occurring.

The issue of timely signal delivery can be managed by the application by controlling the signal mask and scheduling priority. The DLPAR-aware code can be directly incorporated into the algorithm. Also, the signal handler can be cascaded across multiple shared libraries so that notification can be incorporated in a more modular way.

To integrate the DLPAR event using APIs, do the following:

- 1. Catch the SIGRECONFIG signal by using the sigaction system call. The default action is to ignore the
- 2. Control the signal mask in at least one of the threads so that the signal can be delivered in real time.
- 3. Ensure that the scheduling priority for the thread that is to receive the signal is sufficient so that it will run quickly after the signal has been sent.
- 4. Run the dr_reconfig system call to obtain the type of resource, type of action, phase of the event, as well as other information that is relevant to the current request.

Note: The **dr reconfig** system call is used inside the signal handler to determine the nature of the DLPAR request.

Managing an Application's DLPAR Dependencies

A DLPAR remove request can fail for a variety of reasons. The most common of these is that the resource is busy, or that there are not enough system resources currently available to complete the request. In these cases, the resource is left in a normal state as if the DLPAR event never happened.

The primary cause of processor removal failure is processor bindings. The operating system cannot ignore processor bindings and carry on DLPAR operations or applications might not continue to operate properly. To ensure that this does not occur, release the binding, establish a new one, or terminate the application. The specific process or threads that are impacted is a function of the type of binding that is used. For more information, see "Processor Bindings."

The primary cause of memory removal failure is that there is not enough pinned memory available in the system to complete the request. This is a system-level issue and is not necessarily the result of a specific application. If a page in the memory region to be removed has a pinned page, its contents must be migrated to another pinned page, while atomically maintaining its virtual to physical mappings. The failure occurs when there is not enough pinnable memory in the system to accommodate the migration of the pinned data in the region that is being removed. To ensure that this does not occur, lower the level of pinned memory in the system. This can be accomplished by destroying pinned shared memory segments. terminating programs that implement the **plock** system call, or removing the **plock** on the program.

The primary cause of PCI slot removal failure is that the adapters in the slot are busy. Note that device dependencies are not tracked. For example, the device dependency might extend from a slot to one of the following: an adapter, a device, a volume group, a logical volume, a file system, or a file. In this case, resolve the dependencies manually by stopping the relevant applications, unmounting file systems, and varying off volume groups.

Processor Bindings

Applications can bind to a processor by using the bindprocessor system call. This system call assumes a processor-numbering scheme starting with zero (0) and ending with N-1, where N is the number of online CPUs. N is determined programmatically by reading the _system_configuration.ncpus system variable. As processors are added and removed, this variable is incremented and decremented using dynamic logical partitioning.

Note that the numbering scheme does not include holes. Processors are always added to the Nth position and removed from the Nth-1 position. The numbering scheme used by the bindprocessor system call cannot be used to bind to a specific logical processor, because any processor can be removed and this is not reflected in the numbering scheme, because the Nth-1 CPU is always deallocated. For this reason, the identifiers used by the bindprocessor system call are called bind CPU IDs.

Changes to the _system_configuration.ncpus system variables have the following implications:

- · Applications must be prepared to receive an error from the bindprocessor system call if the last processor is removed after the applications have read the variable. This error condition was first introduced by the Dynamic Processor Deallocation (runtime deallocation of defective processors).
- Applications that are designed to scale with the number of processors must reread the **_system_configuration.ncpus** system variable when the number of processors changes.

Applications can also bind to a set of processors using a feature of Workload Manager (WLM) called Software Partitioning. It assumes a numbering scheme that is based on logical CPU IDs, which also start with zero (0) and end with N-1. However, N in this case is the maximum number of processors that can be supported architecturally by the partition. The numbering scheme reflects both online and offline processors.

Therefore, it is important to note the type of binding that is used so that the correct remedy can be applied when removing a processor. The bindprocessor command can be used to determine the number of online processors. The ps command can be used to identify the processes and threads that are bound to the last online processor. After the targets have been identified, the bindprocessor command can be used again to define new attachments.

WLM-related dependencies can be resolved by identifying the particular software partitions that are causing problems. To resolve these dependencies, do the following:

Note: The system schedules bound jobs around offline or pending offline processors, so no change is required if the particular software partition has another online CPU.

- 1. Use the **Isrset** command to view the set of software partitions that are used by WLM.
- 2. Identify these software partitions by using the **Isclass** command.
- 3. Identify the set of classes that use these software partitions by using the **chclass** command.
- 4. Reclassify the system using the wimctrl command.

At this point, the new class definitions take effect and the system automatically migrates bound jobs away from the logical processor that is being removed.

Integrating the DLPAR Operation into the Application

The DLPAR operation can be integrated into the application in the following ways:

- · Using a script-based approach, where the user installs a set of DLPAR scripts into a directory. These scripts are invoked while the DLPAR operation is being run. The scripts are designed to externally reconfigure the application.
- Using the SIGRECONFIG signal, which is used to catch the signal of every process that has registered. The signal method assumes that the application has been coded to catch the signal, and that the signal handler will reconfigure the application. The signal handler invokes an interface to determine the nature of the DLPAR operation.

Both of these methods follow the same high level structure. Either method can be used to provide support for DLPAR, although only the script-based mechanism can be used to manage DLPAR dependencies related to Workload Manager software partitions (processor sets). No APIs are associated with Workload Manager, so the use of a signal handler is not a suitable vehicle for dealing with the Workload Manager-imposed scheduling constraints. The applications themselves are not Workload Manager-aware. In this case, the system administrator might want to provide a script that invokes Workload Manager commands to manage DLPAR interactions with Workload Manager.

The decision of which method to use should be based on how the system or resource-specific logic was introduced into the application. If the application was externally directed to use a specific number of threads or to size its buffers, use the script-based approach. If the application is directly aware of the system configuration and uses this information accordingly, use the signal-based approach.

The DLPAR operation itself is divided into the following phases:

check phase

The check phase is invoked first and it enables applications to fail the current DLPAR request before any state in the system is changed. For example, the check phase could be used by a CPU-based license manager to fail the integration of a new processor if that CPU addition makes the number of processors in the system exceed the number of licensed processors. It could also be used to preserve the DLPAR safeness of a program that is not DLPAR-safe. In the latter case, consideration must be given to services provided by the application, because it might be better to stop the program, complete the request, and then restart the program.

pre phase and post phase

The pre phase and post phase are provided to stop the program, complete the request, and then restart the program.

The system attempts to ensure that all of the check code across the different mediums is executed in its entirety at the system level before the DLPAR event advances to the next phase.

Actions Taken by DLPAR Scripts

Application scripts are invoked for both add and remove operations. When removing resources, scripts are provided to resolve conditions imposed by the application that prevent the resource from being removed. The presence of particular processor bindings and the lack of pinnable memory might cause a remove request to fail. A set of commands is provided to identify these situations, so that scripts can be written to resolve them.

To identify and resolve the DLPAR dependencies, the following commands can be used:

- The ps command displays bindprocessor attachments and plock system call status at the process
- The bindprocessor command displays online processors and makes new attachments.
- · The kill command sends signals to processes.
- The **ipcs** command displays pinned shared-memory segments at the process level.
- The **Isrset** command displays processor sets.
- The Isclass command displays Workload Manager classes, which might include processor sets.
- The **chclass** command is used to change class definitions.

Scripts can also be used for scalability and general performance issues. When resources are removed, you can reduce the number of threads that are used or the size of application buffers. When the resources are added, you can increase these parameters. You can provide commands that can be used to dynamically make these adjustments, which can be triggered by these scripts. Install the scripts to invoke these commands within the context of the DLPAR operation.

High-Level Structure of DLPAR Scripts

This section provides an overview of the scripts, which can be Perl scripts, shell scripts, or commands. Application scripts are required to provide the following commands:

· scriptinfo

Identifies the version, date, and vendor of the script. It is called when the script is installed.

register

Identifies the resources managed by the script. If the script returns the resource name cpu or mem, the script will be automatically invoked when DLPAR attempts to reconfigure processors and memory. respectively. The register command is called when the script is installed with the DLPAR subsystem.

· usage resource name

Returns information describing how the resource is being used by the application. The description should be relevant so that the user can determine whether to install or uninstall the script. It should identify the software capabilities of the application that are impacted. The usage command is called for each resource that was identified by the register command.

checkrelease resource_name

Indicates whether the DLPAR subsystem should continue with the removal of the named resource. A script might indicate that the resource should not be removed if the application is not DLPAR-aware and the application is considered critical to the operation of the system.

prerelease resource name

Reconfigures, suspends, or terminates the application so that its hold on the named resource is released.

postrelease resource name

Resumes or restarts the application.

undoprerelease resource name

Invoked if an error is encountered and the resource is not released.

checkacquire resource_name

Indicates whether the DLPAR subsystem should proceed with the resource addition. It might be used by a license manager to prevent the addition of a new resource, for example cpu, until the resource is licensed.

• preacquire resource name

Used to prepare for a resource addition.

undopreacquire resource name

Invoked if an error is encountered in the preacquire phase or when the event is acted upon.

postacquire resource name

Resumes or starts the application.

Installing Application Scripts Using the drmgr Command

The drmgr command maintains an internal database of installed-script information. This information is collected when the system is booted and is refreshed when new scripts are installed or uninstalled. The information is derived from the **scriptinfo**, **register**, and **usage** commands. The installation of scripts is supported through options to the drmgr command, which copies the named script to the /usr/lib/dr/scripts/all/ directory where it can be later accessed. You can specify an alternate location for this repository. To determine the machine upon which a script is used, specify the target host name when installing the script.

To specify the location of the base repository, use the following command:

```
drmgr -R base directory path
```

To install a script, use the following command:

```
drmgr -i script name [-f] [-w mins] [-D hostname]
```

The following flags are defined:

- The -i flag is used to name the script.
- The -f flag must be used to replace a registered script.
- The -w flag is used to specify the number of minutes that the script is expected to execute. This is provided as an override option to the value specified by the vendor.
- The **-D** flag is used to register a script to be used on a particular host.

To uninstall a script, use the following command:

```
drmgr -u script name [-D hostname]
```

The following flags are defined:

- The **-u** flag is used to indicate which script should be uninstalled.
- The -D flag is used to uninstall a script that has been registered for a specific directory.

To display information about scripts that have already been installed, use the following command: drmgr -1

Naming Conventions for Scripts

It is suggested that the script names be built from the vendor name and the subsystem that is being controlled. System administrators should name their scripts with the sysadmin prefix. For example, a system administrator who wanted to provide a script to control Workload Manager assignments might name the script sysadmin wlm.

Script Execution Environment and Input Parameters

Scripts are invoked with the following execution environment:

- · Process UID is set to the UID of the script.
- · Process GID is set to the GID of the script.
- PATH environment variable is set to the /usr/bin:/etc:/usr/sbin directory.
- LANG environment variable might or might not be set.
- Current working directory is set to /tmp.
- · Command arguments and environment variables are used to describe the DLPAR event.

Scripts receive input parameters through command arguments and environment variables, and provide output by writing name=value pairs to standard output, where name=value pairs are delimited by new lines. The name is defined to be the name of the return data item that is expected, and value is the value associated with the data item. Text strings must be enclosed by parentheses; for example, DR ERROR="text". All environment variables and name=value pairs must begin with DR, which is reserved for communicating with application scripts.

The scripts use **DR_ERROR** *name=value* environment variable pair to provide error descriptions.

You can examine the command arguments to the script to determine the phase of the DLPAR operation, the type of action, and the type of resource that is the subject of the pending DLPAR request. For example, if the script command arguments are checkrelease mem, then the phase is check, the action is remove, and the type of resource is memory. The specific resource that is involved can be identified by examining environment variables.

The following environment variables are set for memory add and remove:

Note: In the following description, one frame is equal to 4 KB.

• DR FREE FRAMES=0xffffffff

The number of free frames currently in the system, in hexadecimal format.

• DR MEM SIZE COMPLETED=n

The number of megabytes that were successfully added or removed, in decimal format.

• DR_MEM_SIZE_REQUEST=n

The size of the memory request in megabytes, in decimal format.

DR PINNABLE FRAMES=0xFFFFFFFF

The total number of pinnable frames currently in the system, in hexadecimal format. This parameter provides valuable information when removing memory in that it can be used to determine when the system is approaching the limit of pinnable memory, which is the primary cause of failure for memory remove requests.

• DR TOTAL FRAMES=0xFFFFFFFF

The total number of frames currently in the system, in hexadecimal format.

The following environment variables are set for processor add and remove:

DR BCPUID=N

The bind CPU ID of the processor that is being added or removed in decimal format. A bindprocessor attachment to this processor does not necessarily mean that the attachment has to be undone. This is only true if it is the Nth processor in the system, because the Nth processor position is the one that is always removed in a CPU remove operation. Bind IDs are consecutive in nature, ranging from 0 to N and are intended to identify only online processors. Use the bindprocessor command to determine the number of online CPUs.

DR LCPUID=N

The logical CPU ID of the processor that is being added or removed in decimal format.

The operator can display the information about the current DLPAR request using the detail level at the HMC to observe events as they occur. This parameter is specified to the script using the **DR_DETAIL_LEVEL=***N* environment variable, where *N* can range from 0 to 5. The default value is zero (0) and signifies no information. A value of one (1) is reserved for the operating system and is used to present the high-level flow. The remaining levels (2-5) can be used by the scripts to provide information with the assumption that larger numbers provide greater detail.

Scripts provide detailed data by writing the following name=value pairs to standard output:

name=value pair	Description
DR_LOG_ERR=message	Logs the message with the syslog level of the LOG_ERR environment variable.
DR_LOG_WARNING=message	Logs the message with the syslog level of the LOG_WARNING environment variable.
DR_LOG_INFO=message	Logs the message with the syslog level of the LOG_INFO environment variable.
DR_LOG_EMERG=message	Logs the message with the syslog level of the LOG_EMERG environment variable.
DR_LOG_DEBUG=message	Logs the message with the syslog level of the LOG_DEBUG environment variable.

In addition, the operator can also set up a log of information that is preserved by using the syslog facility, in which case, the above information is routed to that facility as well. You must configure the syslog facility in this case.

DLPAR Script Commands

This section describes the script commands for DLPAR:

scriptinfo	Provides information about the installed scripts, such as their creation date and resources.
register	Invoked to collect the list of resources that are managed by the script. The drmgr command uses these lists to invoke scripts based on the type of resource that is being reconfigured.
usage	Provides human-readable strings describing the service provided by the named resource. The context of the message should help the user decide the implications on the application and the services that it provides when named resource is reconfigured. This command is invoked when the script is installed, and the information provided by this command is maintained in an internal database that is used by the drmgr command. Display the information using the -/ list option of the drmgr command.

checkrelease

When removing resources, the drmgr command assesses the impacts of the removal of the resource. This includes execution of DLPAR scripts that implement the checkrelease command. Each DLPAR script in turn will be able to evaluate the peculiarities of its application and indicate to the drmgr command that is using the script's return code whether the resource removal will affect the associated application. If it finds that the removal of the resource can be done safely, an exit status of success is returned. If the application is in a state that the resource is critical to its execution and cannot be reconfigured without interrupting the execution of the application, then the script indicates the resource should not be removed by returning an error. When the FORCE option is specified by the user, which applies to the entire DLPAR operation including its phases, the drmgr command skips the checkrelease command and begins with the prerelease

prerelease

Before a resource is released, the DLPAR scripts are directed to assist in the release of the named resource by reducing or eliminating the use of the resource from the application. However, if the script detects that the resource cannot be released from the application, it should indicate that the resource will not be removed from the application by returning an error. This does not prevent the system from attempting to remove the resource in either the forced or non-forced mode of execution, and the script will be called in the post phase, regardless of the actions or inactions that were taken by the prerelease command. The actions taken by the operating system are safe. If a resource cannot be cleanly removed, the operation will fail.

The DLPAR script is expected to internally record the actions that were taken by the prerelease command, so that they can be restored in the post phase should an error occur. This can also be managed in post phase if rediscovery is implemented. The application might need to take severe measures if the force option is specified.

postrelease

After a resource is successfully released, the postrelease command for each installed DLPAR script is invoked. Each DLPAR script performs any post processing that is required during this step. Applications that were halted should be restarted.

The calling program will ignore any errors reported by the postrelease commands, and the operation will be considered a success, although an indication of any errors that may have occurred will also be reported to the user. The DR_ERROR environment variable message is provided for this purpose, so the message should identify the application that was not properly reconfigured.

undoprerelease

After a prerelease command is issued by the drmgr command to the DLPAR script, if the drmgr command fails to remove or release the resource, it will try to revert to the old state. As part of this process, the drmgr command will issue the undoprerelease command to the DLPAR script. The undoprerelease command will only be invoked if the script was previously called to release the resource in the current DLPAR request. In this case, the script should undo any actions that were taken by the prerelease command of the script. To this end, the script might need to document its actions, or otherwise provide the capability of rediscovering the state of the system and reconfiguring the application, so that in effect, the DLPAR event never occurred.

checkacquire

This command is the first DLPAR script-based command that is called in the acquire-new-resource sequence. It is called for each installed script that previously indicated that it supported the particular type of resource that is being added. One of the primary purposes of the checkacquire phase is to enable processor-based license managers, which might want to fail the addition of a processor. The checkacquire command is always invoked, regardless of the value of the FORCE environment variable, and the calling program honors the return code of the script. The user cannot force the addition of a new processor if a script or DLPAR-aware program fails the DLPAR operation in the check phase.

In short, the FORCE environment variable does not really apply to the checkacquire command, although it does apply to the other phases. In the preacquire phase, it dictates how far the script should go when reconfiguring the application. The force option can be used by the scripts to control the policy by which applications are stopped and restarted similar to when a resource is released, which is mostly a DLPAR-safe issue.

preacquire	Assuming that no errors were reported in the checkacquire phase, the system advances to the preacquire phase, where the same set of scripts are invoked to prepare for the acquisition of a new resource, which is supported through the preacquire command. Each of these scripts are called, before the system actually attempts to integrate the resource, unless an error was reported and the FORCE environment variable was not specified by the user. If the FORCE environment variable was specified, the system proceeds to the integrate stage, regardless of the script's stated return code. No errors are detected when the FORCE environment variable is specified, because all errors are avoidable by unconfiguring the application, which is an accepted practice when the FORCE environment variable is specified. If an error is encountered and the FORCE environment variable is not specified, the system will proceed to the undopreacquire phase, but only the previously executed scripts in the current phase are rerun. During this latter phase, the scripts are directed to perform recovery actions.
undopreacquire	The undopreacquire phase is provided so that the scripts can perform recovery operations. If a script is called in the undopreacquire phase, it can assume that it successfully completed the preacquire command.
postacquire	The postacquire command is executed after the resource has been successfully integrated by the system. Each DLPAR script that was previously called in the check and pre phases is called again. This command is used to incorporate the new resource into the application. For example, the application might want to create new threads, expands its buffers, or the application might need to be restarted if it was previously halted.

Making Kernel Extensions DLPAR-Aware

Like applications, most kernel extensions are DLPAR-safe by default. However, some are sensitive to the system configuration and might need to be registered with the DLPAR subsystem. Some kernel extensions partition their data along processor lines, create threads based on the number of online processors, or provide large pinned memory buffer pools. These kernel extensions must be notified when the system topology changes. The mechanism and the actions that need to be taken parallel those of DLPAR-aware applications.

Registering Reconfiguration Handlers

The following kernel services are provided to register and unregister reconfiguration handlers:

```
#include sys/dr.h
```

The parameters for the **reconfig_register** subroutine are as follows:

- The handler parameter is the kernel extension function to be invoked
- The *actions* parameter allows the kernel extension to specify which of the following events require notification:
 - DR_CPU_ADD_CHECK
 - DR_CPU_ADD_PRE
 - DR_CPU_ADD_POST
 - DR_CPU_ADD_POST_ERROR
 - DR CPU REMOVE CHECK
 - DR_CPU_REMOVE_PRE
 - DR_CPU_REMOVE_POST
 - DR_CPU_REMOVE_POST_ERROR
 - DR_MEM_ADD_CHECK
 - DR_MEM_ADD_PRE

- DR_MEM_ADD_POST
- DR MEM ADD POST ERROR
- DR_MEM_REMOVE_CHECK
- DR_MEM_REMOVE_PRE
- DR MEM REMOVE POST
- DR_MEM_REMOVE_POST_ERROR
- The h_arg parameter is specified by the kernel extension, remembered by the kernel along with the function descriptor for the handler, and then passed to the handler when it is invoked. It is not used directly by the kernel, but is intended to support kernel extensions that manage multiple adapter instances. In practice, this parameter points to an adapter control block.
- The *h_token* parameter is an output parameter and is intended to be used when the handler is unregistered.
- The name parameter is provided for information purposes and can be included within an error log entry if the driver returns an error. It is provided by the kernel extension and should be limited to 15 ASCII characters.

The reconfig register function returns 0 for success and the appropriate errno value otherwise.

The reconfig_unregister function is called to remove a previously installed handler.

Both the reconfig register and reconfig unregister function can only be called in the process environment.

If a kernel extension registers for the pre phase, it is advisable that it register for the check phase to avoid partial unconfiguration of the system when removing resources.

Reconfiguration Handlers

The interface to the reconfiguration handler is as follows:

```
struct dri cpu {
                                                                              /* Logical CPU Id of target CPU */
                                              1cpu;
               cpu_t
               cpu t
                                              bcpu;
                                                                            /* Bind Id of target CPU
};
struct dri mem {
              size64_t req_memsz_change; /* user requested mem size */
size64_t sys_memsz; /* system mem size at start */
size64_t act_memsz_change; /* mem added/removed so far */
rpn64_t sys_free_frames; /* Number of free frames */
rpn64_t sys_pinnable_frames;/* Number of pinnable frames */
rpn64_t sys_total_frames: /* Total_number_of_frames */
                                             sys_total_frames; /* Total number of frames */
               rpn64 t
               unsigned long long lmb_addr; /* start addr of logical memory block */
size64_t lmb_size; /* Size of logical memory block being added */
};
```

The parameters to the reconfiguration handler are as follows:

- · The event parameter is passed to the handler and is intended to be used only, when calling the reconfig handler complete subroutine.
- The *h_arg* parameter is specified at registration time by the handler.

int (*handler)(void *event, void *h_arg, int req, void *resource_info);

- The reg parameter indicates the DLPAR operation performed by the handler.
 - DR CPU ADD CHECK
 - DR_CPU_ADD_PRE
 - DR_CPU_ADD_POST
 - DR CPU ADD POST ERROR
 - DR_CPU_REMOVE_CHECK

- DR CPU REMOVE PRE
- DR CPU REMOVE POST
- DR_CPU_REMOVE_POST_ERROR
- DR_MEM_ADD_CHECK
- DR MEM ADD PRE
- DR_MEM_ADD_POST
- DR_MEM_ADD_POST_ERROR
- DR MEM REMOVE CHECK
- DR MEM REMOVE PRE
- DR_MEM_REMOVE_POST
- DR MEM REMOVE POST ERROR
- The resource_info parameter identifies the resource specific information for the current DLPAR request. If the request is processor-based, then the resource info data is provided through a dri cpu structure. Otherwise a dri mem structure is used.

Reconfiguration handlers are invoked in the process environment.

Kernel extensions can assume the following:

- · Only a single type of resource is being configured or removed at a time
- · Multiple processors will not be specified at the same time. However, kernel extensions should be coded to support the addition or removal of multiple logical memory blocks. The customer may initiate a request to add or remove gigabytes of memory.

The check phase provides the ability for DLPAR-aware applications and kernel extensions to react to the user's request before it has been applied. Therefore, the check-phase kernel extension handler is invoked once, even though the request might devolve to multiple logical memory blocks. Unlike the check phase, the pre phase, post phase, and post-error phase are applied at the logical memory block level. This is different for application notification, where the pre phase, post phase, or alternatively the post-error phase are invoked once for each user request, regardless of the number of underlying logical memory blocks. Another difference is that the post-error phase for kernel extensions is used when a specific logical memory block operation fails, while the post-error phase for applications is used when the operation, which in this case is the entire user request, fails.

In general, during the check phase, the kernel extension examines its state to determine whether it can comply with the impending DLPAR request. If this operation cannot be managed, or if it would adversely effect the proper execution of the extension, then the handler returns DR_FAIL. Otherwise the handler returns **DR_SUCCESS**.

During the pre-remove phase, kernel extensions attempts to remove any dependencies that it might have on the designated resource. An example is a driver that maintains per-processor buffer pools. The driver might mark the associated buffer pool as pending delete, so that new requests are not allocated from it. In time, the pool will be drained and it might be freed. Other items that must be considered in the pre-remove phase are timers and bound threads, which need to be stopped and terminated, respectively. Alternatively, bound threads can be unbound.

During the post-remove phase, kernel extensions attempts to free any resources through garbage collection, assuming that the resource was actually removed. If it was not, timers and threads must be re-established. The DR resource POST ERROR request is used to signify that an error occurred.

During the pre-add phase, kernel extensions should pre-initialize any data paths that are dependent on the new resource, so that when the new resource is configured, it is ready to be used. The system does not guarantee that the resource will not be used prior to the handler being called again in the post phase.

During the post-add phase, kernel extensions can assume that the resource has been properly added and can be used. This phase is a convenient place to start bound threads, schedule timers, and increase the size of buffers.

If possible, within a few seconds, the reconfiguration handlers return DR SUCCESS to indicate successful reconfiguration, or DR FAIL to indicate failure. If more time is required, the handler returns DR WAIT.

Extended DR Handlers

If a kernel extension expects that the operation is likely to take a long time, that is, several seconds, the handler returns DR_WAIT to the caller, but proceed with the request asynchronously. In the following case, the handler indicates that it has completed the request by invoking the reconfig handler complete routine.

void reconfig handler complete(void *event, int rc);

The event parameter is the same parameter that was passed to the handler when it was invoked by the kernel. The rc parameter must be set to either DR_SUCCESS or DR_FAIL to indicate the completion status of the handler.

The **reconfig_handler_complete** kernel service can be invoked in the process or interrupt environments.

Using the xmemdma Kernel Service

On systems that are capable of DLPAR, such as the dynamic removal of memory, calls to the **xmemdma** kernel service without the XMEM_DR_SAFE flag result in the specified memory being flagged as not removable. This is done to guarantee the integrity of the system, because the system has no knowledge of how the caller intends to utilize the real memory address that was returned. Dynamic memory removal operations are still possible for other memory, but not for the memory specified by the xmemdma call.

If the caller is using the real memory address for informational purposes only (such as for trace buffers or debug information), the caller can set the XMEM DR SAFE flag. This is an indication to the system that the real memory address can be exposed to the caller without any risk of data corruption. When this flag is present, the system will still allow the specified memory to be dynamically removed.

If the caller is using the real memory address to perform actual data access, either by turning off data translation and performing CPU load/store access to the real memory, or by programming DMA controllers to target the real memory, the XMEM DR SAFE flag should not be set. If the flag is set, the system's data integrity could be compromised when the memory is dynamically removed. For information on converting a kernel extension that uses real memory addresses in this way to be DLPAR-aware, contact your IBM Service Representative.

For more information, see the **xmemdma** kernel service.

Related Information

Sample scripts are available in the /usr/samples/dr/scripts directory.

drmgr, drslot command in AIX 5L Version 5.2 Commands Reference, Volume 2.

dr reconfig System Call and reconfig register and reconfig unregister Kernel Services in AIX 5L Version 5.2 Technical Reference: Kernel and Subsystems Volume 1

Hardware Management Console Installation and Operations Guide

Planning for Partitioned-System Operations

Chapter 16. 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 performs its editing without interacting with the person requesting the editing. This method of operation allows **sed** 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</pre>
```

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, <i>FileName</i> 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\\nText	
	Writes the lines contained in <i>Text</i> to the output stream after <i>Line1</i> . The a command must appear at the end of a line.	
change lines	[Line1 [,Line2]]c\\nText	
	Deletes the lines specified by <i>Line1</i> and <i>Line2</i> as the <i>delete lines</i> command does. Then it writes <i>Text</i> 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 \\nText
	Writes the lines contained in <i>Text</i> to the output stream before <i>Line1</i> . 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 <i>Line1</i> to <i>Line2</i> 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 <i>Pattern</i> . When it finds a match, the command replaces that set of characters with the set of characters specified by <i>String</i> .

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 <i>FileName</i> at the point in the editing process that the w command occurs.
	If <i>FileName</i> 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 <i>FileName</i> .
read file	[Line1]r FileName
	Reads FileName and appends the contents after the line indicated by Line1.
	Include exactly one space between r and <i>FileName</i> . If <i>FileName</i> 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 <i>Line1</i> and <i>Line2</i> if present, to the holding area.
pick up copy, appended	[Line1 [,Line2]]H
	Copies the contents of the pattern space indicated by <i>Line1</i> and <i>Line2</i> 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 <i>Line1</i> and <i>Line2</i> 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 <i>Line1</i> and <i>Line2</i> 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 <i>Line1</i> and <i>Line2</i> 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 <i>not</i> selected by <i>Line1</i> and <i>Line2</i> .

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 <i>Line1</i> and <i>Line2</i> . 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 <i>Label</i> is a string of up to 8 bytes. Each <i>Label</i> in the editing stream must be different from any other <i>Label</i> .
branch to label, unconditional	[Line1 [,Line2]]bLabel
	Branches to the point in the editing stream indicated by <i>Label</i> and continues processing the current input line with the commands following <i>Label</i> . If <i>Label</i> 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 <i>Label</i> must appear as a <i>Label</i> in the editing stream.
test and branch	[Line1 [,Line2]]tLabel
	If any successful substitutions were made on the current input line, branches to <i>Label</i> . 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 <i>Line1</i> .

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 <i>String</i> is replaced by the set of characters in the input lines that matched <i>Pattern</i> . 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 <i>String</i> is replaced by the set of characters in the input lines that matches the d th substring in <i>Pattern</i> . 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 <i>String</i> for all instances of <i>Pattern</i> in the indicated line(s). Characters in <i>String</i> are not scanned for a match of <i>Pattern</i> after they are inserted. For example, the command: s/r/R/g changes: From: the red round rock To: the Red Round Rock
р	Prints (to STDOUT) the line that contains a successfully matched <i>Pattern</i> .
w FileName	Writes to <i>FileName</i> the line that contains a successfully matched <i>Pattern</i> . if <i>FileName</i> 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 <i>FileName</i> .

Chapter 17. Shared Libraries and Shared Memory

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 -I flag may end in .so, as well as in .a. That is, a reference to -Ifoo 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 **Id** 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 diclose 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.

In AIX 4.2.1 and later, linker option that links a module so that only some of its dependents are loaded -blazy 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:

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 -I and -L flags with the Id 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 0x00000000000000 to 0xFFFFFFFFF, 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 (23 2 bytes), but can access the same virtual address space as 64-bit processes (28 of bytes).

The Physical Address Space of 64-bit Systems

0xFFFFFFFFFFFFFFFF, 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 379 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 387 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 Chapter 18, "System Memory Allocation Using the malloc Subsystem," on page 391 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 operationsshmget Gets or creates a shared memory segment

shmat Attaches a shared memory segment from a processshmdt 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.

- "Comparing mmap with shmat" on page 380
- "mmap Compatibility Considerations" on page 381
- "Using the Semaphore Subroutines" on page 382

- "Mapping Files with the shmat Subroutine" on page 382
- "Mapping Shared Memory Segments with the shmat Subroutine" on page 383

Comparing mmap 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. The portion of address space available for mapping files consists of addresses in the rangesof 0x30000000-0xCFFFFFFF and 0xE0000000-0xEFFFFFFF for a total of 2.75G bytes of address space. In AIX 5.2 and later, a 32-bit process run with the very large address-space model has the range 0x30000000-0xFFFFFFFF available for mappings, with a total of up to 3.25GB of address space. For more information on the very large address-space model, see "Understanding the Large Address-Space Model" on page 175.

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 00000000-0x07FFFFF 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.

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 385 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 Id 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 386 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 <i>file name</i> of a file.
shmid	The identifier assigned to the shared segment for use within a particular process. It is similar in use to a <i>file descriptor</i> 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 386 for additional information.

Inter-Process Communication (IPC) Limits

This document describes system limits for IPC mechanisms.

On some UNIX systems, the system administrator can edit the /etc/master file and define limits for IPC mechanisms (semaphores, shared memory segments, and message queues). The problem with this method is that the higher the limits, the more memory the operating system uses, and performance can be adversely affected.

AIX uses a different method. In AIX, upper limits are defined for the IPC mechanisms, which are not configurable. The individual IPC data structures are allocated and deallocated as needed, so memory requirements depend on the current system usage of IPC mechanisms.

This difference in methods sometimes confuses users who are installing or using databases. The limit that is the most confusing is the maximum number of shared memory segments that can be attached simultaneously per process. For 64-bit processes, the maximum number of shared memory segments is 268435456. For 32-bit processes, the maximum number of shared memory segments is 11, unless the extended **shmat** capability is used. For more information on extending **shmat**, see "Understanding Memory Mapping" on page 379.

The following tables summarize the semaphore limits on IPC mechanisms.

Semaphores	4.3.0	4.3.1	4.3.2	5.1	5.2
Maximum number of semaphore IDs:	4096	4096	131072	131072	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

The following tables summarize the message queue limits on IPC mechanisms.

Message Queue	4.3.0	4.3.1	4.3.2	5.1	5.2
Maximum message size	4 MB				
Maximum bytes on queue	4 MB				
Maximum number of message queue IDs	4096	4096	131072	131072	131072
Maximum messages per queue ID	524288	524288	524288	524288	524288

The following tables summarize the shared memory limits on IPC mechanisms.

Shared Memory	4.3.0	4.3.1	4.3.2	5.1	5.2
Maximum segment size (32-bit)	256 MB	2 GB	2 GB	2 GB	2 GB
Maximum segment size (64-bit)	256 MB	2 GB	2 GB	64 GB	1 TB
Minimum segment size	1	1	1	1	1
Maximum number of shared memory IDs	4096	4096	131072	131072	131072
Maximum number of segments per process (32-bit)	11	11	11	11	11
Maximum number of segments per process (64-bit)	268435456	268435456	268435456	268435456	268435456

Note: For 32-bit processes, the maximum number of segments per process is limited only by the size of the address space when the extended shmat capability is used.

IPC Limits on AIX 4.3

- · For semaphores and message queues, the table shows the system limits
- For shared memory, the maximum shared memory segment size is 256GB.
- · For shared memory without the extended **shmat** capability:
 - A process can attach a maximum of 11 shared memory segments.
- · For shared memory with the extended **shmat** capability:
 - When a shared memory segments is attached, its size is rounded to a multiple of 4096 bytes
 - A process can attach as many shared memory segments as will fit in the available address space. The maximum available address space size is 11 segments, or 11 times 256 MB.
- The extended shmat capability is used if the environment variable EXTSHM has the value ON when the process starts executing.
- · The available address space for attaching shared memory segments is reduced if the large or very large address-space model is used. For more information, see Chapter 8, "Large Program Support," on page 175.

IPC Limits on AIX 4.3.1

• The maximum size of a shared memory segment increases from 256 MB to 2 GB. When a shared memory segment larger than 256 MB is attached, its size is rounded to a multiple of 256 MB, even if the extended **shmat** capability is being used.

IPC Limits on AIX 4.3.2

- The maximum number of message queues, semaphore IDs, and shared memory segments is 131072.
- The maximum number of messages per queue is 524288.

IPC Limits on AIX 5.1

· The maximum size of a shared memory segment for 64-bit processes is 64 GB. A 32-bit process cannot attach a shared memory segment larger than 2 GB.

IPC Limits on AIX 5.2

- · The maximum size of a shared memory segment for 64-bit processes is 1 TB. A 32-bit process cannot attach a shared memory segment larger than 2 GB.
- · 32-bit applications can use the shmat capability to obtain more than 11 segments when using the very large address space model without having to use extended shmat. For more information on the very large address space model, see "Understanding the Very Large Address-Space Model" on page 176.
- · Applications can query the IPC limits on the system using the vmgetinfo system call.

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 **Iseek** 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)</pre>
      (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:

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)
}</pre>
```

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.

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.

Returns paging device status. swapgry

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_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 Perform memory operations.

memmove

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

monstartupStarts and stops execution profiling using default-sized data areasmsem_initInitializes a semaphore in a mapped file or shared memory region

msem_lockLocks a semaphoremsem_removeRemoves a semaphoremsem_unlockUnlocks 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

shmctlControls shared memory operationsshmdtDetaches a shared memory segmentshmgetGets 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.

Related Information

For further information on this topic, see the following:

- AIX 5L Version 5.2 Performance Management Guide has a section on Virtual Memory Manager (VMM) which describes the VMM and its page-stealing algorithms in detail.
- Memory Kernel Services in AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts describes the various kernel extensions available to manipulate kernel memory.
- Paging Space Overview in AIX 5L Version 5.2 System Management Guide: Operating System and Devices

Subroutine References

- The diclose subroutine, diopen subroutine, _end, _etext or _edata identifier, exec subroutine, exit subroutine, ftok subroutine, fsync subroutine, load subroutine, loadquery subroutine, loadbind subroutine, malloc, free, realloc, calloc, mallopt, mallinfo, or alloca subroutine, monitor subroutine, moncontrol subroutine, monstartup subroutine in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 1.
- **shmat** subroutine, **shmctl** subroutine, **shmdt** subroutine, **shmget** subroutine, **unload** subroutine in *AIX* 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2.

Commands Refernces

The ar command, as command, cc command in AIX 5L Version 5.2 Commands Reference, Volume 1.

The dump command, ipcs command in AIX 5L Version 5.2 Commands Reference, Volume 2.

The ipcrm command, Id command in AIX 5L Version 5.2 Commands Reference, Volume 3.

The **pagesize** command,**rtl_enable** command, **update** command in *AIX 5L Version 5.2 Commands Reference, Volume 5.*

The vmstat command in AIX 5L Version 5.2 Commands Reference, Volume 6.

Files References

The XCOFF object (a.out) file format.

Chapter 18. 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 the following fundamental memory operations:

Allocation:

Performed by the malloc and calloc subroutines.

· Deallocation:

Performed by the free subroutine.

· Reallocation:

Performed 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 compatibility with the Berkeley Compatibility Library.

For additional information, see the following sections:

- "Understanding System Allocation Policy" on page 392
- "Understanding the Default Allocation Policy" on page 392
- "Understanding the malloc 3.1 Allocation Policy" on page 393
- "Comparing the Default and malloc 3.1 Allocation Policies" on page 395

Working with the Heap in 32-bit Applications

A 32-bit application program running on the system has an address space that is divided into the following segments:

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 0xcfffffff	Available for use by shared memory or mmap services
0xd0000000 to 0xdfffffff	Contains shared library text
0xe0000000 to 0xefffffff	Available for use by shared memory or mmap services

Working with the Heap in 64-bit Applications

A 64-bit application program running on the system has an address space that is divided into the following segments:

```
0x0000 0000 0000 0000 to 0x0000 0000 0fff ffff
                                                       Contains the kernel
0x0000 0000 f000 0000 to 0x0000 0000 0fff ffff
                                                       Reserved
0x0000 0001 0000 0000 to 0x07ff ffff ffff
                                                       Contains the application program text and application
                                                       program data, and shared memory or mmap services
0x0800 0000 0000 0000 to 0x08ff ffff ffff
                                                       Privately loaded objects
                                                       Shared library text and data
0x0900 0000 0000 0000 to 0x09ff ffff ffff ffff
0x0f00 0000 0000 0000 to 0x0fff 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 up the edata location 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 using 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 malloc 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 malloc 3.1 allocation policy has some unique behavioral characteristics that may be beneficial in specific circumstances, as described under "Comparing the Default and malloc 3.1 Allocation Policies" on page 395. However, the malloc 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 405
- "Malloc Buckets" on page 407

"Debug Malloc Tool" on page 400

Allocation

The number of bytes required for a block is calculated using a roundup function. The equation is as

```
If x \mod y = 0, then
Roundup(x,y) = x
Roundup(x,y) = (x/y \text{ rounded down to the nearest integer} + 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 adjoining block 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. A new block of the requested size is then 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 smaller one is returned to the free tree.

Understanding the malloc 3.1 Allocation Policy

The malloc 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 malloc 3.1 allocation policy (64-bit programs will continue to use the default allocation policy). Setting the MALLOCTYPE environment variable to any value other than 3.1 causes the default allocation policy to be used.

The malloc 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. Because the prefix immediately precedes the block, an entire page is required solely for the prefix.

Block Size	Sizes Mapped	Pages Used
16	0 8	
32	9 24	
64	25 56	
128	57 120	
256	121 248	
512	249 504	
1K	505 1K-8	
2K	1K-7 2K-8	
4K	2K-7 4K-8	2
8K	4K-7 8K-8	3
16K	8K-7 16K-8	5
32K	16K-7 32K-8	9
64K	32K-7 64K-8	17
128K	64K-7 128K-8	33
256K	128K-7 256K-8	65
512K	256K-7 512K-8	129
1M	256K-7 1M-8	257
2M	1M-7 2M-8	513
4M	2M-7 4M-8	1K + 1
8M	4M-7 8M-8	2K + 1
16M	8M-7 16M-8	4K + 1
32M	16M-7 32M-8	8K + 1
64M	32M-7 64M-8	16K + 1
128M	64M-7 128M-8	32K + 1
256M	128M-7 256M-8	64K + 1
512M	256M-7 512M-8	128K + 1
1024M	512M-7 1024M-8	256K + 1
2048M	1024M-7 2048M-8	512K + 1
	16 32 64 128 256 512 1K 2K 4K 8K 16K 32K 64K 128K 256K 512K 1M 2M 4M 8M 16M 32M 64M 128M 256M 512M 1024M	16 0 8 32 9 24 64 25 56 128 57 120 256 121 248 512 249 504 1K 505 1K-8 2K 1K-7 2K-8 4K 2K-7 4K-8 8K 4K-7 4K-8 8K 4K-7 16K-8 32K 16K-7 32K-8 64K 32K-7 64K-8 128K 64K-7 128K-8 256K 128K-7 256K-8 512K 256K-7 1M-8 8M 256K-7 1M-8 8M 4M-7 2M-8 8M 4M-7 2M-8 8M 4M-7 2M-8 8M 4M-7 2M-8 8M 4M-7 4M-8 8M 4M-7 32M-8 64M 32M-7 64M-8 16M 32M-7 64M-8 128M 64M-7 128M-8 256M 128M-7 256M-8 512M 526M-7 512M-8

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,
bucket = (\log(needed)/\log(2) \text{ rounded down to the nearest integer}) - 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 malloc 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 is used instead.

The malloc 3.1 allocation policy does *not* support any of the following capabilities:

- Malloc Multiheap
- Malloc Buckets
- Debug Malloc

Comparing the Default and malloc 3.1 Allocation Policies

Because the malloc 3.1 allocation policy 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 malloc 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 malloc 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 allocates only the number of bytes requested.

As another example, because of the inefficient space reclamation of the malloc 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 the malloc subroutine 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 reallocates a structure to a slightly greater size, the malloc 3.1 allocator may not need to move the structure very often. In many cases, the realloc subroutine can make use of the extra space provided by the rounding implicit in the malloc 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 called by the malloc subroutine just above it. This may present the appearance of a degradation in realloc subroutine performance when the default allocator is used instead of the malloc 3.1 allocator. In reality, it is the surfacing of a cost that is implicit in the application program's structure.

Optimizing Malloc

To reduce the overhead of glink code, the function descriptor for the malloc subsystem functions are overwritten with the function descriptor for the actual implementation. Because some programs might not work when function pointers are modified, the no overwrite option can be used to disable this optimization.

To disable optimization, set the MALLOCTYPE environment variable as follows: MALLOCTYPE=no overwrite

Note: The no_overwrite option can be used with any other MALLOCTYPE option. Separate multiple options with a comma.

User-Defined Malloc Replacement

Users can replace the memory subsystem (malloc, calloc, realloc, free, mallopt and mallinfo subroutines) 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 must be thread-safe so that it works in both threaded and non-threaded processes. Because there are no checks to verify that it is, 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 32-bit shared object named mem32.o and the 64-bit 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 user-shared objects can optionally export the following symbol:

malloc once

Execution does not stop if the entry point does not exist.

The functions are defined as follows:

```
void * malloc (size t):
```

This function is the user equivalent of the **malloc** subroutine.

```
void __free__(void *) :
```

This function is the user equivalent of the **free** subroutine.

```
void * realloc (void *, size t) :
```

This function is the user equivalent of the **realloc** subroutine.

```
void *__calloc__(size_t, size_t) :
```

This function is the user equivalent of the **calloc** subroutine.

```
int __mallopt__(int, int) :
```

This function is the user equivalent of the **mallopt** subroutine.

```
struct mallinfo mallinfo ():
```

This function is the user equivalent of the mallinfo subroutine.

```
void malloc once ()
```

This function will be called once before any other user-defined malloc entry point is called.

The following functions are used by the thread subsystem to manage the user-defined memory subsystem in a multi-threaded environment. They 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 functions must be defined and exported. Otherwise, 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 the fork subroutine 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 the fork subroutine 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. Separate modules must exist for 32- and 64-bit implementations placed in an archive. For example:

mem.exp module:

```
malloc
 free
 realloc
 calloc
 mallopt
 mallinfo
 malloc init
__malloc_prefork_lock_
__malloc_postfork_unlock_
__malloc_once
```

• mem functions32.o module:

Contains all of the required 32-bit functions

mem functions64.o module:

Contains all of the required 64-bit functions

The following examples are for creating the shared objects. The -1pthreads parameter is needed only if the object uses pthread functions.

Creating 32-bit shared object:

```
1d -b32 -m -o mem32.o mem functions32.o \
-bE:mem.exp \
-bM:SRE -lpthreads -lc
```

Creating 64-bit shared object:

```
1d -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
```

Enabling the User-Defined Memory Subsystem

The user-defined memory subsystem can be enabled by using one of the following:

- The MALLOCTYPE environment variable
- The malloc user defined name global variable 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 malloc user defined name global variable, 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.

Note:

- 1. When a setuid application is run, 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 MALLOCTYPE environment variable and the malloc user defined name global variable are used to specify the archive name, the archive specified by MALLOCTYPE will override the one specified by malloc user defined name.

32-bit and 64-bit Considerations

If the archive does not contain both the 32-bit and 64-bit shared objects and the user-defined memory subsystem was enabled using the MALLOCTYPE environment variable, there will be problems executing 64-bit processes from 32-bit applications and 32-bit processes from 64-bit applications. When a new process is created using the **exec** subroutine, the process inherits the environment of the calling application. This means that the MALLOCTYPE environment variable 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 program, the load will fail and the new process will exit.

Thread Considerations

All of the provided 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 () (that is, malloc () should initially assume that malloc init () has not yet run.) After malloc init () has completed, malloc () can rely on any work done by malloc init (). This is required because the pthread initialization uses malloc() before __malloc_init_() is called.

The following variables are provided to prevent unneeded thread-related routines from being called:

- The multi threaded variable is zero until a thread is created when it becomes non-zero and will not be reset to zero for that process.
- The n pthreads variable 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 might look similar to the following:
if ( multi threaded)
pthread_mutex_lock(mutexptr);
/* ..... work ..... */
if ( multi threaded)
pthread mutex unlock(mutexptr);
```

In this example, __malloc__() is prevented from executing pthread functions before pthreads is fully initialized. Single-threaded applications are also accelerated 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 because the **setlocale** subroutine uses malloc() to initialize the locales. If malloc() fails then the setlocale subroutine cannot finish and the application is still in the POSIX locale. Therefore, only the default English messages will be displayed.

Existing statically built programs cannot use the user-defined memory subsystem without recompiling.

Error Reporting

The first time the malloc subroutine 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 displays and the application exits. 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 is terminated and the list of missing symbols displays.

Debug Malloc Tool

Debugging applications that are mismanaging memory allocated by using 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 memory subsystem includes an optional debug capability to allow users to identify memory overwrites, overreads, duplicate frees and reuse of freed memory allocated by the malloc subroutine. Memory problems detected by Debug Malloc result in an abort call 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 3.1 malloc.

Enabling Debug Malloc

Debug Malloc is not enabled by default, but 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. If the application being debugged frequently calls the malloc subroutine, it might be necessary to enable the application to access additional memory by using the **ulimit** command and the *-bmaxdata* option of the **Id** command. For more information, see "Disk and Memory Considerations" on page 404.

MALLOCDEBUG Configuration 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

To set the MALLOCDEBUG environment variable, use 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:

Each configuration option should only be specified once when setting the MALLOCDEBUG environment variable. If a configuration option is specified more than once per setting, only the final instance will apply.

The options corresponding to Malloc Debug 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

The MALLOCDEBUG options are described as follows:

alian:n By default, the **malloc** subroutine returns a pointer aligned on a 2-word boundary (4-word in 64-bit 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 (for example, 0, 1, 2, 4, ...). The values 0 and 1 are treated as the same, that is, there is no alignment so any memory accesses outside the allocated area will cause an abort subroutine call.

Note:

- 1. For more information, see "Calculating Overreads and Overwrites Using the align:n Option" on page 403.
- 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 result in undefined behavior.

postfree checking: [onl off]

By default, the malloc subsystem allows the calling program to access memory that was previously freed. This is 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 is produced.

If postfree checking and the allow overreading option are turned off, a significant amount of space will be saved because of the removal of a guard page and page rounding that is needed for both of these options. In this case each allocation request will only be increased by 4 times the size of unsigned long. If this option is the only option turned off, nothing will occur. The default is on.

validate ptrs

By default, the **free** subroutine does not validate its input pointer to ensure that it actually references memory previously allocated by the malloc subroutine. If the parameter passed to the free subroutine is a NULL value, the free subroutine will return to the caller without taking any action. If the parameter is invalid, the results are undefined. A core dump might not occur in this case, depending upon the value of the invalid parameter. Specifying the Debug Malloc validate_ptrs option will cause the free subroutine to perform extensive validation on its input parameter. If the parameter is found to be invalid (that is, it does not reference memory previously allocated by a call to the malloc or realloc subroutine), 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 either of the following ways:

- · Memory access errors (such as trying to read or write past the end of allocated memory) cause a segmentation violation (SIGSEGV), resulting in a core dump.
- · For other types of errors (such as trying to free space that was already freed), Debug Malloc will output an error message, then call theabort function, which will send a SIGIOT signal to end the current process.

If the calling program is blocking or catching the SIGSEGV and 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).

Note:

- 1. The override signal handling option can be ineffective in a threaded application environment because Debug Malloc uses the sigprocmask subroutine and many threaded processes use the pthread_sigmask subroutine.
- 2. If a thread calls the sigwait subroutine 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 might be undetected.

allow overreading: [on | off]

By default, if the calling program attempts to read past the end of allocated memory, Debug Malloc will respond with a segmentation violation and a core dump. 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.

If allow_overreading and the postfree_checking option are turned off, a significant amount of space will be saved because of the removal of a guard page and page rounding that is needed for both of these options. In this case each allocation request will only be increased by 4 times the size of unsigned long. If this option is the only option turned off, then each allocation will have to be at least a page because of the need for page rounding. The default is on.

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 as described for the record_allocations option.

Note:

- 1. Specifying the *report_allocations* option automatically enables the *record_allocations*
- 2. One allocation record will always be listed for the atexit subroutine that produces 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 the malloc subroutine
- · A six-function traceback starting from the call to the malloc subroutine

The original size passed to the malloc subroutine

Each allocation record will be retained until the memory associated with it is freed.

debug_range:min:max

By default, if the Debug Malloc option is selected, Debug Malloc is invoked throughout the life of the program. If the debug range option is specified, any allocation requests that fall between min and max will be allocated using Debug Malloc. Otherwise, the Default Allocator will be invoked. This option allows the user to control memory usage during Debug Malloc by only using this implementation in specific cases.

Page alignment is necessary because Debug Malloc features are invoked at certain points in the program. Therefore, every allocation will be at least a page size in length.

If the **realloc** subroutine is called with an allocation request that falls within the range of *min* and max, Debug Realloc is invoked even if the previous allocation that returned the pointer was not within the range. The reverse of this is also true.

If 0 is specified as a minimum value, then anything that is less than the maximum value will use Debug Malloc. If 0 is specified as a maximum value, then anything that is greater than the minimum value will use Debug Malloc.

Note: If the override_signal option is set in conjunction with the debug_range option, the overriding of the SIGIOT and SIGSEGV signal behavior is performed for both Debug and Default allocations. For more information, see "Disk and Memory Considerations" on page 404.

The debug range option also decreases memory usage by only invoking Debug Malloc in certain cases. Because allocations must be page aligned, every allocation request is rounded to a multiple of the PAGESIZE macro.

functionset:function1name:function2name:...

By default, if the Debug Malloc option is selected, Debug Malloc is invoked throughout the life of the program. If the dbg funcset option is specified, any allocation request that was called by a function in the user-provided list of functions invokes Debug Malloc.

The user-provided list of functions is a list of the function names. If the realloc subroutine is called with an allocation from a function in the list, Debug Realloc is invoked even if the previous allocation that returned the pointer was not in the list. The reverse of this is also true.

This option will not check for invalid functions in the list.

Note: If the override_signal option is set in conjunction with the debug_range option, the overriding of the SIGIOT and SIGSEGV signal behavior is performed for both Debug and Default allocations. For more information, see "Disk and Memory Considerations" on page 404.

The functionset option allows memory to be used more significantly in needed cases. Because Debug Malloc features are invoked at certain points of the program, all allocations must be page aligned and each allocation is rounded to a multiple of the PAGESIZE macro.

Calculating Overreads and Overwrites Using the align:n Option

To calculate how many bytes of overreads or overwrites Debug Malloc will allow for a given allocation request when MALLOCDEBUG=align:n and n is the number of bytes to be allocated, use the following formula:

```
((((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 or overwrites with Debug Malloc enabled:

1. In this example, 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 handles overreads and overwrites as follows:

- · When an even number of bytes is allocated, Debug Malloc allocates exactly the number of bytes requested, which will allow for 0 bytes of overreads or overwrites.
- · When an odd number of bytes is allocated, Debug Malloc allocates the number of bytes requested, plus one additional byte to satisfy the required alignment. This allows for 1 byte of overreads or overwrites.
- 2. In this example, 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 or overwrites in all cases, regardless of the number of bytes requested.

Debug Malloc Output

All memory problems detected by Debug Malloc result in a call to the abort subroutine, 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 subroutine (malloc, free, realloc or calloc) detects an error, a message is displayed and the abort subroutine is called.

Performance Considerations

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 can 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.

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 run with Debug Malloc enabled when trying to debug a known problem. After 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 times the size of unsigned long, then rounded up to the next multiple of the PAGESIZE macro. Debug Malloc might 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.

By using the off configuration of the allow overreading and postfree checking options, memory usage will be significantly lowered. Each malloc request will either be increased by 4 times the size of unsigned long if both options are turned off, or rounded to the next multiple of the PAGESIZE macro if only the allow overreading option is set to off.

If the application being debugged frequently calls the malloc subroutine, it might encounter memory usage problems with Debug Malloc enabled that could prevent the application from executing properly in a single segment. If this occurs, it might be helpful to enable the application to access additional memory by using the ulimit command and the -bmaxdata option of the Id command.

For the purpose of running with Debug Malloc enabled, set the **ulimit** for both the data (-d) and stack (-s) variables as follows:

```
ulimit -d unlimited
ulimit -s unlimited
```

To reserve the maximum of 8 segments for a 32-bit process, the -bmaxdata option should be specified as -bmaxdata:0x80000000.

When Debug Malloc is turned off, the default values for ulimit and -bmaxdata should be restored.

For more information about the ulimit command and the -bmaxdata option, see Chapter 8, "Large Program Support," on page 175.

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 might 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 using 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. 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 (for example, 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.

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 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. The malloc subsystem can therefore only service one thread at a time, resulting in 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, 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 the malloc subroutine in a multiprocessor environment.

Activation and configuration of the malloc multiheap capability is available at process startup through 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.

MALLOCMULTIHEAP Options

The MALLOCMULTIHEAP environment variable options are as follows:

- · heaps:n
- considersize

Each of these options is described in detail later in this document.

To set the MALLOCMULTIHEAP environment variable, use 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 preceding 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.

The **MALLOCMULTIHEAP** options are described as follows:

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** subroutine calls. However, because of the additional processing required, the considersize heap-selection algorithm is somewhat slower than the default heap selection algorithm.

If the heaps are unable to allocate space, the malloc subroutine will return NULL and set errno to ENOMEM. If there is no available memory in the current heap, the malloc subsystem will check the other heaps for available space.

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 succeeding bucket 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 preceding 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.

The number of blocks per bucket, number of buckets, and bucket-sizing factor are all set with 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 configured 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, but 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.

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" on page 409.

To set the the MALLOCBUCKETS environment variable, use the following syntax:

```
\texttt{MALLOCBUCKETS} = [[ number_of_buckets:n \mid bucket_sizing_factor:n \mid blocks per bucket:n \mid blocks per bucket
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 valid delimiters 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 the MALLOCBUCKETS environment variable. 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 writes a warning message to standard error and then continues execution using a documented default value.

The MALLOCBUCKETS environment variable will be recognized by the malloc subsystem only 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 shows 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 path name

If a user-specified path name 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.

Note:

- 1. One additional allocation request will always be shown in the first bucket for the atexit subroutine 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. Before modifying the default configuration, 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 malloc 3.1 allocation policy.

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 for system-wide use. For optimal performance, malloc buckets should be enabled and configured on a per-application basis.

Malloc Report

Malloc Error Reporting provides an optional error reporting and detection extension to the malloc subsystem. Information on errors that occurred in the malloc environment will be reported and actions can be performed if specified.

Malloc Error Reporting is not enabled by default, but can be enabled and configured prior to process startup through the MALLOCDEBUG environment variable. All errors located in the malloc subsystem are output to standard error, along with detailed information.

Malloc Error Reporting is not available for applications using User-Defined Malloc.

Malloc Report allows the user to provide a function that the malloc subsytem will call when it encounters an error. Before returning, Malloc Report calls the user-provided function.

A global function pointer is available for use by the user. In the code, the following function pointer should be set to the user's function:

```
extern void (*malloc err function)(int, ...)
```

The user provided function must set the following:

```
void malloc err function(int, ...)
```

For example, the following code must be inserted into the user's application:

```
malloc err function = &abort sub
```

Check Arena

If the Default Allocator is enabled, you can check the structures that contain the free blocks before every allocation request is processed. This option will ensure that the arena is not corrupted. Also, the arena will also be checked when an error occurs.

For Default Malloc, the checkarena option checks for NULL pointers in the free tree or pointers that do not fall within a certain range. If an invalid pointer is encountered during the descent of the tree, the program might perform a core dump depending on the value of the invalid address.

The checkarena option is not available for applications using the malloc 3.1 allocation policy or User-Defined Malloc.

Enabling Error Reporting

Error reporting is not enabled by default, but is enabled and configured by setting the MALLOCDEBUG environment variable as follows:

MALLOCDEBUG=verbose

To enable a check of the arena before allocation request is processed or when an error occurs, set the MALLOCDEBUG environment variable to the following:

MALLOCDEBUG=checkarena

More than one option can be specified in any order, as long as they are comma-separated, as in the following example:

MALLOCDEBUG=checkarena, verbose

Malloc Trace

Malloc Trace provides an optional extension to the malloc subsystem for use with the trace facility. Traces of the malloc, realloc, and free subroutines are recorded for use in problem determination and performance analysis.

Malloc Trace is not enabled by default, but can be enabled and configured prior to process startup through the MALLOCDEBUG environment variable.

Events Recorded by Malloc Trace

The tracehook IDs used for Malloc Trace are as follows:

- HKWD_LIB_MALL_COMMON
- HKWD_LIB_MALL_INTERNAL

When tracing is enabled for HKWD_LIB_MALL_COMMON, the input parameters, as well as return values for each call to malloc, realloc, and free subroutines are recorded in the trace subsystem. In addition to providing trace information about the malloc subsystem, Malloc Trace also performs checks of its internal data structures. If these structures have been corrupted, these checks will likely detect the corruption and provide temporal data, which is useful in problem determination.

When tracing is enabled for HKWD_LIB_MALL_INTERNAL and corruption is detected, information about the internal data structures are logged through the trace subsystem.

Enabling Malloc Trace

Malloc Trace is not enabled by default. It is enabled and configured by setting the MALLOCDEBUG environment variable. To enable Malloc Trace, set the MALLOCDEBUG environment variable by typing the following on the command line:

MALLOCDEBUG=trace

To enable other Malloc Debug features, set the MALLOCDEBUG environment variable as follows:

```
MALLOCDEBUG=[trace, other option]
```

Malloc Trace can be used concurrently with the following malloc subsystem capabilities:

- Debug Malloc (MALLOCTYPE=debug)
- Malloc Buckets (MALLOCTYPE=buckets)
- Malloc Multiheap (MALLOCMULTIHEAP=heap:n)
- Malloc 3.1 (MALLOCTYPE=3.1)

Malloc Log

Malloc Log is an optional extension of the malloc subsystem, enabling the user to obtain information showing the number of active allocations of a given size and stack traceback made by the malloc subsystem. This data can then be used in problem determination and performance analysis.

Malloc Log is not enabled by default, but can be enabled and configured prior to process startup by setting the MALLOCDEBUG environment variable.

Data Recorded in the Malloc Log

Malloc Log records the following data for each malloc or realloc subroutine invocation:

- · The size of the allocation.
- The stack traceback of the invocation. The depth of the traceback that is recorded is a configurable
- The number of currently active allocations that match the size and stack traceback.

The data is stored into the following global structure:

```
struct malloc log * malloc log table;
#ifndef MALLOC LOG STACKDEPTH
#define MALLOC LOG STACKDEPTH 4
#endif
struct malloc log {
  size t size;
   size t cnt;
  uintptr t callers[MALLOC LOG STACKDEPTH];
size_t malloc_log_size;
```

The size of the **malloc log** structure can change. If the default call-stack depth is greater than 4, the structure will have a larger size. The current size of the malloc log structure is stored in the globally exported malloc log size variable. A user can define the MALLOC LOG STACKDEPTH macro to the stack depth that was configured at process start time.

The **malloc log table** can be accessed in the following ways:

- · Using a debugging tool
- Using the **get malloc log** API as follows:

```
#include malloc.h
size t get malloc log (void *addr, void *buf, size t bufsize);
```

This function copies the data from malloc_log_table into the provided buffer. The data can then be accessed without modifying the malloc_log_table. The data represents a snapshot of the malloc log data for that moment of time.

Using the get_malloc_log_live API as follows:

```
#include malloc.h
struct malloc_log * get_malloc_log_live (void *addr);
```

The advantage of this method is that no data needs to be copied, therefore performance suffers less. Disadvantages of this method are that the data referenced is volatile and the data may not encompass the entire malloc subsystem, depending on which malloc algorithm is being used.

To clear all existing data from the malloc log tables, use the **reset malloc log** API as follows:

```
#include malloc.h
void reset_malloc_log(void *addr);
```

Enabling Malloc Log

Malloc Log is not enabled by default. To enable Malloc Log with the default settings, set the MALLOCDEBUG environment variable as follows:

MALLOCDEBUG=1og

To enable Malloc Log with user-specified configuration options, set the MALLOCDEBUG environment variable as follows:

```
MALLOCDEBUG=log:records per heap:stack depth
```

Note: The records per heap and stack depth parameters must be specified in order. Leaving a value blank will result in setting that parameter to the default value.

The predefined **MALLOCDEBUG** configuration options include the following:

```
records per heap
```

Used to specify the number of Malloc Log records that are stored for each heap. This parameter affects the amount of memory that is used by Malloc Log. The default value is 4096, the maximum value is 65535.

stack_depth

Used to specify the depth of the function-call stack that is recorded for each allocation. This parameter affects the amount of memory used by Malloc Log, as well as the size of the malloc log structure. The default value is 4, the maximum is 32.

Limitations

The performance of all programs can degrade when Malloc Log is enabled, due to the cost of storing data to memory. Memory usage will also increase.

Malloc Disclaim

Malloc Disclaim is an optional extension of the malloc subsystem, providing the user with a means to enable the automatic disclaiming of memory returned by the free subroutine. This is useful in instances where a process has a high paging-space usage, but is not actually using the memory.

Malloc Disclaim is not enabled by default. It can be enabled and configured prior to process startup through the MALLOCDISCLAIM environment variable, as follows:

Chapter 19. Packaging Software for Installation

This article provides information about preparing applications to be installed using the 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.

A software product installation package is a 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:

- "Installation Procedure Requirements"
- "Package Control Information Requirements" on page 416
- · "Package Partitioning Requirements" on page 416
- "Software Product Packaging Parts" on page 417
- · "Format of a Software Package" on page 416
- "Package and Fileset Naming Conventions" on page 417
- · "Fileset Revision Level Identification" on page 419
- · "Contents of a Software Package" on page 419
- "The lpp_name Package Information File" on page 420
- "The liblpp.a Installation Control Library File" on page 430
- "Further Description of Installation Control Files" on page 433
- "Installation Control Files Specifically for Repackaged Products" on page 438
- "Installation Files for Supplemental Disk Subsystems" on page 440
- "Format of Distribution Media" on page 440
- "The Table of Contents File" on page 442
- "The installp Processing of Product Packages" on page 443

Note: 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 installation package to use this service, see Chapter 20, "Documentation Library Service," on page 451.

Installation Procedure Requirements

• Installation must *not* require user interaction. Product configuration requiring user interaction must occur before or after installation.

- All installations of independent filesets 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

Package control information must:

- · Specify all installation requirements the filesets have on other filesets.
- · Specify all file system size requirements for the fileset installation.

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 440 for information about the format used for product packages on each type of media.

Package Partitioning Requirements

In order to support diskless or dataless 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** or **/opt** 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 diskless or dataless client system. The machine—specific (root) part should include everything that is not in the **/usr** or **/opt** file systems.

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 (**Islpp**) and verify (**Ippchk**) 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 name, owner, group, permission, checksum, and size 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
- · Maintenance level and APAR information

The destination directory and installer for non-installed packaged software, where applicable.

Software Product Packaging Parts

In order to support installation in the client/server environment, the installation packaging is divided in the following parts:

Contains the part of the product that can be shared among several machines with compatible usr

hardware architectures. For a standard system, these files are stored in the /usr or /opt file tree.

Contains the part of the product that cannot be shared among machines. Each client must have its root

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.

Sample File System Guide for Package Partitioning

Following is a brief description of 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

Machine configuration files such as hosts and passwd /etc

System utilities needed to boot the system /sbin System-specific data files and log files /var

Some usr-part directories and their contents include:

/usr/bin Commands and scripts (ordinary executables)

/usr/sbin System administration commands

/usr/include

/usr/lib Libraries, non-user commands, and architecture-dependent data

Libraries, non-user commands, and scripts normally associated with non-operating system /opt

products.

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. If a package has only one installable fileset, the fileset name can be the same as the PackageName. All package names must be unique.
- · A fileset name has the form:

ProductName.PackageName.FilesetName.extension

where:

- ProductName identifies the product or solution group.
- PackageName identifies a functional group within the product.
- FilesetName (optional) identifies a specific functional set of files and libraries to be installed.
- Extension (optional) further describes the contents. See "Fileset Extension Naming Conventions" on page 418 for more information.
- · A fileset name contains more than one character and begins with a letter.

- · All characters in the fileset name must be ASCII characters. Valid characters are upper and lower case letters, digits, underscores (), plus signs (+), and minus signs. The period (.) is used as a separator in the fileset name.
- A fileset name cannot end with a period or dot.
- 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

Compatibility code that may be removed in a future release .compat

.diag Diagnostics support

Fonts .fnt

.help. Language Common Desktop Environment (CDE) help files for a particular language

.loc.language

.msg. Language Message files for a particular language

Run-time environment or minimum set for a product .rte

.ucode Microcode

Special Naming Considerations for Device Driver Packaging

The configuration manager command (cfgmgr) automatically installs software support for detectable devices that are available on the installation media and packaged with the following naming convention: devices.BusTypeID.CardID.Etension

where:

- BusTypeID specifies the type of bus to which the card attaches (for example, pci for PCI
- CardID specifies the unique hexadecimal identifier associated with the card type
- · Extension specifies the part of the driver that is included (for example, rte is the extension for runtime and diag is the extension for diagnostics.)

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 (instreg) 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. The full path for the file name cannot be more than 128 characters.

Fileset Revision Level Identification

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.Modification.FixLevel

where:

- Version is a numeric field of 1 to 2 digits that identifies the version number.
- · Release is a numeric field of 1 to 2 digits that identifies the release number.
- Modification is a numeric field of 1 to 4 digits that identifies the modification level.
- FixLevel is a numeric field of 1 to 4 digits that identifies the fix level.

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 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, 5.2.0.0 is a newer level than 4.3.0.0).

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 PackageNamelFilesetNamelFilesetLevel.

The usr part of an installation or update package contains the following installation control files:

- ./Ipp_name: This file provides information about the software package to be installed or updated. For performance reasons, the **Ipp_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 420 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 Library File" on page 430 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.

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: The file from the root part of the package was restored under an <code>inst_root</code> directory. Files installed for the machine-dependent root part of a package are restored relative to an <code>inst_root</code> 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 <code>inst_root</code> 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.

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 3.1
			• 3 - AIX 3.2
			• 4 - AIX 4.1 and later

built. The values are: R - RISC 1 - Intel N - Neutral 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 ME - Maintenance level update Mew 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 To Volume The level of the fileset to be installed. The format is: Version. Release. Modification. FixLevel 7. Volume Integer White space Indicates the volume number where the fileset is located if shipped on multivolume media. 8. Bosboot Character White space Indicates whether a bosboot is needed following the installation. The values are: N - Do not invoke bosboot D - Invoke bosbo	2. Platform	Character	White space	Indicates the platform for which this package was
. I - Intel . N - Notural 3. Package Type Character White space Jindicates whether this is an installation or update package and what type. The values are: . I - Installation . S - Single update required . ML - Maintenance level update . Maintenance level update				
S. 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 - SR - Single update - SR - Single update - ML - Maintenance level				• R - RISC
S. Package Type Character White space Indicates whether this is an installation or update package and what type. The values are: Installation S. Single update required				I - Intel
package and what type. The values are: I Installation S Single update SR - Single update required ML - Maintenance level update The name of the software package (PackageName) Revalues Revalu				N - Neutral
S. Single update SR - Single update SR - Single update required Mt. Maintenance level update Mthe space Mthe space Indicates the beginning of the repeatable sections of filesest-specific data. The complete name of the fileset. This field begins the heading information for the fileset or fileset update. Shown in Description Column Mthite space Mthite space Indicates the volume number where the fileset is located if shipped on multivolume media. B. Bosboot Character White space Mthite space Indicates whether a bosboot is needed following the installation. The values are: N - Do not invoke bosboot Indicates the parts included in the fileset or fileset update. The values are: N - Do not invoke bosboot Indicates the parts included in the fileset or fileset update. The values are: N - B - usr and root part U - usr part only The level of the fileset of the fileset of the fileset of fileset update. The values are: B - usr and root part U - usr part only The level of the fileset of the fileset of the fileset of fileset update. The values are: B - usr and root part U - usr part only The values are: B - usr and root part U - usr part only The description is limited to 60 characters. The description is limited to 60 characters. Power line Character New line Coptional) Additional comments. Indicates the beginning of the body of the fileset information The fileset and License and License Agreement information. See Size and License Agreement Information. See Size and License Agreement Information See Size and License Agreement Information between requisite and size information. The fileset specified later in this article for detailed description.	3. Package Type	Character	White space	·
SR - Single update required ML - Maintenance level update The name of the software package (PackageName). Remainder of the software package (PackageName). Remainder of the software package (PackageName). SFileset name Character White space White space Character White space Shown in Description column New line The level of the fileset. This field begins the heading information for the fileset or fileset update. The level of the fileset to be installed. The format is: Version. Release. Modification. FixLevel Indicates the volume number where the fileset is located if shipped on multivolume media. Bosboot Character White space Indicates whether a bosboot is needed following the installation. The values are: Nounce bosboot Character White space Indicates the parts included in the fileset or fileset update. The values are: Bours and root part Uousr part only Luarguage Character White space Should be set to the language displayed if the Clocale is selected. Generally set to en_US. Language Character White space Should be set to the language displayed if the Clocale is selected. Generally set to en_US. Character New line Character New line Indicates the beginning of the body of the fileset information Poscribed following table New line Indicates the separation between requisite and size information. Size requirements by directory and license agreement information. See Size and License Agreement Information between requisite and size information See Size and License Agreement Information See Size and License Agree				I- Installation
ML - Maintenance level update				S- Single update
4. PackageName Character White space The name of the software package (PackageName). 5. Fileset name Character White space The complete name of the fileset. This field begins the heading information for the fileset or 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 The space The level of the fileset to be installed. The format is: Version.Release.Modification.FixLevel or the fileset pack. 7. Volume Integer White space Indicates the volume number where the fileset is located if shipped on multivolume media. 8. Bosboot Character White space Indicates whether a bosboot is needed following the installation. The values are: N - Do not 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 U -usr part only 10. Language Character White space Should be set to the language displayed if the C locale is selected. Generally set to en_US. 11. Description Character # or new line Fileset description. The description is limited to 60 characters. 12. Comments Character New line (Optional) Additional comments. I New line Indicates the beginning of the body of the fileset information. 13. Requisite information. White space Indicates the separation between requisite and size information. New line Indicates the separation between requisite and size information. 14. Size and License Agreement Information Section later in this chapter information Section later in this article for detailed description.				SR - Single update required
Recommendation Reco				ML - Maintenance level update
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 To Volume Integer White space White space Indicates the volume number where the fileset is located if shipped on multivolume media. 8. Bosboot Character White space White space Indicates whether a bosboot is needed following the installation. The values are: N - Do not invoke bosboot b - Invoke bosboot The values are: N - B - usr and root part U-usr part only 10. Language Character White space White space White space White space Indicates the volume number where the fileset is located if shipped on multivolume media. N - Do not invoke bosboot b - Invoke bosboot Should be set to the language displayed if the C locate is selected. Generally set to en_US. 11. Description Character Whole space Should be set to the language displayed if the C locate is selected. Generally set to en_US. 12. Comments Character New line Indicates the beginning of the body of the fileset information. New line Described following table New line Optional) Additional comments. New line Optional installation dependencies the fileset has on other filesets and fileset updates. See the section following this table for detailed description. 14. Size and License Agreement information Section later in this chapter Indicates the separation between requisite and size information. See Size and License Agreement information Section later in this article for detailed description is limited to for detailed description this chapter Indicates the separation between requisite and size information.	4. PackageName	Character	White space	
begins the heading information for the fileset or fileset update. Shown in Description column 7. Volume Integer White space Indicates the volume number where the fileset is located if shipped on multivolume media. 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 Character White space Indicates whether a bosboot is needed following the installation. The values are: N - Do not invoke bosboot b - Invoke bosboot Character White space Indicates the parts included in the fileset or fileset update. The values are: B -usr and root part U -usr part only 10. Language Character White space Should be set to the language displayed if the C locale is selected. Generally set to en_US. 11. Description Character # or new line Fileset description. The description is limited to 60 characters. 12. Comments Character New line (Optional) Additional comments. Indicates the beginning of the body of the fileset information 13. Requisite information Wew line New line (Optional) Installation dependencies the fileset has on other filesets and fileset updates. See the section following this table for detailed description. 14. Size and License Agreement information. See Size and License Agreement information. See Size and License Agreement information Section later in this article for detailed description.		{	New line	
Description column Integer White space Indicates the volume number where the fileset is located if shipped on multivolume media.	5.Fileset name	Character	White space	begins the heading information for the fileset or
Bosboot Character White space Indicates whether a bosboot is needed following the installation. The values are: N - Do not invoke bosboot b - Invoke bosboot b - Invoke bosboot New line Indicates the parts included in the fileset or fileset update. The values are: B - us rand root part U - usr part only	6. Level	Description	White space	
the installation. The values are: N - Do not invoke bosboot Character White space Indicates the parts included in the fileset or fileset update. The values are: B - U - usr part only Character White space Character White space Should be set to the language displayed if the C locale is selected. Generally set to en_US. Character # or new line Fileset description. The description is limited to 60 characters. Character New line (Optional) Additional comments. Indicates the beginning of the body of the fileset information. See Size and License Agreement information. New line New line New line Indicates the separation between requisite and size information. See Size and License Agreement information Section later in this article for detailed description. New line Indicates the separation between size and	7. Volume	Integer	White space	
9. Content Character White space Indicates the parts included in the fileset or fileset update. The values are: B -usr and root part U -usr part only 10. Language Character White space Should be set to the language displayed if the C locale is selected. Generally set to en_US. 11. Description Character # or new line Fileset description. The description is limited to 60 characters. 12. Comments Character New line Indicates the beginning of the body of the fileset information. 13. Requisite information Described following table New line Vew line Indicates the separation 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. 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	8. Bosboot	Character	White space	
9. Content Character White space Indicates the parts included in the fileset or fileset update. The values are: B -usr and root part U -usr part only 10. Language Character White space Should be set to the language displayed if the C locale is selected. Generally set to en_US. 11. Description Character # or new line Fileset description. The description is limited to 60 characters. 12. Comments Character New line (Optional) Additional comments. [New line Indicates the beginning of the body of the fileset information. 13. Requisite information Pescribed following table 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. See Size and License Agreement Information See Size and License Agreement Information See Size and License Agreement Information Seetion later in this article for detailed description. New line Indicates the separation between size and				N - Do not invoke bosboot
update. The values are: B -usr and root part U -usr part only 10. Language Character White space Should be set to the language displayed if the C locale is selected. Generally set to en_US. 11. Description Character # or new line Fileset description. The description is limited to 60 characters. (Optional) Additional comments. Indicates the beginning of the body of the fileset information. New line Oescribed following table Pescribed following table New line 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 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 License Agreement Information Section later in this article for detailed description. New line Indicates the separation between size and				b - Invoke bosboot
• U -usr part only 10. Language Character White space Should be set to the language displayed if the C locale is selected. Generally set to en_US. 11. Description Character # or new line Fileset description. The description is limited to 60 characters. 12. Comments Character New line (Optional) Additional comments. [New line Indicates the beginning of the body of the fileset information. 13. Requisite information Poscribed following table (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 Section later in this chapter Agreement Information Section later in this article for detailed description. New line Indicates the separation between size and License Agreement Information Section later in this article for detailed description. Indicates the separation between size and License Agreement Information Section later in this article for detailed description.	9. Content	Character	White space	·
10. Language Character White space Should be set to the language displayed if the C locale is selected. Generally set to en_US. 11. Description Character # or new line Fileset description. The description is limited to 60 characters. 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 See Size 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				B -usr and root part
locale is selected. Generally set to en_US.				U -usr part only
characters. 12. Comments Character New line (Optional) Additional comments. Indicates the beginning of the body of the fileset information. 13. Requisite Indicates the beginning of the body of the fileset information. 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 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	10. Language	Character	White space	
[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 this chapter 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	11. Description	Character	# or new line	· · · · · · · · · · · · · · · · · · ·
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 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	12. Comments	Character	New line	(Optional) Additional comments.
information following table 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		[New line	
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			New line	
Agreement information this chapter agreement information. See Size and License Agreement Information Section later in this article for detailed description. New line Indicates the separation between size and		%	New line	· · · · · · · · · · · · · · · · · · ·
l '	Agreement		New line	agreement information. See Size and License Agreement Information Section later in this article
		%	New line	

	%	New line	Indicates the separation between licensing and supersede information.
15. Supersede Information	Described later in article	New line	Information regarding a prior fileset that this fileset replaces.
	%	New line	Indicates the separation between licensing and fix 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 23
| | | |
                                   7 8 9 10
                                                   11
4 RSfarm.apps {
                        | | |
 5--> farm.apps.hog04.01.0000.0003 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--> ranch.hog 4.1.0.0
16--> IX51366 Hogs producing eggs.
16--> IX81360 Piglets have too many ears.
```

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 Ipp name file. If the -q 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 **installa** 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 for reasons described in "Supersede Information Section" on page 428, 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

A prerequisite indicates that the specified fileset must be installed at the specified fileset level or at a higher level before the current fileset can 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 prerequisite on a fileset within the same package is not guaranteed.

Syntax

*prereq Fileset RequiredFilesetLevel

Alternate Syntax

Fileset RequiredFilesetLevel

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.

Corequisite

A corequisite 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 corequisites that are not met. A corequisite can be used to indicate requisites between filesets within the same package.

Syntax

*coreq Fileset RequiredFilesetLevel

If Requisite

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 plum.tree (1.1.0.0) 1.1.2.3
```

Syntax

*ifreq Fileset [(InstalledFilesetLevel)] RequiredFilesetLevel

If the plum.tree fileset is not already installed, this example does not cause it to be installed. If the plum. tree 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 plum.tree 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 Installed Fileset Level and the Required Fileset Level are assumed to be the same. If the Fix Level 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

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 always contain an installed requisite for another fileset in its product.

Syntax

*instreg Fileset RequiredFilesetLevel

Group Requisite

A group requisite indicates that different requisite conditions can satisfy the requisite. A group requisite can contain prerequisites, corequisites, 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.

Syntax

>Number { RequisiteExpressionList }

Requisite Information Section Examples

1. The following example illustrates the use of corequisites. 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:

```
*coreg 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:

```
*prereg database.rte 1.2.0.0
*coreq spreadsheet.rte 1.3.1.0
*ifreq wordprocessorA.rte (4.1.0.0) 4.1.1.1
*ifreq wordprocessorB.rte 4.1.1.1
```

The database.rte fileset must be installed at level 1.2.0.0 or higher before the new.fileset.rte fileset can be installed. If database.rte and new.fileset.rte are installed in the same installation session, the installation program installs the database fileset before the new.fileset.rte fileset.

The spreadsheet.rte fileset must be installed at level 1.3.1.0 or higher for the new.fileset.rte fileset to function properly. The spreadsheet.rte 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.rte fileset is not installed by the end of the installation session, a warning message will be issued stating that the corequisite is not met.

If the wordprocessorA.rte fileset is installed (or being installed with new.fileset.rte) at level 4.1.0.0, then the wordprocessorA.rte fileset update must be installed at level 4.1.1.1 or higher.

If the wordprocessorB.rte fileset is installed (or being installed with new.fileset.rte) at level 4.1.1.0, then the wordprocessorB.rte 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:

```
*instreg Super.Widget 2.1.0.0
```

The Super.msg.fr FR.Widget fileset cannot be installed automatically when the Super.Widget fileset is not installed. The Super.msg.fr FR.Widget fileset can be installed when the Super.Widget fileset is not installed if the fileset is listed explicitly on the list of filesets to be 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.rte fileset must be at level 1.2.0.0 or higher or the spreadsheet 2.rte fileset must be at level 1.3.0.0 or higher.

```
*prereq spreadsheet 1.rte 1.2.0.0
*prereq spreadsheet_2.rte 1.3.0.0
```

Size and License Agreement Information

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 **PermanentSpace** The full path name of the directory that has size requirements.

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 re-link 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 40 20 /lib 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/PackageNamelFilesetName/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.

License Agreement Information

A recent addition to the AIX install process allows product owners to require that a customer sign a license agreement before the install of the product. The install p command reads a fileset's lpp name file as usual for any image. If there are LAF or LAR entries in the size section of the *lpp name* file, the **installp** command calls the inulag command which displays the license in a window and records the customer acceptance of the license. If the customer refuses to accept the license, the install is halted for that product.

Where to place the license file: Placement of the license files is up to the product owner. However, it is strongly suggested that the license be placed in /usr/swlag/LANG. The suggested name for the License Agreement file is given as ProductName VersionRelease.la. There is no requirement to use this name or location. The installp command and this packaging is simply suppling a method to deliver the information to customers. All tools and file content must be provided by the product.

Translation requirements for the license file: If there is a requirement that the License Agreement files be translated into the supported languages, it is recommended that the files be separated into different files by language. This translation can require that several files be created.

How to ship the license file: A product can ship the license file as part of its main product or in a separate fileset for the product. Many products are choosing to create a separate fileset for shipping just the license files. This enables a product with many different features to create one license file and fileset that can ship on all the media required for each feature instead of having to include the files in several different features. The current suggestions for the fileset name are lpp.license or lpp.loc.license. Most products are currently using the first suggestion. If you want the license fileset to be hidden from the customer in standard installs, use 1pp.1oc.1icense because the license fileset does not need to be selected for installation.

How to package the license file: The file itself is never listed in the fileset.al or the fileset.inventory files for the fileset. The installp command finds the license because of the entry in the size section of the *ProductName* file. The types of entries are:

LAF

License Agreement File tells the installp command that this particular license file is shipped in this fileset.

License agreement files are indicated by a size section entry with:

LAF<lang>license_file size

Term Meaning

LAF stands for License Agreement File

<lang></lang>	language to which the file is translated. These are typically entries like en_US, fr_FR, Ja_JP, and zh_TW. If <lang> is not specified, then the agreement file is assumed to be non-translated and encoded as ASCII. If the license agreement file is translated, <lang> must be part of the path so that the requirement entry can be associated with the file.</lang></lang>
license_file	is the fullpath to the license file as it will be found in the image and placed on the system. The suggested path takes the form /usr/swlag/en_US/ProductName_VersionRelease.la
size	This is the real size in 512-byte blocks of the license file so as to ensure that installp has allowed enough space to place the license file on the system.

LAR

The License Agreement Requisite tells the installp command that this fileset requires the installation of the listed license agreement file. This is not the same as a prerequisite because it is in the file and not in a fileset. Files and filesets have different formats and different purposes. They should not be confused.

Term	Meaning
LAR	stands for License Agreement Requisite
req_license_file	The full path to the license file required to install this fileset. Typically these entries use %L in the path instead of the actual language name to allow for the appropriate file to be viewed without forcing the customer to view all of the languages.

Generally one fileset ships the files, and contains all of the LAF entries. The other filesets in the product which require this license only ship the LAR entry. The fileset which ships the LAF entries also contains the files listed at their full path location in the BFF image, but the files are not listed in the fileset.al or the fileset.inventory files for the fileset. The design used for electronic licenses does not require that the files be registered to the SWVPD. The **installp** command:

- 1. Finds a requirement on the file.
- 2. Checks the system to see if it was accepted.
- 3. If the file was not accepted
 - a. Finds the fileset that ships the file.
 - b. Extracts (restores) only that file from the BFF image.
 - c. Display the file to the customer.

LAF Fileset Example: The following is an example of a fileset that ships the license file:

```
iced.tea.loc.license 03.01.0000.0000 1 N U en US IcedTea Recipe License Information
INSTWORK 16 160
LAF/usr/swlag/de DE/iced.tea.la 24
LAF/usr/swlag/DE DE/iced.tea.la 24
LAF/usr/swlag/en_US/iced.tea.la 24
LAF/usr/swlag/EN US/iced.tea.la 24
LAF/usr/swlag/es ES/iced.tea.la 24
LAF/usr/swlag/ES ES/iced.tea.la 24
LAF/usr/swlag/fr FR/iced.tea.la 24
LAF/usr/swlag/FR FR/iced.tea.la 24
LAF/usr/swlag/it_IT/iced.tea.la 24
LAF/usr/swlag/IT_IT/iced.tea.la 24
LAF/usr/swlag/ja_JP/iced.tea.la 24
LAF/usr/swlag/JA JP/iced.tea.la 32
LAF/usr/swlag/Ja JP/iced.tea.la 24
```

```
LAF/usr/swlag/ko KR/iced.tea.la 24
LAF/usr/swlag/KO KR/iced.tea.la 24
LAF/usr/swlag/pt BR/iced.tea.la 24
LAF/usr/swlag/PT_BR/iced.tea.la 24
LAF/usr/swlag/ru RU/iced.tea.la 24
LAF/usr/swlag/RU RU/iced.tea.la 48
LAF/usr/swlag/zh CN/iced.tea.la 16
LAF/usr/swlag/zh TW/iced.tea.la 16
LAF/usr/swlag/Zh_TW/iced.tea.la 16
LAF/usr/swlag/ZH_TW/iced.tea.la 24
%
%
]
```

LAR Fileset example: The following is an example of a fileset which ships the license requirement to the license file:

```
iced.tea.server 03.01.0000.0010 1 N B en US Iced Tea Recipe Group
*prereq bos.net.tcp.client 5.1.0.10
*coreg iced.tea.tools 5.1.0.10
*coreg Java14.sdk 1.4.0.1
/usr/bin 624
/usr/lib/objrepos 24
/usr/include 16
/usr/include/sys 56
/usr/lpp/iced.tea 22
/usr/samples/iced.tea 8
/usr/samples/iced.tea/server 504
/usr/lpp/iced.tea/inst root/etc/tea 8
/usr/iced.tea 8
/usr/lpp/iced.tea/inst root/etc/tea/Top 8
INSTWORK 208 96
/lpp/iced.tea 104
/etc/tea 8
/etc/objrepos 8
/etc/tea/Top 8
/tmp 0 6
LAR/usr/swlag/%L/iced.tea.la 0
%
%
1
```

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 4.1-formatted fileset base installation packages and AIX 3.2-formatted fileset update packages.

A newer fileset supersedes any older version of that fileset unless the supersedes section of the **Ipp 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 supersedesfarm.apps.hog 4.1.0.1 and farm.apps.hog 4.1.0.3.

Supersede Section for Fileset Installation Levels (Base Levels)

An AIX 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.

Indicates the fileset can be used to satisfy the requisites of another fileset. A Compatibility Entry

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 **Ipp 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.

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 -q flag is specified, the request is satisfied by the newest superseding update on the installation media.
 - When the -q 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 theinstallp command without the -g flag.
- If an update and a superseding update (both on the installation media) are requested for installation, the installp command installs the newer update only.
 - In this 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. The fix information section entries contain a fix keyword and a 60 characters or less description of the problem fixed. A fix keyword is a 16 characters or less identifier corresponding to the fix. Fix keywords beginning with ix, iy, IY and IX are reserved for use by the operating system manufacturer.

A maintenance level is a fix that is a major update level. 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 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 with the -q flag on systems later than AIX 4.3 to ensure that a 32 bit archive is created. 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. This file can only

be included in the root part of a fileset.

Fileset.fixdata Stanza format file. This file contains information about the fixes contained in a fileset or fileset

update. For more information on this file, see "The Fileset.fixdata File" on page 434.

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. For more information on this file, see "The Fileset.inventory

File" on page 435.

Fileset.namelist List of obsolete filesets and current fileset (if applicable) 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 or Fileset.*.odmadd

Stanzas to be added to ODM (Object Data Manager) databases. For more information, see

the **odmadd** command

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.size This file contains the space requirements for the files included in this fileset as described

earlier in this section.

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.

Ipp.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

Ipp.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**, **Ipp.doc**, **Ipp.instr**, or **Ipp.lps.**

productid Product identification file. This 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 or Fileset.config_u Fileset.odmdel orFileset.*.odmdel Fileset.pre_d	Modifies configuration near the end of the default installation or update process. <i>Fileset.</i> config is used during installation processing only. 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. Indicates whether a fileset may be removed during deinstall. The program must return a value of 0 (zero) if the fileset can be removed. Filesets are removable by default. The program should generate error messages indicating why the fileset is not removable.
Fileset.pre_i or	Runs prior to restoring or saving the files from the apply list in the package, but after
Fileset.pre_u	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 or Fileset.post_u	Runs after restoring the files from the apply list of the fileset installation or update.
Fileset.unconfig Fileset.unconfig_u Fileset.unodmadd	Undoes configuration processing performed in the installation or update during a deinstall or a reject of a fileset. <i>Fileset.</i> unconfig is used during deinstallation processing only. Deletes entries that were added to ODM databases during the installation or update.
Fileset.unpost_i_0 orFileset.unpost_u	Undoes processing performed following restoring the files from the apply list in the installation or update during a deinstall or a reject of a fileset.
Fileset.unpre_i or Fileset.unpre_u	Undoes processing performed prior to restoring the files from the apply list in the installation or update during a deinstall or a reject of a fileset.

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.

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 install p command, the instal or update executable provided with a software package must:

· Process all of the filesets in the software package. It can either process the installation for all the filesets or invoke other executables for each fileset.

instal

- 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 449 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.cfafiles 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 or root).

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.

During the Fileset.post i processing, the product-provided executables merge necessary auto_merge

> 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.

Leaves the installed new version of the file on the system and keeps the old version of user_merge

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. This keyword should be avoided if

possible.

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 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

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 (change.rte) that has one file that can be configured. So, in the root change.rte.cfgfiles, there is one file listed:

```
/etc/change user user merge
```

When migrating from your old fileset (change.obj) to change.rte, you cannot preserve this file because the format has changed. However, when migrating from an older level change.rte to a newer level change.rte, the file can be preserved. In this case, you might want to create a change.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/change user file, they are saved.

The root **change.bar.post_i** script could be as follows:

```
#! /bin/ksh
rc=0
grep -q change.rte $INSTALLED LIST
if [\$? = 0]
mv $MIGSAVE/etc/change user/ /etc/change user
rc=1
fi
exit $rc
```

\$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 438) 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 leafpreserve
./etc/cfg_pudding hold new
./etc/cfg newtonpreserve
```

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 \mid p\}
filesets = FilesetName FilesetLevel
[FilesetName FilesetLevel ...]
[symptom = [Symptom]]
```

- FixKeyword cannot 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. fix data 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
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

Command that uses the Fileset, inventory file to enter the file name, product sysck 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 GB in size. If you ship a file that is greater than 2 GB, 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=Value. Each attribute is described on a separate line.

A *Fileset*.**inventory** stanza has the following format:

inventory:

type = type

class = fileset,inventory,apply,C2 exclude

owner = owner_name
group = group name

mode = TCB | SUID | SUID, permissions

target = fullpath filename

link = fullpath to hardlink [additional hardlinks]

size =<black> VOLATILE | size checksum =<black> VOLATILE | "checksum"

The following list describes the valid attributes of a file:

file_name The full path to the file or link relative to root (./) followed immediately by a colon (:) and then a new

line. The maximum length for this full pathname is 127 characters.

type Type of the file_name entry where a valid type is one of the following:

Type Items Meaning

FILE Standard file

DIRECTORY Directory

SYMLINK The fullpath to a symbolic link

class Specifies how the file_name is to be referenced during installation. This field must have at least two

of the following keywords:

Type Items Meaning

fileset Indicates the fileset to which the file belongs.

inventory Indicates that the file remains after the installation is complete.

The file is added to the inventory SWVPD database. This should

not be used with an A type file in the fileset.il file.

apply Indicates that the file is to be restored from the install media.

The *file_name* field is listed in the apply list (*fileset.al*). This should not be used for an I type file in the *fileset.il* file.

C2_exclude Indicates that this file should be excluded from execution on a

C2 Secure system. If this flag is used, the file should also be

listed in the fileset.tcb file.

owner Specifies the file owner after installation. Do not use the uid for this field. The attribute value must

be the owner name and must be no more than 8 characters.

group Specifies the file group. Do not use the gid for this field. The attribute value must be the group

name and must be no more than 8 characters.

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". If the file is SUID root or

SGID system, then the file must be TCB.

SUID This file has the set user ID bit set. This has no meaning on a

DIRECTORY entry.

SGID This file has the set group ID bit set. If set on a DIRECTORY

entry, this forces all files created in that directory to have the

same group as the directory.

permissions Must be a three-digit octal number, for example, 644.

Note: If the type is SYMLINK, the mode must be 777. No other

entries are valid.

linkLists any hard links to this file. If multiple hard links exist, each fullpath to a hard link is separated by a comma. The hardlinks must reside in the same directory as the source file. If the type of the entry

is SYMLINK, link is not valid. The maximum length of the full path name is 127 characters.

target	Valid only for type=SYMLINK. This is the full path filename to the target of the link. If the link being	

created is from /usr/bin to /bin, the filename would be /bin and the target would be /usr/bin. The

maximum length of the full path name is 127 characters.

size Mode Items Meaning

> blank If this field is blank, the size of the filename is determined at the

> > time of installation. The drawback to using this option is that if a

file was corrupted during installation, the customer is not

VOLATILE If the file size is expected to change through normal operation,

the value for this attribute must be VOLATILE.

size The exact size of the file.

Note: Do not include the size field for DIRECTORY or SYMLINK entries.

checksum

Mode Items Meaning

blank If this field is blank, the value returned from the sum -r

command to be placed in the inventory SWVPD database when

the file is installed.

VOLATILE 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 The exact sum of the file as returned by the sum -r command.

This should be placed in double quotes.

Note: Do not include the checksum field for DIRECTORY or SYMLINK entries.

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.

fileset.inventory Example

The following fileset.inventory example demonstrates the use of type.

```
/usr/bin:
```

owner = bingroup = bin

mode = 755

type = directory

class = apply,inventory,bos.rte

/usr/bin/tcbck:

owner = root

group = security

mode = TCB, SUID, 550

type = file

class = apply,inventory,bos.rte.security

size = 99770

checksum = "17077 98 "

/usr/sbin/tsm:

owner = root

group = security

mode = TCB, SUID, 555

links = /usr/sbin/getty,/usr/sbin/login class = apply,inventory,bos.rte,security

size = 55086

54 " checksum = "57960

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

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.com GooseCreek.rte CedarBayou.stream

While installing the Baytown.com 2.3.0.0 fileset, the **installp** command finds that Pelly.com 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

This file is 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** or **root** 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 Law Practice.client.namelist file in the liblpp.a file for the Law Practice 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.

Example 3:

Filesets bage1.shop.rte and bread.shop.rte have been shipping separately for years. Now, bagel.shop.rte is going to ship as a part of bread.shop.rte. For this to happen, the bread.shop.rte.namelist file would look like:

bread.shop.rte bagel.shop.rte

You must also ship an empty bread.shop.rte.rm inv file to indicate that all files from the bagel.shop.rte fileset should be removed from the system.

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 associated with the listed file:

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.

Note: Links need to be listed twice, once as a standalone stanza and once as an attribute to the file that is linked to.

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. TheRussia.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 /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" on page 441
- "CD-ROM" on page 441
- "Diskette" on page 442

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 442 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 installation 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 442 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 /installp/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 /installp/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 ../ppc.
- The table of contents file (.toc) in the /installp/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.

AIX 5.2 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 /installp/mvCD containing the location of the product package images for A and B are vol%1/A.bff and vol%2/B.bff, respectively. The field in the table of contents file in /installp/ppc of volume 1 contains the location of A as A.bff. The field in the table of contents file in /installp/ppc of volume 2 contains the location of B as B.bff.

The CD-ROM directory structure for AIX 5.1 and later allows for specification of other installers, as well as multiple platforms.

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 <i>MonthDayHourMinuteSecondYear</i> . The <i>Hour</i> should be a value from 00 to 23. All date and time fields contain two digits. Thus, <i>Month</i> should be represented as 03 instead of 3, and <i>Year</i> 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 or later B -mksysb tape (invalid for use by installp)

The Table of Contents File			
Field Name	Format	Separator	Description
4. Location of product package image	Character	White space	For tape or diskette, this is a character string in the form: vvv:bbbbb:ssssssSee 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	Ipp_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:ssssss where each letter represents a digit and has the following meaning:

For tape

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

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 preexisting 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 were 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.

When a fileset is removed, the fileset and its updates are removed from the system Remove

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 **Ipp_name** product package information file for the package from the specified device.
- 2. Verify that the requested filesets exist on the installation media.
- 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. 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. 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/Ipp/Package_Name.
- 9. If this is an installation package rather than a fileset update package, run the /usr/lib/instl/rminstal script to do the following for each fileset being installed.

Note: Unless otherwise specified, files checked for existence must have been restored from the liblpp.a control file library.

- a. If Fileset.pre rm exists, run 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 the files and SWVPD information (except history) for Fileset.
- d. If Work Directory/Fileset.installed list exists, and Fileset.rm inv exists or Fileset.namelist contains more than one fileset or the only fileset listed in Fileset.namelist is bos.obj, then do the following:
 - 1) Remove files and SWVPD inventory information for files listed in the file Fileset.rm inv.
 - 2) Remove files and SWVPD inventory information for files listed in the file *Fileset*.inventory.
 - 3) Remove other SWVPD information for any filesets listed in Fileset.namelist which no longer have any SWVPD inventory information.
- e. 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.
- f. For each part of a product package (usr part only or usr followed by root)
 - 1) Set INUTREE (U for usr and M for root) and INUTEMPDIR (name of created temporary working directory environment variables.
 - 2) If an instal control program exists in the package directory (not recommended), run ./instal, otherwise, run the default script /usr/lib/instl/instal. If an instal control program does not exist in the package directory, set **INUSAVEDIR** environment variable.
 - 3) If an **update** control program exists in the package directory (not recommended), run ./update. If an update control program does not exist in the package directory, run the default script /usr/lib/instl/update.
 - 4) If a status file has been successfully created by instal or update, Use status file to determine the success or failure of each fileset. If a status file has not been created, assume all requested filesets in package failed to apply.
 - 5) If the apply operation for a fileset was successful, update the Software Vital Product Data (SWVPD), then register any associated license agreement requirements. If the apply operation for a fileset was not successful, run /usr/lib/instl/cleanup or the package-supplied lpp.cleanup from package directory to clean up the failed filesets.

Processing of the Default install or 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 "Further Description of Installation Control Files" on page 433.

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 the fileset is an update, Execute *Fileset.*pre u (pre update) if it exists. If the fileset is not an update, 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 this is an update, the files are specified to be saved, and the Ipp.acf archive control file exists. Save off the library archive members being updated.
 - d. If the processing is successful, append this fileset to the list to be installed in the **\$FILESETLIST.new** file.

- 2. If this is an update and file saving is specified, run **inusave** to save current versions of the files.
- 3. If you are processing the root part, run **inucp** to copy files from the apply list to root part. If you are not processing root part, run inurest to restore files from apply list for the usr part.
- 4. Do the following for each fileset listed in **\$FILESETLIST.new** file:

Note: 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. If so, export the INSTALLED_LIST and MIGSAVE environment variables. This is called a *migration*.
- b. If you are processing an update, invoke Fileset.post_u if it exists. If Fileset.post_u does not exist, invoke Fileset.post i if it exists.
- c. If Fileset.cfqfiles exists, run /usr/lib/instl/migrate cfq to handle processing of configuration files according to their specified handling method.
- d. Invoke sysck to add the information in the Fileset.inventory file to the Software Vital Product Database (SWVPD).
- e. 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 tcb enabled attribute is set in the /usr/lib/objrepos/PdAt ODM database.
- f. Invoke **errupdate** to add error templates if *Fileset*.**err** exists.
- g. Invoke trcupdate to add trace report format templates if Fileset.trc exists.
- h. If update or if Work_Directory/Fileset.installed_list exists, invoke each Fileset.odmdel and *Fileset.*.***odmdel** script to process ODM database deletion commands.
- i. Invoke odmadd on each existing Fileset.odmadd and Fileset.*.odmadd to add information to ODM databases.
- j. If this is an update, invoke Fileset.config u (fileset configuration update) if it exists. Otherwise, invoke Fileset.config (fileset configuration) if it exists.
- k. 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:
 - Ipp.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
 - 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 (for example, usr and root):
 - a. Set **INUTREE** (*U* for **usr** and *M* for **root**.) and **INUTEMPDIR** environment variables.
 - b. If reject control file exists in current directory (INULIBDIR), invoke ./lpp.reject. Otherwise, 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 433.

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 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 1i, because 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. Otherwise, invoke Fileset.unconfig if it exists.
 - c. Run any Fileset.*.unodmaddand/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. Invoketrcupdate 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 this is an update, invoke Fileset.unpost_u if it exists. Otherwise, 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 this is an update, invoke *Fileset*.unpre u if it exists. Otherwise, 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 (for example, usr or root):
 - 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 the deinstal control file exists in current directory, run the ./lpp.deinstalscript. If the deinstal control file does not exist in current directory, run the /usr/lib/instl/deinstal default script.
- 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 pre-installation scripts for all updates and the base level installation. Use of this file is *not* recommended.

- 3. If Fileset.unconfig d does not exist,
 - a. For each update for that fileset, do the following:
 - Run all Fileset.unconfig_u scripts to undo any update configuration processing.
 - Run all Fileset.*.unodmadd and Fileset.unodmadd to delete Object Data Manager (ODM) entries added during the update.
 - Run all Fileset.*.rodmadd and 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, do the following:
 - Run any Fileset.*.unodmaddand/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.
inulag	Acts as the front end to the subroutines to manage license agreements.
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 20. 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 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:

- 1. "Language Support" on page 452
- 2. "Writing your HTML Documents" on page 452
- 3. "Calling the Documentation Library Service From Your Documentation" on page 453
- 4. "Creating Indexes of your Documentation" on page 460
- 5. "Removing Indexes of your Documentation" on page 468
- 6. "Packaging your Application's Documentation" on page 468

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 5.2 National Language Support Guide and Reference.

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 © 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 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 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/userquide.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/userquide.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 information on View Definition Files and the Printfile tag, see Create a View Definition File on page 456.

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.
 - 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
 - Home/Search
- 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</tab>
	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
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

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 name my image.gif which is in the web server's /my images directory, and the title browser window to be My Documents, you might insert the following in your set configuration file: page_top_begin <html> <head> <title>My Documents</title> <body> <div align="CENTER"> </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</p></body></head></html>	
myimage.gif which is in the web server's /myimages directory, and the title browser window to be My Documents, you might insert the following in your set configuration file: page_top_begin <html> <head> <title>My Documents</title> <body> <div align="CENTER"> </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</p></body></head></html>	
<pre></pre>	of
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 that users can send mail to you, you might insert the following in your configuration file:	so
<pre>page_bottom_begin <hr/> <div align="CENTER"> Feedback </div> page_bottom_end</pre>	
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 nar myimage.gif which is in the web server's /myimages directory, and the title browser window to be Results of My Search, you might insert the following your view set configuration file:	of the
results_top_begin <html> <head> <fitle>Results of My Search</fitle> <body> <div align="CENTER"> </div> results_top_end</body></head></html>	
Note: If a results_title configuration file entry was specified, it will be igno	

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:
	<pre>results_bottom_begin <hr/> <div align="CENTER"> Feedback </div> results_bottom_end</pre>

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 452.
	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. <i>\$path</i> is the path to your printable file. This path must be a sub-directory of the <code>/usr/share/man/info</code> directory. The link <code>doc_link</code> is automatically placed by the library service in the machine's web server document directory. This link points the web server to the <code>/usr/share/man/info</code> directory and allows the web server to find your print file if it is installed under <code>/usr/share/man/info</code> .
	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 cannot 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). Example: Expand: Yes
Extra	Expand: No Specifies text that is to be displayed after the title, but that should not be a link
Extra	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 lcon 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
V.12	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 an HTML link when
	displayed.

Field	Description and Examples
Version	The version of this entry. When registering documentation, a higher numbered version of 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
- AIX System Management Guides Position:1 Icon:books Operating System and Devices Index:BADMNEN Postion:1 Icon:bookshelf.gif
- 2 Operating System and Devices 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
- URL:/doc_link/en_US/a_doc_lib/aixbman/baseadmn/Ch1.htm#CE13340208vick 4 The System Administrator's Objectives Icon:paper.gif
- URL:/doc link/en US/a doc lib/aixbman/baseadmn/Ch2.htm 3 Chapter 2. Starting and Stopping the System Icon:chapter.gif
- URL:/doc_link/en_US/a_doc_lib/aixbman/baseadmn/starting.htm Starting the System 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 		
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		
	<pre></pre>		
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		
			

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
	

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

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 Documentation Library Service in AIX 5L Version 5.2 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

Complete the following steps to build an index:

- 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	С	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
	1000000			
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	5.1.0
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 be 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. To do change the ownership of the directory, type the following on a command line:

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
```

Where <code>index_name</code> is the 8 character name of the index. The values for <code>-t</code> and <code>-ccsid</code> depend on the language of the documents in the index. Note that all single-byte languages have a <code>-t</code> value of NORM. All multi-byte languages have a <code>-t</code> value of NGRAM and you also must add the <code>-ccsid</code> 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	6011
English (United States) ISO8859-15	NORM	923	6011
English Great Britain ISO8859-1	NORM	819	5997
Catalan ISO8859-1	NORM	819	3820
Catalan ISO8859-15	NORM	923	3820
Czech ISO8859-2	NORM	912	15840
French ISO8859-1	NORM	819	7011
French ISO8859-15	NORM	923	7011

Language	-t	-ccsid	-lang
French Canadian ISO8859-1	NORM	819	7011
German ISO8859-1	NORM	819	4841
German ISO8859-15	NORM	923	4841
German Switzerland ISO8859-1	NORM	819	4839
Hungarian ISO8859-2	NORM	912	8914
Icelandic ISO8859-1	NORM	819	9752
Italian ISO8859-1	NORM	819	9771
Italian ISO8859-15	NORM	923	9771
Norwegian ISO8859-1	NORM	819	14138
Polish ISO8859-2	NORM	912	15840
Portuguese, Brazil ISO8859-1	NORM	819	16072
Portuguese, Portugal ISO8859-1	NORM	819	16077
Portuguese, Portugal ISO8859-15	NORM	923	16077
Russian ISO8859-5	NORM	915	17919
Slovak ISO8859-2	NORM	912	18525
Spanish ISO8859-1	NORM	819	6156
Spanish ISO8859-15	NORM	923	6156
Swedish ISO8859-1	NORM	819	18835
Japanese IBM-932	NGRAM	93210564	JA_JP
Korean IBM-eucKR	NGRAM	949	11438
Traditional Chinese big5	NGRAM	950	4030
Simplified Chinese IBM-eucCN	NGRAM	1381	4029

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
```

b. Next you must specify the language and codeset of the documents that will be inserted into the index. The language specification command has the following format (type 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

⁻p /usr/docsearch/indexes/cal413en/data

⁻pw /usr/docsearch/indexes/cal413en/work

⁻lsse itelsswt

⁻t NORM

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 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 -1 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.

To start building the index, type the following command:

```
/usr/IMNSearch/bin/iteupdix -s server -x index name
```

To test that your documents were indexed successfully, type the following command:

```
/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

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):

```
/usr/IMNSearch/bin/itedomap
  -p /var/docsearch/indexes -c -x index_name
  -sp /doc_link/locale/
  -ti "index_title"
```

Note: There must be a final slash (/) after the locale.

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. The title should be written using the same language and codeset as the documents inside the index. The title should also be enclosed in quotation marks. 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 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

```
doc link -> /usr/share/man/info
```

Your web server will be able to serve the documentation with URLs, such as:

```
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 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. To do this, 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 on one command line:

```
/usr/IMNSearch/bin/itedomap
    -p /var/docsearch/indexes -c -x cal413en
    -sp /doc link/en US/
    -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

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 468). 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 468.

- 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

```
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 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
```

i. 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, type the following on one command line:

```
/usr/IMNSearch/bin/itedomap
    -p /var/docsearch/indexes -c -x index name
    -sp /doc link/locale/
    -ti index_title
```

Note: The following command must end with a slash (/) after the *locale*.

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. The title should be written using the same language and codeset as the documents inside the index. The title chould also be enclosed in quotation marks.

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/
-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 perform test searches of the index to ensure that 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 469
- 3. "Create an install package" on page 470

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 indexXXX 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 19, "Packaging Software for Installation," on page 415.

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 452.

Chapter 21. 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)
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

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
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. 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 (see "User Access Controls")
- · Version access (see "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 -I 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.

Formats and prints specified portions of an SCCS file to standard output prs

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.

Displays current SCCS file editing status. sact

Administration program for the SCCS system. The sccs command contains a set of SCCS

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.

Substitutes assigned values in place of identification keywords. VC Searches a system file for a pattern and displays text that follows it. what

Related Information

Commands Reference

The admin command, cdc command in AIX 5L Version 5.2 Commands Reference, Volume 1.

The delta command, get command in AIX 5L Version 5.2 Commands Reference, Volume 2.

The **prs** command in AIX 5L Version 5.2 Commands Reference, Volume 4.

The sccsdiff command in AIX 5L Version 5.2 Commands Reference, Volume 5.

Files References

The sccsfile file format.

Chapter 22. 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 *libname.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 libname.a, compile your program with the flag -lname (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 -I (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 -1bsd
```

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.

Note:

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.

32 Bit only	04 DIE Equivalent
wstrcat	wcscat
wstrchr	wcschr
wstrcmp	wcscoll
wstrcpy	wcscpy
wstrcspn	wcscspn
wstrdup	Not available and has no equivalents in the 64 bit libc.a

64 Rit Equivalent

32 Rit only

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.
- 3. Programming in a multi-threaded environment requires reentrant subroutines to ensure data integrity. See "List of Multi-threaded Programming Subroutines" on page 488 for more information.

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 a 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:

The difference between this value and 1.0 is this number:

0x0000000000000001, 0x000000000000000 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, 0x000000000000000 0.24651903288156618919116517665087070E-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:

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 493 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
 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?

isprintis character a printing character, including space?isgraphis 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 putc,putchar, fputc, putw Get a character or word from an input stream 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

Idopen, IdaopenOpen a common object fileIdclose, IdacloseClose a common object file

Idahread Reads the archive header of a member of an archive file

Idfhread Reads the file header of a common object file

Idlread, Idlinit, Idlitem Read and manipulate line number entries of a common object file

function

Idshread, IdnshreadRead a section header of a common object fileIdtbreadReads a symbol table entry of a common object file

Idgetname Retrieves a symbol name from a symbol table entry or from the

string table

Idlseek, Idnseek Seek to line number entries of a section of a common object file

Idohseek Seek to the optional file header of a common object file

Idrseek, Idnrseek Seek to the relocation information for a section of a common object

file

Idsseek, Idnsseek Seek to a section of a common object file

Idtbseek Seeks to the symbol table of a common object file

Idtbindex 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

loadbindProvides specific runtime resolution of a module's deferred symbolsloadqueryReturns error information from the load subroutine or the exec

subroutine. Also provides a list of object files loaded for the current

process

monitorStarts and stops execution profilingnlistGets entries from a name list

regcmp, regex Compile and matche regular-expression patterns

setjmp, longjmp Store a location

sgetI, sputIAccesses long numeric data in a machine-independent fashionsysconfDetermines 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 482
- "Manipulating File Systems" on page 483

Controlling Files

pclose

popen

putenv

rename

tmpfile

umask

pathconf, fpathconf

tmpnam or tempnam

truncate. ftruncate

utimes or utime

read, readx, readv, readvx

write, writex, writev, writevx

pipe

access, accessx, or faccessx Determine accessibility of a file

Clears space in a file fcntl, dup, or dup2 Control open file descriptors

fsvnc Writes changes in a file to permanent storage getenv Returns the value of an environment variable

getutent, getutid, getutline, putuline, 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

Iseek or IIseek 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

Closes an open pipe

Creates an interprocess channel Initiates a pipe to a process

Retrieve file implementation characteristics

Sets an environment variable Read from a file or device

Renames directory or file within a file system

Get file status

Creates a temporary file

Construct a name for a temporary file

Make a file shorter

Gets and sets the value of the file creation mask

Set file access or modification time

Write to a file or device

Working with Directories

statx, stat, fstatx, fstat, fullstat, fullstat

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 Updates all file systems information to disk sync Gets and sets the value of the file creation mask umask

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

Return the complex dot product of two vectors CDOTU. ZDOTU 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

SROTM, DROTM SROTMG, DROTMG Returns the dot product of two vectors plus a constant

Apply the modified Givens transformation 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

Perform rank 1 operation

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

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:

CGERC, ZGERC

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:

a64I. I64a

abs, div, labs, ldiv, imul_dbl, umul_dbl, llabs, lldiv Compute absolute value, division, and multiplication of

asin, asinl, acos, acosl, atan, atanl, atan2, atan2l

asinh, acosh, atanh

atof, atoff, strtod, strtold, strtof bessell: j0, j1, jn, y0, y1, yn class, finite, isnan, unordered

copysign, nextafter, scalb, logb, ilogb

nrand48, mrand48, jrand48, srand48, seed48, lcong48 Irand48_r, mrand48_r, nrand48_r, seed48_r, or

srand48_r

drem or remainder ecvt, fcvt, gcvt erf, erfl, erfc, erfcl

exp, expl, expm1, log, log1, log10, log10l, log1p, pow,

floor, floorl, ceil, ceill, nearest,

Convert between long integers and base-64 ASCII strings

integers

Compute inverse trigonometric functions Compute inverse hyperbolic functions

Convert an ASCII string to a floating point number

Compute bessel functions

Determine types of floating point functions Compute certain binary floating-point functions

Generate pseudo-random sequences Generate pseudo-random sequences

Compute an IEEE remainder

Convert a floating-point number to a string Compute error and complementary error functions Compute exponential, log, and power functions

trunc, rint, itrunc, uitrunc, fmod, fmodl, fabs, fabsl

fp_any_enable, fp_is_enabled, fp_enable_all,

fp_enable, fp_disable_all, fp_disable

fp_clr_flag, fp_set_flag, fp_read_flag, or fp_swap_flag

fp_invalid_op, fp_divbyzero, fp_overflow,

fp_underflow, fp_inexact, fp_any_xcp

fp_iop_snan, fp_iop_infsinf, fp_iop_infdinf,

fp iop zrdzr, fp iop infmzr, fp iop invcmp

fp_read_rnd,fp_swap_rnd

frexp, frexpl, Idexp, Idexpl, modf, modfl

l64a_r

Igamma, Igammal, gamma

hypot, cabs

13tol, Itol3

 $madd,\ msub,\ mult,\ mdiv,\ pow,\ gcd,\ invert,$

rpow, msqrt, mcmp, move, min, omin,

fmin, m_in, mout, omout, fmout, m_out, sdiv, itom

rand, srand rand_r

random, srandom, initstate, setstate

rsqrt

sin, cos, tan

sinh, sinhl, cosh, coshl, tanh, tanhl

sqrt, sqrtl, cbrt

strtol, strtoll, strtoul, strtoull, atol, atoi

Round floating-point numbers

Allow operations on the floating-point exception status

Allow operations on the floating-point exception status

Test to see if a floating-point exception has occurred

Test to see if a floating-point exception has occurred

Read and set the IEEE rounding mode

Manipulate floating point numbers

Manipulate floating point numbers

Converts base-64 long integers to strings Compute the logarithm of the gamma function

Compute Euclidean distance functions and absolute

values

Convert between 3-byte integers and long integers

Provide multiple precision integer arithmetic

Generate random numbers Generates random numbers Generate better random numbers

Computes the reciprocal of the square root of a number Compute trigonometric and inverse trigonometric functions

Computes hyperbolic functions

Compute square root and cube root functions

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

Ildiv Computes the quotient and remainder of the division of two long long integers

strtollConverts a string to a signed long long integerstrtoullConverts a string to an unsigned long long integer

wcstollConverts a wide character string to a signed long long integerwcstoullConverts 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.

acoslComputes the inverse cosine of a floating-point number in long double format

Computes the inverse sine of a floating-point number in long double format

atan2I 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

floorI 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

Idexpl Multiplies a floating-point number in long double format by an integral power of 2

Igammal 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 formatlogI 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

sinlComputes the sine of a floating-point number in long double formatsqrtlComputes the square root of a floating-point number in long double formatstrtoldConverts a string to a floating-point number in long double formattanlComputes 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, 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**.

The subroutines are listed in the following categories:

- · "Process Initiation"
- "Process Suspension" on page 487
- "Process Termination" on page 487
- "Process and Thread Identification" on page 487
- · "Process Accounting" on page 487
- "Process Resource Allocation" on page 487
- · "Process Prioritization" on page 487
- "Process and Thread Synchronization" on page 488
- "Process Signals and Masks" on page 488
- · "Process Messages" on page 488

Process Initiation

exec:, execl, execv, execle, execve, execlp, execvp, or exect

Execute new programs in the calling process

fork or vforkCreate a new processrebootRestarts the system

siginterrupt

Sets subroutines to restart when they are interrupted by

specific signals

Process Suspension

pauseSuspends a process until that process receives a signalwait, wait3, waitpidSuspend 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

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

getprocsGets process table entriesgetthrdsGets thread table entriessetpgid or setpgrpSet 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 accountingptrace Traces the execution of a process

Process Resource Allocation

brk or sbrkChange data segment space allocationgetdtablesizeGets 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
Locks processes, text, and data into memory

profil Starts and stops program address sampling for

execution profiling

ulimit Sets user process limits

Process Prioritization

getpriReturns 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 addUpdates a single word variable atomically

fetch and and fetch and or Sets or clears bits in a single word variable atomically

semctlControls semaphore operationssemgetGets a set of semaphoressemopPerforms 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

Sets signal stack context

Changes the set of blocked signals Implement a software signal facility

Process Messages

sigstack

sigsuspend

ssignal or gsignal

msgctlProvides message control operationsmsggetDisplays a message queue identifiermsgrcvReads 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 non-reentrant 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

The following lists the non-reentrant subroutines in libc.

asctime getgrent gsignal setkey

auditread	getgrgid	hcreate	setlogmask
closelog	getgrnam	hdestroy	setnetent
crypt	getgroupsbyuser	hsearch	setnetgrent
ctime	getgrset	inet_ntoa	setprotoent
dirname	gethostbyaddr	initstate	setpwent
drand48	gethostbyname	innetgr	setpwfile
ecvt	gethostent	iso_addr	setrpcent
endttyent	getlogin	iso_ntoa	setservent
encrypt	getnetbyaddr	jrand48	setstate
asctime	getnetbyname	l64a	setttyent
endfsent	getnetent	lcong48	setutent
endfsent	getnetgrent	link_ntoa	setutxent
endgrent	getopt	localtime	srand48
endhostent	getprotobyname	Irand48	srandom
endnetent	getprotobynumber	mrand48	ssignal
endnetgrent	getprotoent	mtime	strerror
endprotoent	getpwent	ndutent	strtok
endpwent	getpwnam	nrand48	syslog
endrpcent	getpwuid	ns_ntoa	ttyname
endservent	getrpcbyname	openlog	utmpname
endttyent	getrpcbynumber	pututline	wcstok
endutxent	getrpcent	pututxline	
erand48	getservbyname	rand	
ether_aton	getservbyport	random	
ether_ntoa	getservent	rcmd	
fcvt fgetgrent	getttyent	rcmd2	
fgetpwent	getttynam	readdir	
getdate	getuinfo	rexec	
getfsent	getutent	re_comp	
getfsent	getutid	re_exec	
getfsfile	getutline	seed48	
getfsfile	getutxent	setfsent	
getfsspec	getutxid	setgrent	
getfstype	getutxline	sethostent	

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) anystr (String1, String2) Determines whether String contains Character Determines the offset in String1 of the first character that also occurs in String2

balbrk (String, Open, Close, End)	Determines the offset in <i>String</i> 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
Caran (Camig)	in String
dname (p)	Determines which directory contains the file <i>p</i>
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 <i>pref</i> is an initial
In a list the ability of the country of the	substring of <i>String</i>
lockit (LockFile, Count, pid)	Creates a lock file
move (String1, String2, n)	Copies the first <i>n</i> characters of <i>String1</i> to <i>String2</i>
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 <i>Old</i>
Top: (ourng, old, rion)	in <i>String</i> with the character <i>New</i>
satoi (String, *ip)	Converts <i>String</i> to integer and saves it in * <i>ip</i>
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 <i>new</i>
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 (t)	Creates a pipe
xunlink (f)	Removes a directory entry
xwrite (fd, buffer, n)	Writes <i>n</i> bytes to the file associated with <i>fd</i> from
	buffer
zero (p, n)	Zeros <i>n</i> bytes starting at address <i>p</i>
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"
- "Auditing Subroutines"
- "Identification and Authentication Subroutines"
- "Process Subroutines" on page 492

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 I 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

getgrgid_r

or endpwent

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

Gets or sets the auditing mode of a system data object auditobj

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

Gets a group database entry for a group ID in a

multithreaded environment

Searches a group database for a name in a multithreaded getgrnam_r

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, Access the basic user information in the user database

Finds the value associated with a user getuinfo

getuserattr, IDtouser, nextuser, or putuserattr getuserpw, putuserpw, or putuserpwhist

loginfailed loginrestrictions loginsuccess newpass passwdexpired setpwdb or endpwdb setuserdb or enduserdb

system tcb Access the user information in the user database

Access the user authentication data Records an unsuccessful login attempt

Determines if a user is allowed to access the system

Records a successful login

Generates a new password for a user

Checks the user's password to determine if it has expired

Open or close the authentication database

Open or close the user database

Runs a shell command

Alters the Trusted Computing Base status of a file

Process Subroutines

getgid or getegid getgroups getpcred getpenv getuid or geteuid

getuid or geteuid initgroups

kleenup

setgid, setrgid, setegid, or setregid

setgroups setpcred setpenv

setuid, setruid, setuid, or setreuid

usrinfo

Get the real or group ID of the calling process Gets the concurrent group set of the current process Gets the current process security credentials Gets the current process environment

Get the real or effective user ID of the current process Initializes the supplementary group ID of the current

process

Cleans up the run-time environment of a process

Set the group IDs of the calling process

Sets the supplementary group ID of the current process

Sets the current process credentials Sets the current process environment

Set the process user IDs

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

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,
digent, alnument, entrlent, spaceent, totent, nonprntent, lineent, tabent;
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;
                         /*the function used by qsort and
int compar();
                         /*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*/
                        /*the library routine for quick sort*/
void qsort();
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 /lib/profiled/libbsd.a

/usr/ccs/lib/libcurses.a

/usr/ccs/lib/libc.a

/lib/profiled/libc.a

/usr/ccs/lib/libdbm.a

/usr/ccs/lib/libi18n.a

/usr/lib/liblvm.a

/usr/ccs/lib/libm.a

/usr/ccs/lib/libp/libm.a

/usr/lib/libodm.a /usr/lib/libPW.a

/usr/lib/libpthreads.a

/usr/lib/libqb.a

/usr/lib/librpcsvc.a

/usr/lib/librts.a

/usr/lib/libs.a

/usr/lib/libsm.a

/usr/lib/libsrc.a

/usr/lib/libmsaa.a

/usr/ccs/lib/libp/libmsaa.a

/usr/ccs/lib/libtermcap.a

/usr/lib/liby.a

/usr/lib/lib300.a /usr/lib/lib300s.a /usr/lib/lib300S.a

/usr/lib/lib4014.a

/usr/lib/lib450.a

/usr/lib/libcsys.a

/usr/ccs/lib/libdbx.a /usr/lib/libgsl.a

Berkeley library

Berkeley library profiled

Curses library

Standard I/O library, standard C library

Standard I/O library, standard C library profiled

Database Management library

Layout library

LVM (Logical Volume Manager) library

Math library

Math library profiled

ODM (Object Data Manager) library Programmers Workbench library POSIX compliant Threads library

Queue Backend library

RPC (Remote Procedure Calls) library

Run-Time Services library

Security functions

System management library

SRC (System Resource Controller) library SVID (System V Interface Definition) math library

SVID (System V Interface Definition) math library profiled

Terminal I/O

YP (Yellow Pages) library

Graphics subroutines for DASI 300 workstations Graphics subroutines for DASI 300s workstations Graphics subroutines for DASI 300S workstations Graphics subroutines for Tektronix 4014 workstations

Graphics subroutines for DASI 450 workstations Kernel extensions services Debug program library

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/libxqsl.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

Note: The information in this section is specific to AIX 5.1 earlier.

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 sort and itrunc Subroutines

The general-use version of the sart subroutine is in the libm.a library. The sart 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 xIc or xIf 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 -0 -o prog prog.c -lrs2 -lm
xlf -0 -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
```

Related Information

For further information on this topic, see the following:

- List of Time Data Manipulation Services in AIX 5L Version 5.2 System Management Guide: Operating System and Devices.
- Header Files Overview in AIX 5L Version 5.2 Files Reference.

Subroutine References

The **itrunc** subroutine,**printf** in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 1.

The **scanf** subroutine, **setlocale** subroutine **sqrt** subroutine in *AIX 5L Version 5.2 Technical Reference:* Base Operating System and Extensions Volume 2.

Chapter 23. 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:

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.

Default object repository for system information and can be used to store your /usr/lib/objrepos

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 (Esc+4)=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 that lists the valid choices. The associated commands are defined in the sm cmd opt.cmd to list field. The user gets this list by invoking the F4 (Esc + 4)=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.

Dialog screens can have the following options assigned to the entries:

Option	Function
#	Signifies that a numerical value is expected.
*	Signifies that an entry is mandatory.
+	Signifies that a listing of choices can be obtained using the F4 key.

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:

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
                = "top menu" *object's (menu screen) name
 id
                = "050"
 id_seq_num
 next_id
               = "commo"
                             *id of objects for next menu screen
               = "Communications Applications & Services"
 text
 text_msg_file = ""
 text_msg_set = 0
              = 0
 text_msg_id
 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 514 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:

```
*--- used for selector screens
sm name hdr:
 = "" = 0
 name msg file
 name_msg_id
                    = ""
 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 516 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:

```
*--- used for dialog screens
sm cmd hdr:
 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
name_msg_id
                         = 0
                         = 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 522 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
                  = "" *name of this object
 id
 id_seq_num
                   = "" *"0" if associated with selector screen
 disc_field_name
                   = ""
                    = ""
                          *text describing this entry
 name
                    = ""
 name msg file
                    = 0
 name msg set
                    = 0
 name msg id
                    = ""
 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
                    = 0
 values msg set
 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 518 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 <code>sm_cmd_opt.id</code> field values. SMIT allows up to five IDs (separated by commas) to be specified in a <code>sm_cmd_hdr.option_id</code> field, so that **sm_cmd_opt** objects with any of five different <code>sm_cmd_opt.id</code> 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.

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"

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.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 exect 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 I 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 (Is) command with a parameter that requests default values. SMIT uses the standard output of the show (Is) 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 (Is) command to get current values for a given instance such as a particular device. SMIT uses the standard output of the show (Is) command to fill in the values before displaying the dialog. The show (Is) 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 "-1 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 exect 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
а	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 I 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 503 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 501 and "Selector Screens" on page 502 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 -I 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 501.
- 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:

"Debugging SMIT Database Extensions" 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.
- 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 -I 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 two ways to retrieve SMIT help information:

- "Man Pages Method" on page 514
- "Message Catalog Method" on page 514.

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.

Note: An existing .msg file can 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 .msq 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.

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

next_id

	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. 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

The ID or name of the object. The value of id is a string with a maximum length of 64

next_id is a string with a maximum length of 64 characters.

The description of the task that is displayed as the menu item. The value of text is a string text with a maximum length of 1024 characters. This string can be formatted with embedded \n

(newline) characters.

The file name (not the full path name) that is the Message Facility catalog for the string, text_msg_file text. The value of text msq 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.

The Message Facility set ID for the string, text. Set IDs can be used to indicate subsets of text_msg_set a single catalog. The value of text_msg_set is an integer. Set to 0 if you are not using the

Message Facility.

The Message Facility ID for the string, text. The value of text_msg_id is an integer. Set to text_msg_id

0 if you are not using the Message Facility.

The type of the next object if this item is selected. Valid values are: next_type

> "m" Menu; the next object is a menu (sm_menu_opt).

 $^{\prime\prime}$ d $^{\prime\prime}$ 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.

Contains the string with the value of the name file contained in the file library indicated by help_msg_book

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.

Set to "" (empty string). id_seq_num

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). Set to "" (empty string). text_msg_file

Set to 0. text_msg_set text msg id Set to 0.

next_type The fast path screen type. The value of **next_type** is a string. Valid values are:

> Menu; the next id field specifies a menu screen fast path. $^{\prime\prime}$ d $^{\prime\prime}$ 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).

Set to "" (empty string). help_msg_id Set to "" (empty string). help_msg_loc Set to "" (empty string). help_msg_base 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 514, and "Message Catalog Method" on page 514 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.

Specifies the header object for the subsequent screen; set to the value next id

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

has_name_select

name

name msq file

name_msg_set

name_msg_id

type

ghost

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.

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.

 $^{\prime\prime}V^{\prime\prime}$ 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.

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.

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. 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.

The Message Facility ID for the string, name. The value of name_msg_id is an integer.

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 "i"

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.

- 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).

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.

Yes; display only the pop-up panel produced by the command "**v**" 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

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.
- $^{\prime\prime}$ d $^{\prime\prime}$ 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 514, and "Message Catalog Method" on page 514 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

id_seq_num

disc field name

name

name msq file

name_msg_set

name msg id

op_type

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.

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. 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.

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.

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.

The Message Facility set ID for the string, name. The value of name_msg_set is an integer. Set to 0 if not used.

The Message Facility message ID for the string, name. The value of **name_msg_id** is an integer. Set to 0 if not used.

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.
- "I" 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", "1", 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. 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.

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. 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.

entry_size

required

prefix

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. 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.

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.

Defines if the user can make multiple selections from a list of valid values produced by the command in the <code>cmd_to_list</code> 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.

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.

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.

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.

The Message Facility set ID for the values in the disp_values fields. Set to 0 if not used.

The Message Facility message ID for the values in the disp_values fields. Set to 0 if not used.

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.

cmd_to_list

cmd_to_list_postfix

multi_select

value_index

disp_values

values_msg_file

values_msg_set

values_msg_id

aix_values

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.

The file name sent as a parameter to the man command for retrieval of help help_msg_loc

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.

The fully qualified path name of a library that SMIT reads for the file names help_msg_base

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 514 and "Message Catalog Method" on page 514 located in "Creating SMIT Help Information for a New Task" on page 513.

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.

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.

"**v**" 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.

The text displayed as the title of the dialog screen. The value of **name** 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. 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.

The Message Facility ID for the string, name. Message IDs can be name_msg_id

created by the message extractor tools owned by the Message Facility.

The value of **name_msg_id** is an integer.

name_msg_set

option_id

cmd_to_exec

ask

exec mode

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.

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").

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 ${\bf cmd_to_exec}$ 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

cmd to discover

help_msg_id

help_msg_loc

help_msg_base

help_msg_book

cmd_to_discover_postfix

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.

"**v**" 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.

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.

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.

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.

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. The fully qualified path name of a library that SMIT reads for the file

names associated with the correct 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 514 and "Message Catalog Method" on page 514 located in "Creating SMIT Help Information for a New Task" on page 513.

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:
              = "apps"
  id
  id_seq_num = "010"
              = "demo"
  next id
              = "SMIT Demos"
  t.ext
  next type
              = "m"
sm menu opt:
  id
              = "demo"
              = "010"
  id seq num
```

```
= "n"
  next type
sm_menu_opt:
  id
                = "demo"
              = "020"
  id seq num
              = "demo mle inst lang hdr"
  next id
                 = "Demo 2: Add Language for Application Already Installed"
  text
  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"
  next_type
                  = "y"
  alias
sm_menu_opt:
                = "030"
  id seq num
                 = "demo"
  id
  next_id = "demo_lspv"
text = "Demo_3. List
                 = "Demo 3: List Contents of a Physical Volume"
  text
  text_msg_file = "smit.cat"
                  = "n"
  next_type
sm menu opt:
  id_seq_num
                = "040"
                = "demo"
  next_id = "demo_date"
text = "Demo 4: Change / Show Date, Time"
  text_msg_file = "smit.cat"
                  = "n"
  next_type
# 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
              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 \overline{to} classify is appended to the next id,
# since the type is "c", for cooked. So, the next_id will be
# either demo queuel 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:
```

= "demo_queue"

id

```
next id
                          = uc..._
= "demo_queue_uu....._
= "Add a Print Queue"
"-mit.cat"
                             = "demo queue"
                             = "demo queue dummy opt"
   option id
   name
   name_msg_file
                           = 52
   name_msg_set
   name_msg_id
                           = 41
                            = "c"
   type
                             = "y"
   ghost
                           = "\
   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"
                               = "n"
   next type
# 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"
                             = "mkpq"
   next id
                             = "demo_queue_dummy_opt"
   option_id
   has_name_select
                             = "y"
   ghost
                             = "n"
   next_type
                             = "c"
   type
                             = "\
   cmd to classify
gui_check()
   if [ SMIT = \mbox{"m}\mbox"]; then
     echo gui
   fi
}
   gui_check"
sm_name_hdr:
                   = "mkpqgui"
   id
                  = "invoke_gui"
   next id
                 = "d"
= "der
   next_type
                    = "demo_queue_dummy_opt"
   option_id
                    = "y"
   ghost
# Topics: 7
# Note: the exec_mode of this command is "e", which
# exits SMIT before running the cmd_to_exec.
#---
sm cmd hdr:
                    = "invoke_gui"
   id
                    = "/usr/bin/X11/xprintm"
   cmd_to_exec
                    = "e"
   exec mode
                    = "y"
   ghost
```

```
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 queuel 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"
                          = "demo_queue1"
  next id
                          = "demo queue opt"
  option id
                           = "Add a Print Queue"
  name
                          = "smit.cat"
  name msg file
                          = 52
  name msg set
                          = 41
  name_msg_id
                          = "v"
  ghost
                          = "\
  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 "
                                   = "n"
  next type
#----
# 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:
                           = "demo_queue_opt"
  id
                          = "0"
  id seg num
                          = ""
  disc_field_name
                          = "Add a Print Queue"
  name
                           = "smit.cat"
  name msg file
  name_msg_set
                           = 52
  name_msg_id
                           = 41
                           = "1"
  op type
  cmd_to_list
                           = "x()\
if [ $SMIT = \"a\" ] \n\
dspmsg -s 52 smit.cat 56 \
'Press Enter to automatically install the printer software.\n\
Press F3 to cancel.\n\
else \n\
dspmsg -s 52 smit.cat 57 'Click on this item to automatically install
```

```
the printer software.\n'\n\
fi\n\
} \n\
χ"
   #-----
# 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
        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.
#----
                      = "demo_mle_inst_lang_hdr"
= "demo_mle_inst_lang"
= "demo_mle_inst_lang_select"
= "Add Language for Application Already Installed"
= "smit.cat"
= 53
= 35
sm_name_hdr:
   id
    next id
    option id
    name
    name_msg_file
    name_msg_set
                         = 35
= "j"
    name_msg_id
    type
                            = "n"
    ghost
                       = "\
    cmd to_classify
        foo() {
            echo $1 | sed -n \"s/[^[]*\\[(\([^]]*\\).*/\\1/p\"
        foo"
    next_type
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"
                             = "smit.cat"
    name msg file
                              = 53
    name msg set
```

```
name_msg_id
                      = "]"
   op type
  entry_type
                     = "n"
                     = 0
   entry_size
  required
                     = ""
  # 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:
  = 35
= "j"
   name_msg_id
   type
                      = "n"
   ghost
                      = "\
   cmd_to_classify
      foo() {
         echo $1
      foo"
                   = "add_lang_input"
   cooked_field_name
                     = "d"
   next type
                     = "2850328"
   help msg id
sm_cmd opt:
  #----
# 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"
              = "demo_mle_add_app_lang"
   option id
```

```
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:1:S:X Option \"$@\"
      do
         case $Option in
            d) device=$OPTARG;;
            1) language=$OPTARG;;
            S) software=$OPTARG;;
            X) extend fs="-X";;
         esac
      done
       if [[ `/usr/lib/assist/check_cd -d $device` = '1' ]]
         /usr/lib/assist/mount cd $device
         CD MOUNTED=true
      fi
      if [[ $software = \"ALL\" ]]
         echo "Installing all software for $language..."
      else
         echo "Installing $software for $language..."
       fi
      exit $RC
      foo"
                             = "y"
   ask
                            = "n"
   ghost
                             = "2850325"
   help_msg_id
sm cmd opt:
                            = "demo_mle_add_app_lang"
   id
                            = "0"
   id seq num
                          = "add_lang_language"
   disc_field_name
                           = "LANGUAGE translation to install"
                            = "smit.cat"
   name msg file
                            = 53
   name msg set
                             = 20
   name msg id
                            = "n"
   entry_type
   entry_size
                            = 0
                            = "y"
   required
                            = "-1 "
   prefix
                            = "a"
   cmd_to_list_mode
                             = "2850328"
   help_msg_id
#----
# 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:
                       = "demo_mle_add_app_lang"
   id
                       = "020"
   id seg num
                       = "add_lang_input"
   disc field name
                        = "INPUT device/directory for software"
   name
   name_msg_file
                         = "smit.cat"
   name msg set
                         = 53
                         = 11
   name msg id
                         = "n"
   entry type
```

```
sm cmd opt:
                      = 0
   entry_size
   required
                      = "y"
   prefix
                       = "-5 "
                      = ""
   cmd_to_list_mode
cmd_to_list
                        = "\
      list messages ()
         language=$1
         device=$2
         lslpp -Lqc | cut -f2,3 -d':'
      list messages"
   \label{eq:cmd_to_list_postfix} \mbox{ = "add_lang_language add_lang_input"}
   multi_select = ","
value_index = 0
disp_values = "ALL"
   disp values
                      = "2850329"
   help_msg_id
#----
# 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} = \overline{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 = "1"
  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 = "1"
  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 = " ,-1,-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.
#-----
#----
# 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\")\} \setminus
  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\
   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 \overline{to} exec = "date proc () \
# MM dd hh mm ss yy\n\
                        # 3 4 5 6 7 2\n\
# dialogue param order
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 = "SE\overline{C}ONDS (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"
```

Related Information

For further information on this topic, see the following:

- · Chapter 14, "Object Data Manager (ODM)," on page 335
- System Management Interface Tool (SMIT) Overview in AIX 5L Version 5.2 System Management Guide: Operating System and Devices.

Commands References

The dspmsg command, gencat command in AIX 5L Version 5.2 Commands Reference, Volume 2.

The ksh command, man command in AIX 5L Version 5.2 Commands Reference, Volume 3.

The **odmadd** command, **odmcreate** command, **odmget** command in *AIX 5L Version 5.2 Commands Reference, Volume 4.*

The smit command in AIX 5L Version 5.2 Commands Reference, Volume 5.

Files References

The **ispaths** file.

Chapter 24. 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:

- · "SRC Objects" on page 538
- "SRC Communication Types" on page 542
- "Programming Subsystem Communication with the SRC" on page 545
- "Defining Your Subsystem to the SRC" on page 551
- "List of Additional SRC Subroutines" on page 552

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 5.2 System Management Guide: Operating System and Devices for a more detailed description of the characteristics of a subsystem.

A subserver 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 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. For more information on available subroutines, see "List of Additional SRC Subroutines" on page 552.

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, Issrc -I, 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 545 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 539
- "Subserver Type Object Class" on page 541
- "Notify Object Class" on page 541

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.

Table 5. Subsystem Object Descriptors and Default Values

Descriptors	Default Values	Flags
Subsystem name		-S
Path to subsystem command		-р
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	-0
stderr	/dev/console	-e
Communication type	Sockets	-K -I -S
Subsystem message type		-m
Communication IPC queue key		-1
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.

Specifies the full path name for the program executed by the subsystem start Subsystem command path

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 gueue 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. This field is required.

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.

Communication type

Note: Catastrophic errors are sent to the error log. 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. Indicates whether the status of an inoperative subsystem can be displayed on

Issrc -a or **Issrc -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 551 for information on defining and modifying subsystem objects.

Subserver Type Object Class

Display value

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).

Specifies the name of the subsystem that owns the subserver object. This field Owning subsystem name

is defined as a link to the SRC subsystem object class.

Specifies a decimal number that identifies the subserver. The code point is Code point

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 551 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.

Refer to the following sections to learn more about SRC communication types:

- "Signals Communication" on page 543
- "Sockets Communication" on page 544
- "IPC Message Queue Communication" on page 544

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 Adds a subsystem definition to the SRC configuration database mkssys

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 5.2 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 Is 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 545 for more information.

srcrrqs Saves the destination address of your subsystem's response to a received packet. (Also see

threadsafe version **srcrrqs_r**)

Sends your subsystem response packet to a request that your subsystem received. srcsrpy

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 5.2 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 542 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 sigset, sighold, sigrelse, or sigignore subroutine Specifies the action to take upon the delivery of a signal. Enhances the signal facility and provides signal management for application processes.

See "Signals Communication" on page 543 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 5.2 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 5.2 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 **srcrras** 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 547 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 gueue subsystems to receive SRC reguest 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 **srcreg** structure. This structure should be used any time an SRC packet is received.
- When the subsystem has been started, use the **msqqet** 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 5.2 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 gueue. 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;
short action;
   short object;
                         /*object to act on*/
                        /*action START, STOP, STATUS, TRACE,\
  short parm1;
                      /*reserved for variables*/
/*reserved for
   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 538 and "Defining Your Subsystem to the SRC" on page 551 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. Allows a subsystem to send a reply to a request. srcsrpy

See "Responding to Trace Requests" on page 550 and "Responding to Refresh Requests" on page 551 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 548 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 550 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.)	Х	Х	
SRCACT	Started and active.	Х	X	
SRCINAC	Not active.			
SRCINOP	Inoperative.	Х	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.		Х	
SRCSTPG	Stopping.	Х	X	
SRCTST	TEST active.			
SRCTSTPN	TEST pending.			

The SRC Issrc 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. This field must be a NULL terminated string.
- 5. Fill in the objname field with the name of the subsystem or subserver for which the objtext field applies. This field must be a NULL terminated string.

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:

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 a svrreply structure with the rtncode field 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 539 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. Changes a subsystem definition in the SRC configuration database. chssys Changes a subserver definition in the SRC configuration database. chserver 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:

Adds a subsystem definition to the SRC configuration database addssys Changes a subsystem definition in the SRC configuration database chssys

Initializes a new subsystem definition with default values defssys

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 Gets a subserver definition from the SRC configuration database getsubsvr

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 **getsubsyr** 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:

Returns message text for SRC errors encountered by SRC library routines. src_err_msg

(Also see threadsafe version src_err_msg_r)

srcsbuf Requests status from the subsystem in printable format.

(Also see threadsafe version **srcsbuf_r**)

Sends a message or request to the subsystem. srcsrqt

(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.

Gets the text representation for an SRC status code. srcstattxt

(Also see threadsafe version **srcstattxt_r**)

srcstop Requests termination of the subsystem. srcstrt Requests the start of a subsystem.

Related Information

For further information on this topic, see the following:

- Chapter 14, "Object Data Manager (ODM)," on page 335
- Chapter 24, "System Resource Controller," on page 537
- TCP/IP Protocols in AIX 5L Version 5.2 System Management Guide: Communications and Networks.
- System Resource Controller Overview in AIX 5L Version 5.2 System Management Guide: Operating System and Devices
- SRC Request Structure Example in AIX 5L Version 5.2 Files Reference.
- Sockets Overview, Reading Internet Datagrams Example Program, Understanding the Network Systems (NS) Protocol Family, and Checking for Pending Connections Example Program in AIX 5L Version 5.2 Communications Programming Concepts
- Message Queue Kernel Services in AIX 5L Version 5.2 Kernel Extensions and Device Support Programming Concepts.

Subroutine References

The addssys subroutine, defssys subroutine, getsockname subroutine, msgget subroutine, msgrcv or msgxrcv subroutine in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 1

The recvfrom subroutine, select subroutine, signal subroutine, socket subroutine, srcsbuf subroutine, srcrrqs subroutine, srcsrpy subroutine, srcstat subroutine, srcstathdr subroutine in AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2.

Commands References

The mknotify command, mkserver command, mkssys command in AIX 5L Version 5.2 Commands Reference, Volume 3

The refresh command, rmnotify command, srcmstr daemon, in AIX 5L Version 5.2 Commands Reference, Volume 4.

tracesoff command, traceson command,

Files References

The spc.h, srcobj.h file.

Chapter 25. 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.

For more information on the trace facility, refer to the following:

- "The Trace Facility Overview"
- · "Controlling the Trace"
- · "Recording Trace Event Data" on page 556
- "Generating a Trace Report" on page 557
- "Extracting trace data from a dump" on page 557
- "Trace Facility Commands" on page 557
- "Trace Facility Calls and Subroutines" on page 558
- "Trace Facility Files" on page 559
- "Trace Event Data" on page 559
- "Trace Facility Generic Trace Channels" on page 559

The Trace Facility Overview

The trace facility is in the **bos.sysmgt.trace** file set. To see if this file set is installed, type the following on the command line:

1slpp -1 | 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 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 -I 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 (-i) or exclude (-k).

To display the program names associated with trace hooks, certain hooks must be enabled. These are specified using the tidhk trace event group. 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 tidhk -j 254

Tracing occurs. To stop tracing, type the following on a command line:

trcstop trcrpt -0 exec=on

The **-O exec=on** trcrpt option shows the program names, see the **trcrpt** command for more information.

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.

Tracing can 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 25, "Trace Facility," on page 555.

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:

trace

trcdead /0 /tmp/tlog /var/adm/ras/vmcore.0

creates a trace log file /tmp/tlog which may then be formatted with the following: trcrpt /tmp/tlog

Trace Facility Commands

The following commands are part of the trace facility:

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 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. trenm 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 uses the kernel name list file to interpret addresses when formatting a report from a trace log file. Note: It is recommended that you use the -n trace option instead of trcnm. This puts name list information into the

trace log file instead of a separate file, and includes

symbols from kernel extentions.

trcrpt

trcstop trcupdate 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 <code>/etc/trcfmt</code> file to determine how to interpret the data recorded for each event.

Stops the tracing of system events.

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

utrchook, utrchook64

trcoff

trcon

trcstart

Records trace events of more than five words of data. The **trcgen** subroutine can 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). 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. Use **trcgenk** and **trcgenkt** in the kernel. C programmers should always use the **TRCGEN** and **TRCGENK** macros.

Records trace events of up to five words of data. These subroutines can be used to record an event as part of the system event trace (trace channel 0). Kernel programmers can use **trchook** and **trchook64**. C programmers should always use the **TRCHKL0** - **TRCHKL5** and **TRCHKL0T** - **TRCHKL5T** macros.

If you are not using these macros, you need to build your own trace hook word. The format is documented with the /etc/trcfmt file. Note that the 32-bit and 64-bit traces have different hook word formats.

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.

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.

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.

Frees and deactivates a generic trace channel.

trcstop

Trace Facility Files

/etc/trcfmt

/var/adm/ras/trcfile

/usr/include/sys/trchkid.h /usr/include/sys/trcmacros.h

Contains the trace formatting templates used by the **trcrpt** command to determine how to interpret the data recorded for each event.

Contains the default trace log file. The **trace** command allows you to specify a different trace log file.

Contains trace hook identifier definitions.

Contains commonly used macros for recording trace events.

Trace Event Data

See the /etc/trcfmt file for the format of the trace event data.

Trace Hook Identifiers

A trace hook identifier is a three-digit hexadecimal number that identifies an event being traced. However, when you use tracing macros such as **TRCHKL1**, you specify the trace hook as: hhh00000

where hhh is the hook id.

Most 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.

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** or **TRCGENT** macros, or if necessary, **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, most of 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.

Start the Trace Facility

Use the following procedures to configure and start a system trace:

- · "Configuring the trace Command"
- · "Recording Trace Event Data"
- · "Using Generic Trace Channels"
- · "Starting a Trace"
- "Stopping a Trace" on page 561
- "Generating a Trace Report" on page 561

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. For information on the syntax of this command, seetrace Daemon.

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. However, when tracing using the tracing macros such as TRCHKL1, you specify the trace hook as hhh00000, where hhh is the hook id. 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 -i 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.

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 trestart subroutine to activate a trace channel and to determine the channel number. The subsystem modules can then record trace events TRCGEN or TRCGENT macros, or the trcgen, trcgent, trcgenk, or trcgenkt subroutines. 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 trestart subroutine. The subsystem must provide the user interface for activating and deactivating subsystem tracing.

Starting a Trace

Use 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
```

Related Information

Subroutine References

The **trcgen** subroutine, **trchook** subroutine, **trcoff** subroutine, **trcston** subroutine, **trcston** subroutine in *AIX 5L Version 5.2 Technical Reference: Base Operating System and Extensions Volume 2.*

Commands References

The trace daemon, trcdead command, trcnm command, trcrpt command, trcstop command, trcupdate command in AIX 5L Version 5.2 Commands Reference, Volume 5.

Chapter 26. 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

This overview provides information on following topics:

- · "TTY Subsystem Objectives"
- · "tty Subsystem Modules"
- "TTY Subsystem Structure" on page 564
- "Common Services" on page 565
- "Synchronization" on page 567

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 5.2 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" on page 564
- · "Line Disciplines" on page 564
- · "Converter Modules" on page 564

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:

cxma128-port asynchronous PCI controller.cxpa8-port asynchronous PCI controller.

Ift Low-function terminal. The tty name is $\frac{\text{dev/Ift }Y}{\text{dev}}$, where $Y \ge 0$.

pty pseudo-terminal device driver.

sf Universal asynchronous receiver/transceivers (UARTs) on system planar.

The section, "TTY Drivers" on page 571, 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:

Idterm Terminal devices (see "Line Discipline Module (Idterm)" on page 567)

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)

Ic_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 **Idterm** line discipline.

"Converter Modules" on page 570 provides more information on converters.

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 (**Ic_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" on page 566:
 - **cfgetispeed** subroutine
 - cfgetospeed subroutine
 - cfsetispeed subroutine
 - cfsetospeed subroutine
 - tcsendbreak subroutine
- "Flow Control Services" on page 566:
 - tcdrain subroutine
 - tcflow subroutine
 - tcflush subroutine
- "Terminal Information and Control" on page 566:
 - isatty subroutine
 - tcgetattr subroutine
 - tcsetattr subroutine
 - ttylock subroutine
 - ttylocked subroutine
 - ttyname subroutine
 - ttyunlock subroutine
 - ttywait subroutine
- "Window and Terminal Size Services" on page 566:
 - termdef subroutine
 - TIOCGWINSZ ioctl operation
 - TIOCSWINSZ ioctl operation
- "Process Group Management Services" on page 566:
 - 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

Gets terminal state tcgetattr tcsetattr Sets terminal state

ttylock, ttywait, ttyunlock, or ttylocked Controls tty locking functions ttvname 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

Indicates the horizontal size (in pixels) of the window or terminal ws xpixel 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 joctl 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 8192 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 8192 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 (Idterm)

The Idterm 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 interface in new applications.

This section describes the features provided by the **Idterm** line discipline. For more information about controlling Idterm, see "termios.h File" in AIX 5L Version 5.2 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 1flag 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

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 1flag 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

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 1flag 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 1flag field of termios structure), input bytes are not assembled into lines, and erase and kill processing does not occur.

Represents the minimum number of bytes that should be received when the read request is successful MIN 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. MIN and MAX have values from 0 to 265. The four possible combinations 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 can lead to performance issues if overused; 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 modern status lines. If the **CLOCAL** flag is clear, the modern 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:

- · The nls module
- · The uc siis and lc siis 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 5.2 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 siis and lc siis 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 Ic_siis lower converter is pushed onto the stream below the line discipline. The uc siis and lc siis 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 5L Version 5.2 Files Reference.

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 PCI adapter card.
- The cxpa driver supports the 8-port PCI adapter card.
- The **sf** driver supports the native ports on the system planar.

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

For further information on this topic, see the following:

- Chapter 7, "Input and Output Handling," on page 171
- STREAMS Overview in AIX 5L Version 5.2 Communications Programming Concepts.

Subroutine References

The **setcsmap** subroutine.

Commands References

The rlogind daemon in AIX 5L Version 5.2 Commands Reference, Volume 4.

The setmaps command, stty command in AIX 5L Version 5.2 Commands Reference, Volume 5.

The xdm daemonAIX 5L Version 5.2 Commands Reference, Volume 6.

Files References

The eucioctl.h file, Ift file, Ip file, pty file, setmaps file, termios.h file, termiox.h file, tty file.

Chapter 27. 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

Uses loader domain to limit search

Adds shared libraries to 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.

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.

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/00domain 1 is a regular file, then setting the LIBPATH environment variable to the string

```
/etc/loader domain/00domain 1:/lib:/usr/lib
```

causes processes to create and use the loader domain associated with the /etc/loader domain/00domain 1 file.

 If /etc/loader domain/00domain 1 is a regular file, then the ldom program is built with the following command:

```
cc -o ldom ldom.c -L/etc/loader domain/00domain 1
```

The path name /etc/loader domain/00domain 1 is inserted as the first entry in the LIBPATH information of the loader section for the 1dom file. When 1dom is executed, it creates and uses the loader domain associated with the /etc/loader domain/00domain 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.

Appendix A. High-Resolution Time Measurements Using POWER-based Time Base or POWER family Real-Time Clock

Note: The information in this section is specific to AIX 5.1 earlier.

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

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.

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.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Dept. LRAS/Bldg. 003
11400 Burnet Road
Austin, TX 78758-3498
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation Licensing 2-31 Roppongi 3-chome, Minato-ku Tokyo 106, Japan

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

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.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

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.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

(c) (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. (c) Copyright IBM Corp. _enter the year or years_. All rights reserved.

Trademarks

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

- AIX
- AIX 5L
- eServer
- IBM
- NetView
- pSeries

Java and all Java-based trademarks and logos are registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, MS-DOS, Windows, and Windows NT 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 trademarks or service marks of others.

Index

Special characters	adb debug program (continued)
_exit subroutine 247	external variables
_LARGE_FILES 134	displaying 46 files
_system_configuration.max_ncpus variable	
dynamic logical partitioning 352	locating values in 44 writing 45
_system_configuration.ncpus field	i-node dumps
dynamic logical partitioning 352	example 54
_system_configuration.ncpus variable	instructions
dynamic logical partitioning 354	displaying 41
_system_configuration.original_ncpus variable	integers
dynamic logical partitioning 352	using in expressions 35
	list of operators 49
Numarias	list of subcommands 49
Numerics	list of variables 52
216840 105	maps
41Map203831 48	memory, changing 43
	maps, address
A	displaying 47
A	maximum offsets
access subroutine 153	setting 39
adb debug program	memory
address maps	changing 45
displaying 47	memory maps
addresses	changing 43
displaying 42	numbers, computing 40
finding current 46	operators
forming 41	using in expressions 36
binary files	operators, list of 49
patching 44	output widths
breakpoints 33, 34	setting 39
C stack backtrace	program execution
displaying 42	controlling 31
commands, combining 37	programs
computing numbers 40	continuing execution 34
creating scripts 37	preparing for debugging 31
customizing 37	running 32
data	single-stepping 34
displaying 41 data formats	stopping 35
choosing 43	stopping with keys 35
data formatting, example 56	prompts, using 31
default input formats	sample programs 52 scripts, creating 37
setting 40	shell commands, using 31
directory dumps	source files
example 54	displaying and manipulating 41
displaying text 40	starting 29, 30, 31
examples	stopping a program 35
data formatting 56	subcommands, list of 49
directory dumps 54	symbols
i-node dumps 54	using in expressions 35
tracing multiple functions 58	text, displaying 40
exiting 31	tracing multiple functions, example of 58
expressions	using prompts 31
list of 48	using shell commands 31
using integers 35	values
using operators 36	locating in files 44
using symbols 35	Č

adb debug program (continued)	chdir subroutine 121
variables	checkacquire command
displaying external 46	dynamic logical partitioning 357, 360
list of 52	checkrelease command
using 45	dynamic logical partitioning 357, 360
addresses	chmod subroutine 154
disk 127	chown subroutine 154
program	chroot subroutine 121
overview 377	cleanup handlers 204
alarm subroutine 243	close subroutine
alerts 113	closing files with 146
aliases	commands 113
SMIT 507	backup 113
allocation	bindprocessor 355, 356
compressed file system 127	chclass 355, 356
JFS 126	checkacquire 357, 360
allocation bitmaps 131	checkrelease 357, 360
allocation groups 131	cron 112
allocation policy 392	diag 99
default 392	drmgr, flags 357
async-cancel safety 201	drmgr, options 357
attributes object 196, 206, 212	errclear 112, 114
• , , ,	errdemon 114
	errlogger 114
В	errmsg 113
back end program 156	errpt 99, 108, 111, 114
backup command 113	errstop 112
binding a thread 182	errupdate 108, 114
bindprocessor command	ipcs 356
dynamic logical partitioning 355, 356	kill 356
bindprocessor system call	ls 113
bind CPU ID 354	Isclass 355, 356
blocks	Isrset 355, 356
	mycmd 560
boot 131	postacquire 357, 361
data 131	postrelease 357, 360
full logical 126	preacquire 357, 361
indirect 128, 148	prerelease 357, 360
logical 126	ps 355, 356
partial logical 126	register 356, 359
super 131	SCCS
boot block 131	list of 476
buffer size operations 567	scriptinfo 356, 359
	trace 555, 560
C	trcrpt 560, 561
	trostop 561
cancelability 200	undopreacquire 357, 361
cancelation 200, 201	undoprerelease 357, 360
capacity upgrade on demand (CUoD) 351	usage 356, 359
cbreak mode 19	wlmctrl 355
characters	compiling a multi-threaded program 261
adding characters to windows 11	compressed file system 127
adding to windows 14	condition variable
allowing 8-bit character return 17	attributes
converting control characters to printables 15	creation 212
current	destruction 212
curses 5	
deleting 16	object 212
getting from standard input 17	process-shared 250 creation 213
returning if no input is available 17	definition 212
chclass command	
dynamic logical partitioning 356	destruction 213, 215

condition variable (continued)	curses (continued)
signalling 215	refreshing
usage 216	pads 9
wait 216	windows 9
waiting 214	screen 5
contention scope 231	scrolling
global 231	windows 15
local 231	setting options 26
process 231	starting 6
system 231	stopping 6
contention-scope attribute 231	strings
control characters	adding to windows 14
converting to printables 15 controlling terminal 563	sub-windows 8, 9 creating 8
converters 564	termcap
creat subroutine 142	converting to terminfo 23
creating files with 145	terminology 5
creating	windows 5, 7, 8
destroying 236	clearing 16
keys 236	copying 10
creating a thread 196	creating 8
creation and destruction 207, 213	deleting 8
cron command 112	drawing boxes around 10
CUoD 351	moving 10
curses	overlapping 10
adding characters to windows 11, 14	refreshing 9
characters	screens 5
current 5	scrolling 15
returning if no input is available 17	cursors
control characters 15	controlling placement after refresh 11
converting control characters to printables 15	getting location of logical cursor 11
converting termcap to terminfo 23	logical
current character 5	curses 5
current line 5	moving logical curses 11
CUISOIS	moving physical 11 physical
controlling placement after refresh 11 logical 5	curses 5
moving the logical cursor 11	Cuises 5
moving the logical cursor 11	
physical 5	D
deleting characters 16	data blocks 126, 131
displays 5	data types
getting characters from standard input 17	pthread_once_t 235
inserting	dbx
blank lines in windows 15	debugging multi-threaded program using 264
lines	symbolic debug program 61
current 5	dbx command
deleting 16	print subcommand 71
logical cursor 5	step subcommand 75
macros 5	stop subcommand
moving	thbp and thp aliases 66
logical cursor 11	thread subcommand 71
physical cursor 11	dbx debug program
naming conventions 5	files
pads 5	current, displaying 64
creating 8	dbx plug-in
deleting 8	alias
refreshing 9	callback routine 92
physical cursor 5	event types 81
printing formatted printf on windows 15	example 93
formatted printf on windows 15	

dbx plug-in (continued) fds	dbx symbolic debug program (continued)
	files (continued)
callback routine 83	current, changing 64
file format 78	reading dbx subcommands from 78
file names 79	source, displaying 64
get_pthread_context	folding variables to lowercase and uppercase 72
callback routine 88	handling signals 68
get_thread_context	list of subcommands 94
callback routine 87	machine level debugging 74
header file 80	machine level programs
Interface 80	debugging 74
loading 79	running 75
locate_symbol	machine registers 74
callback routine 90	memory addresses 75
location 79	modifiers
modules	for expressions 71
callback routine 84	multiple processes 67
naming 78	multiple threads 65
overview 78	names, scoping 71
print	new dbx prompt
callback routine 92	defining 76
process	operators
callback routine 83	for expressions 71
pthreads	print output
callback routine 87	changing variables 73
read_memory	procedures
callback routine 90	calling 70
regions	current, changing 64
callback routine 85	program control 62
session	· · ·
	programs
callback routine 82	controlling 62
set_pthread_context	machine level 74
callback routine 89	machine level, running 75
set_thread_context	multiple processes 67
callback routine 88	multiple threads 65
threads	running 62
callback routine 86	separating output from dbx 63
unloading 79	prompts
version control 80	defining 76
what_function	reading dbx subcommands from a file 78
callback routine 91	running programs 62
write_memory	running shell commands 61
callback routine 90	scoping names 71
dbx plug-in callback routines 82	separating dbx output from program output 63
dbx symbolic debug program	signals, handling 68
.dbxinit file 77	source directory path
aliases	changing 64
dbx subcommand, creating 77	source files
breakpoints 62	displaying and manipulating 64, 68, 76
calling procedures 70	stack trace, displaying 70
changing the current file 64	starting 61
command line editing 62	subcommands, list of 94
current file	thread-related objects 65
displaying 64	tracing execution 63
dbx subcommand aliases	type checking in expressions 72
creating 77	using 61
expressions	variables
modifiers and operators for 71	changing print output 73
type checking 72	displaying and modifying 70
files	folding, lowercase and uppercase 72
dbxinit 77	deadlock 210
JUDANIA / /	UCALIUUN LIV

deamons	DR_LOG_WARNING environment variable
srcmstr 538	dynamic logical partitioning 359
trace 555	DR_MEM_ADD_CHECK event 361, 363
debugging multi-threaded programs 264	DR_MEM_ADD_POST event 362, 363
dbx, using 264	DR_MEM_ADD_POST_ERROR event 362, 363
kernel debug program, using 264	DR_MEM_ADD_PRE event 361, 363
descriptors 142	DR_MEM_REMOVE_CHECK event 362, 363
destructor routine 237	DR_MEM_REMOVE_POST event 362, 363
detached state 225	DR_MEM_REMOVE_POST_ERROR event 362, 363
detachstate attribute 197	DR_MEM_REMOVE_PRE event 362, 363
device driver packaging	DR_MEM_SIZE_COMPLETED environment variable
naming considerations 418	dynamic logical partitioning 358
diag command 99	DR_MEM_SIZE_REQUEST environment variable
dialogs	dynamic logical partitioning 358
SMIT 503	DR_PINNABLE_FRAMES environment variable
directories	dynamic logical partitioning 358
changing	dr_reconfig
current 121	dynamic logical partitioning 354
root 121	DR_resource_POST_ERROR 363
linking 141	DR_TOTAL_FRAMES environment variable
overview 119	dynamic logical partitioning 359
status 141	Draft 7 applications
working with	porting 263
overview 120	dri_cpu structure 363
subroutines for 121	dri_mem structure 363
disk address format 127	drmgr command
disk fragments 131	dynamic logical partitioning 357
disk i-nodes 123, 132	flags 357
disk space allocation 126	options 357
documentation search service 451	scripts, information about 358
documents 451	scripts, installing 357
searching 451	scripts, uninstalling 357
DR_BCPUID environment variable	dynamic logical partitioning 351
dynamic logical partitioning 359	_system_configuration.max_ncpus variable 352
DR_CPU_ADD_CHECK event 361, 362	_system_configuration.ncpus field 352
DR_CPU_ADD_POST event 361, 362	_system_configuration.ncpus variable 354
DR_CPU_ADD_POST_ERROR event 361, 362	_system_configuration.original_ncpus variable 352
DR_CPU_ADD_PRE event 361, 362	add operations 356
DR_CPU_REMOVE_CHECK event 361, 362	application binding 355
DR_CPU_REMOVE_POST event 361, 363	application scripts 356
DR_CPU_REMOVE_POST_ERROR event 361, 363	bindprocessor command 355, 356
DR_CPU_REMOVE_PRE event 361, 363	bindprocessor system call 354
DR_DETAIL_LEVEL environment variable	chclass command 356
dynamic logical partitioning 359	check phase 356, 363
DR_ERROR environment variable	checkacquire command 357
dynamic logical partitioning 358	checkrelease command 357
DR_FAIL	commands 356
DR_SUCCESS 363	dependencies 354
DR_FREE_FRAMES environment variable	DR_BCPUID environment variable 359
dynamic logical partitioning 358	DR_DETAIL_LEVEL environment variable 359
DR_LCPUID environment variable	DR_ERROR environment variable 358
dynamic logical partitioning 359	DR_FAIL 363, 364
DR_LOG_DEBUG environment variable	DR_FREE_FRAMES environment variable 358
dynamic logical partitioning 359	DR_LCPUID environment variable 359
DR_LOG_EMERG environment variable	DR_LOG_DEBUG environment variable 359
dynamic logical partitioning 359	DR_LOG_EMERG environment variable 359
DR_LOG_ERR environment variable	DR_LOG_ERR environment variable 359
dynamic logical partitioning 359	DR_LOG_INFO environment variable 359
DR_LOG_INFO environment variable	DR_LOG_WARNING environment variable 359
dynamic logical partitioning 359	DR_MEM_SIZE_COMPLETED environment
	variable 358

dynamic logical partitioning (continued)	dynamic logical partitioning (continued)
DR_MEM_SIZE_REQUEST environment	script commands (continued)
variable 358	scriptinfo 359
DR_PINNABLE_FRAMES environment	undopreacquire 361
variable 358	undopreacquire 360
	•
dr_reconfig system call 354	usage 359
DR_resource_POST_ERROR 363	scriptinfo command 356
DR_SUCCESS 363, 364	scripts, execution 358
DR_TOTAL_FRAMES environment variable 359	scripts, installing 357
DR_WAIT 364	scripts, naming 358
drmgr command 357	scripts, using 355
flags 357	sigaction system call 354
dynamic logical partitioning-aware 352	signal delivery 353
Dynamic Processor Deallocation and 355	signals, using 356
environment variable	SIGRECONFIG signal 353, 354, 355
memory add and remove 358	troubleshooting 352, 354
processor add and remove 359	memory removal failure 354
handlers, extended 364	PCI slot removal failure 354
input parameters 358	processor removal failure 354
integrating events 353	undopreacquire command 357
ipcs command 356	undoprerelease command 357
kernel extensions 361	usage command 356
kill command 356	var.v_ncpus field 352
large pages 353	var.v_ncpus_cfg variable 352
Isclass command 356	Workload Manager and 355
Isrset command 356	Workload Manager dependencies 355
operations 355	dynamic logical partitioning-aware 351
check phase 356	APIs, using 353
post phase 356	dynamic logical partitioning-friendly 351
pre phase 356	dynamic logical partitioning-safe 351
scripts, using 355	Dynamic Processor Deallocation 183, 351
post phase 356, 363	•
post-add phase 364	_
post-error phase 363	E
post-remove phase 363	ECHO directive 287
postacquire command 357	electronic license agreement 426
postrelease command 357	entry point routine 198
pre phase 356, 363	environment variables
pre-add phase 363	DR_BCPUID 359
pre-remove phase 363	DR_DETAIL_LEVEL 359
preacquire command 357	DR_ERROR 358
prerelease command 357	DR_FREE_FRAMES 358
processor bindings 354	DR_LCPUID 359
processor numbering 354	DR_LOG_DEBUG 359
program types 353	DR_LOG_EMERG 359
ps command 355, 356	DR_LOG_ERR 359
reconfig_handler_complete 364	DR_LOG_INFO 359
reconfig_handler_complete kernel service 364	DR_LOG_WARNING 359
reconfiguration handlers 362	DR_MEM_SIZE_COMPLETED 358
register command 356	DR MEM SIZE REQUEST 358
registering kernel extension handlers 361	DR_PINNABLE_FRAMES 358
remove operations 356	DR_TOTAL_FRAMES 359
requests, displaying 359	LANG 358
script commands	
checkacquire 360	memory add and remove 358
checkrelease 360	NUM_SPAREVP 263
postacquire 361	PATH 358
postacquire 361 postrelease 360	processor add and remove 359
preacquire 361	errclear command 112
prerelease 360	errmsg command 113
register 359	error log transferring to another system 100

error log descriptors 99	F
error logging	-
adding messages 113	fast paths SMIT 507
alerts 113	fclear subroutine 147
cleaning an error log 112	fdpr
commands 114	debugging reordered executables 76
copying an error log 113	FIFO (first-in, first-out)
example report 108, 111	understanding 151
files 114	file descriptor tables
generating a report 112	definition 142
kernel services 114	file descriptors
managing 100 overview 99	definition 142
reading an error report 106	duplicating
stopping an error log 112	dup subroutine 143
subroutines 114	fcntl subroutine 143
error logging facility 99	fork subroutine 143
error report	managing 143
detailed example 108	preset values 144
generating 112	resource limit 145
summary example 111	file systems 131
errpt command 99, 108, 111, 114	bitmap 127
errstop command 112, 114	compressed 127 controlling 159
errupdate command 108, 114	fragment map 127
events	fragmented 126
DR_CPU_ADD_CHECK 361, 362	layout 131
DR_CPU_ADD_POST 361, 362	overview 117
DR_CPU_ADD_POST_ERROR 361, 362	quotas 129
DR_CPU_ADD_PRE 361, 362	subroutines 159
DR_CPU_REMOVE_CHECK 361, 362	types
DR_CPU_REMOVE_POST_ERROR361_363	creating 154
DR_CPU_REMOVE_POST_ERROR 361, 363 DR_CPU_REMOVE_PRE 361, 363	file types
DR_MEM_ADD_CHECK 361, 363	overview 117
DR_MEM_ADD_POST 362, 363	file-system helpers
DR_MEM_ADD_POST_ERROR 362, 363	execution syntax 156
DR_MEM_ADD_PRE 361, 363	operations 155
DR_MEM_REMOVE_CHECK 362, 363	files 117, 142
DR_MEM_REMOVE_POST 362, 363	access modes 153
DR_MEM_REMOVE_POST_ERROR 362, 363	allocating space to 126
DR_MEM_REMOVE_PRE 362, 363	closing 146
example	creating 145
dbx plug-in 93	input and output (I/O) 146
example programs 493	large _LARGE_FILES 134
manipulating characters	64-bit file system 136
isalnum (ctype) routine 493	allocation in file systems 127
isalpha (ctype) routine 493	common pitfalls 137
isascii (ctype) routine 493	implications for existing programs 133
iscntrl (ctype) routine 493	open protection 134
isdigit (ctype) routine 493	porting applications 134
islower (ctype) routine 493 ispunct (ctype) routine 493	writing programs that access 133
isspace (ctype) routine 493	linking 140
isupper (ctype) routine 493	locking fields 124
exec subroutine 246	masks 145
extended regular expressions	opening 146
lex command 281	overview 117
extension naming conventions, fileset 418	pipes 151
,	reading 147
	SCCS
	controlling 474 creating 472
	CIEAUIU 4/2

files (continued)	I/O offset
SCCS (continued)	absolute 147
detecting damage 475	and read subroutine 147
editing 472	and write subroutine 148
repairing damage 475	description 146
tracking 474	end_relative 147
updating 472	manipulating 147
sharing open 143	relative 147
status 153	in-core i-nodes 124
termcap 21	index nodes 123
terminfo 21	indexes for documents 451
truncating 149	indirect blocks 128
working with	inheritance protocol 233
subroutines for 119	inheritsched attribute 228
writing 148	init command
fileset	SRC 537
extension naming conventions 418	initializations 234
naming conventions 417	input modes 19
fileset revision levels 419	•
	IPC (interprocess channel) 118
first-in first-out scheduling policy 229	ipcs command
floating-point exceptions 162, 163	dynamic logical partitioning 356
disabled and enabled processing 162	
subroutines 161	1
fork cleanup handlers 246	J
fork subroutine 246	JFS
fragment map 127	disk space allocation 126
fragmented file system 126	JFS2
fragments	i-nodes 124
disk 131	
map 127	
front end program 155, 156	K
ftruncate subroutine 149	kernel extensions
full logical block 126	dynamic logical partitioning 361
fullstat subroutine 153	reconfiguration handlers 362
	registering handlers 361
G	kernel programming
generic trace channels 560	multiprocessor issues 188
generic trace charmers 500	kernel services
	reconfig_handler_complete 364
H	kernel thread 193
	key destruction 237
Hardware Management Console (HMC) 351	keys
heap	creating 236
32-bit applications 391	destroying 236
64-bit applications 392	destruction 237
help	kill command
SMIT 513	dynamic logical partitioning 356
helpers 155, 156	kill subroutine 243
hlpadb 49	
•	_
_	L
	LAF (license agreement files) 426
i-nodes 123, 124	large files
definition 117	common pitfalls 137
disk 132	arithmetic overflows 138
i-number byte offset 119	
JFS2 124	file size limits 140
modifying 123	fseek/ftell 138
timestamp	imbedded file offsets 139
changing 153	improper data types 137
	parameter mismatches 137

large files (continued)	Isrset command
common pitfalls (continued)	dynamic logical partitioning 356
string conversions 139	
open protection 134	
porting applications to 134	M
using _LARGE_FILES 134	m4
using 64-bit file system 136	built-in macros 328
writing programs that access 133	changing quote characters 327
large program support 175	checking for defined macros 328
lazy loading 373	conditional expressions 331
lex command 279	creating user-defined macros 325
compiling the lexical analyzer 280	integer arithmetic 329
defining substitution strings 285	macro processing with arguments 327
extended regular expressions 281	manipulating files 329
lexical analyzer 286	manipulating strings 331
matched strings 286	predefined macros 328
passing code to program 284	printing names and definitions 333
start conditions 290	quote characters 326
lex Command	redirecting output 330
end-of-file processing 289	removing macro definitions 328
input/output subroutines 288	system programs 330
lex library 285	unique names 330
lexical analyzer 279	using the macro processor 325
operators 281	macros
lex Library 285	
lexical analyzer 279	PTHREAD_ONCE_INIT 235 main Subroutine 285
parser program 280	
libpthreads_compat.a library 196	malloc subsystem 391 alloca 391
libpthreads.a library 194, 196	
libraries	allocation policy 392
hardware-specific subroutines 499	allocation policy, default 392
library model test 231	calloc 391
librs2.a library 499	free 391
license agreement files (LAF) 426	heap 391, 392
license agreement information 425, 426	mallinfo 391
license file 426	malloc 391
license file placement 426	mallopt 391
line disciplines 564	realloc 391
linking	valloc 391
runtime 371	masks 145
links 141	memory
directory 141	adding and removing 358
hard 140	system
symbolic 141	introduction 377
loader domains	memory management 379
	allocating memory 391
creating and deleting 575 using 573	listing of memory manipulation services 388
locking	program address space
	overview 377
creating user locking services 187	memory mapping
logical block 126	listing of memory mapping services 388
logical partitioning (LPAR) 351	mmap comparison with shmat 380
logical volume manager	mmap compatibility considerations 381
library subroutines 158	overview 379
long locks 239	semaphore subroutines overview 382
longjmp subroutine 244	menus
LPAR 351	SMIT 501
Is command 113	minicurses
Isclass command	characters
dynamic logical partitioning 356	adding characters to windows 11
Iseek subroutine 147	strings
	adding to windows 14

mkfifo subroutine 146	ODM (Object Data Manager) (continued)
mknod subroutine	object classes (continued)
creating regular files with 145	unlocking 338
creating special files with 146	objects
mount command 156	adding to object classes 337
mount helpers	definition 335
execution syntax 157	searching for 342
overview 156	storing 338
multi-threaded program	predicates 342
Compiler Invocation 262	comparison operators 343
compiling 261	constants in 344
debugging 264	descriptor names 343
multiprocessor programming 274	logical operator 344
mutex	offset 147
attributes	one-time initialization 234
creation 206	open subroutine 142
destruction 206	creating files with 145
object 206	opening a file with 146
prioceiling 233	operating system
process-shared 250	libraries 498
creation 207	
definition 206	options (threads library) 247 overviews
destruction 207	make command
locking 209	creating description files 309
protocol attributes 233	creating target files 319
protocols 233	defining and using macros 315
unlocking 209	internal rules 312
usage 209	using with non-SCCS files 321
mycmd command 560	using with SCCS files 320
mycpu system call	
dynamic logical partitioning 352	D
dynamic logical partitioning 352	P
	P packaging
dynamic logical partitioning 352	- packaging
N	•
N notify object class (SRC)	packaging device driver packaging 418 file names 419
N notify object class (SRC) creating a subsystem notification method 541	packaging device driver packaging 418 file names 419 fileset extension naming conventions 418
N notify object class (SRC) creating a subsystem notification method 541 removing a subsystem notification method 541	packaging device driver packaging 418 file names 419 fileset extension naming conventions 418 license agreement information 426
N notify object class (SRC) creating a subsystem notification method 541	packaging device driver packaging 418 file names 419 fileset extension naming conventions 418 license agreement information 426 message catalogs 418
N notify object class (SRC) creating a subsystem notification method 541 removing a subsystem notification method 541	packaging device driver packaging 418 file names 419 fileset extension naming conventions 418 license agreement information 426 message catalogs 418 naming conventions 417
N notify object class (SRC) creating a subsystem notification method 541 removing a subsystem notification method 541 NUM_SPAREVP environment variable 263	packaging device driver packaging 418 file names 419 fileset extension naming conventions 418 license agreement information 426 message catalogs 418 naming conventions 417 partitioning requirements 416
N notify object class (SRC) creating a subsystem notification method 541 removing a subsystem notification method 541 NUM_SPAREVP environment variable 263 O	packaging device driver packaging 418 file names 419 fileset extension naming conventions 418 license agreement information 426 message catalogs 418 naming conventions 417 partitioning requirements 416 requisite information 422
N notify object class (SRC) creating a subsystem notification method 541 removing a subsystem notification method 541 NUM_SPAREVP environment variable 263 O O_DEFER 147	packaging device driver packaging 418 file names 419 fileset extension naming conventions 418 license agreement information 426 message catalogs 418 naming conventions 417 partitioning requirements 416 requisite information 422 pads
N notify object class (SRC) creating a subsystem notification method 541 removing a subsystem notification method 541 NUM_SPAREVP environment variable 263 O O_DEFER 147 object classes 335	packaging device driver packaging 418 file names 419 fileset extension naming conventions 418 license agreement information 426 message catalogs 418 naming conventions 417 partitioning requirements 416 requisite information 422 pads creating 8
N notify object class (SRC) creating a subsystem notification method 541 removing a subsystem notification method 541 NUM_SPAREVP environment variable 263 O O_DEFER 147 object classes 335 SMIT 504	packaging device driver packaging 418 file names 419 fileset extension naming conventions 418 license agreement information 426 message catalogs 418 naming conventions 417 partitioning requirements 416 requisite information 422 pads creating 8 curses 5
N notify object class (SRC) creating a subsystem notification method 541 removing a subsystem notification method 541 NUM_SPAREVP environment variable 263 O O_DEFER 147 object classes 335 SMIT 504 objects 335	packaging device driver packaging 418 file names 419 fileset extension naming conventions 418 license agreement information 426 message catalogs 418 naming conventions 417 partitioning requirements 416 requisite information 422 pads creating 8 curses 5 deleting 8
N notify object class (SRC) creating a subsystem notification method 541 removing a subsystem notification method 541 NUM_SPAREVP environment variable 263 O O_DEFER 147 object classes 335 SMIT 504 objects 335 ODM (Object Data Manager)	packaging device driver packaging 418 file names 419 fileset extension naming conventions 418 license agreement information 426 message catalogs 418 naming conventions 417 partitioning requirements 416 requisite information 422 pads creating 8 curses 5 deleting 8 refreshing 9
N notify object class (SRC) creating a subsystem notification method 541 removing a subsystem notification method 541 NUM_SPAREVP environment variable 263 O O_DEFER 147 object classes 335 SMIT 504 objects 335 ODM (Object Data Manager) descriptors 339	packaging device driver packaging 418 file names 419 fileset extension naming conventions 418 license agreement information 426 message catalogs 418 naming conventions 417 partitioning requirements 416 requisite information 422 pads creating 8 curses 5 deleting 8 refreshing 9 parser
N notify object class (SRC) creating a subsystem notification method 541 removing a subsystem notification method 541 NUM_SPAREVP environment variable 263 O O_DEFER 147 object classes 335 SMIT 504 objects 335 ODM (Object Data Manager) descriptors 339 link 339	packaging device driver packaging 418 file names 419 fileset extension naming conventions 418 license agreement information 426 message catalogs 418 naming conventions 417 partitioning requirements 416 requisite information 422 pads creating 8 curses 5 deleting 8 refreshing 9 parser lexical analyzer 280
N notify object class (SRC) creating a subsystem notification method 541 removing a subsystem notification method 541 NUM_SPAREVP environment variable 263 O O_DEFER 147 object classes 335 SMIT 504 objects 335 ODM (Object Data Manager) descriptors 339 link 339 method 341	packaging device driver packaging 418 file names 419 fileset extension naming conventions 418 license agreement information 426 message catalogs 418 naming conventions 417 partitioning requirements 416 requisite information 422 pads creating 8 curses 5 deleting 8 refreshing 9 parser lexical analyzer 280 partial logical block 126
N notify object class (SRC) creating a subsystem notification method 541 removing a subsystem notification method 541 NUM_SPAREVP environment variable 263 O O_DEFER 147 object classes 335 SMIT 504 objects 335 ODM (Object Data Manager) descriptors 339 link 339	packaging device driver packaging 418 file names 419 fileset extension naming conventions 418 license agreement information 426 message catalogs 418 naming conventions 417 partitioning requirements 416 requisite information 422 pads creating 8 curses 5 deleting 8 refreshing 9 parser lexical analyzer 280 partial logical block 126 pbsearchsort 495
N notify object class (SRC) creating a subsystem notification method 541 removing a subsystem notification method 541 NUM_SPAREVP environment variable 263 O O_DEFER 147 object classes 335 SMIT 504 objects 335 ODM (Object Data Manager) descriptors 339 link 339 method 341 terminal 339 example code 346	packaging device driver packaging 418 file names 419 fileset extension naming conventions 418 license agreement information 426 message catalogs 418 naming conventions 417 partitioning requirements 416 requisite information 422 pads creating 8 curses 5 deleting 8 refreshing 9 parser lexical analyzer 280 partial logical block 126 pbsearchsort 495 pclose subroutine 151
N notify object class (SRC) creating a subsystem notification method 541 removing a subsystem notification method 541 NUM_SPAREVP environment variable 263 O O_DEFER 147 object classes 335 SMIT 504 objects 335 ODM (Object Data Manager) descriptors 339 link 339 method 341 terminal 339 example code 346 adding objects 348	packaging device driver packaging 418 file names 419 fileset extension naming conventions 418 license agreement information 426 message catalogs 418 naming conventions 417 partitioning requirements 416 requisite information 422 pads creating 8 curses 5 deleting 8 refreshing 9 parser lexical analyzer 280 partial logical block 126 pbsearchsort 495 pclose subroutine 151 permissions
N notify object class (SRC) creating a subsystem notification method 541 removing a subsystem notification method 541 NUM_SPAREVP environment variable 263 O O_DEFER 147 object classes 335 SMIT 504 objects 335 ODM (Object Data Manager) descriptors 339 link 339 method 341 terminal 339 example code 346 adding objects 348 creating object classes 346	packaging device driver packaging 418 file names 419 fileset extension naming conventions 418 license agreement information 426 message catalogs 418 naming conventions 417 partitioning requirements 416 requisite information 422 pads creating 8 curses 5 deleting 8 refreshing 9 parser lexical analyzer 280 partial logical block 126 pbsearchsort 495 pclose subroutine 151 permissions directories 153
N notify object class (SRC) creating a subsystem notification method 541 removing a subsystem notification method 541 NUM_SPAREVP environment variable 263 O O_DEFER 147 object classes 335 SMIT 504 objects 335 ODM (Object Data Manager) descriptors 339 link 339 method 341 terminal 339 example code 346 adding objects 348	packaging device driver packaging 418 file names 419 fileset extension naming conventions 418 license agreement information 426 message catalogs 418 naming conventions 417 partitioning requirements 416 requisite information 422 pads creating 8 curses 5 deleting 8 refreshing 9 parser lexical analyzer 280 partial logical block 126 pbsearchsort 495 pclose subroutine 151 permissions directories 153 files 153
N notify object class (SRC) creating a subsystem notification method 541 removing a subsystem notification method 541 NUM_SPAREVP environment variable 263 O O_DEFER 147 object classes 335 SMIT 504 objects 335 ODM (Object Data Manager) descriptors 339 link 339 method 341 terminal 339 example code 346 adding objects 348 creating object classes 346	packaging device driver packaging 418 file names 419 fileset extension naming conventions 418 license agreement information 426 message catalogs 418 naming conventions 417 partitioning requirements 416 requisite information 422 pads creating 8 curses 5 deleting 8 refreshing 9 parser lexical analyzer 280 partial logical block 126 pbsearchsort 495 pclose subroutine 151 permissions directories 153 files 153 pipe subroutine 151
N notify object class (SRC) creating a subsystem notification method 541 removing a subsystem notification method 541 NUM_SPAREVP environment variable 263 O O_DEFER 147 object classes 335 SMIT 504 objects 335 ODM (Object Data Manager) descriptors 339 link 339 method 341 terminal 339 example code 346 adding objects 348 creating object classes 346 list of commands 345	packaging device driver packaging 418 file names 419 fileset extension naming conventions 418 license agreement information 426 message catalogs 418 naming conventions 417 partitioning requirements 416 requisite information 422 pads creating 8 curses 5 deleting 8 refreshing 9 parser lexical analyzer 280 partial logical block 126 pbsearchsort 495 pclose subroutine 151 permissions directories 153 files 153 pipe subroutine 151 pipes
N notify object class (SRC) creating a subsystem notification method 541 removing a subsystem notification method 541 NUM_SPAREVP environment variable 263 O O_DEFER 147 object classes 335 SMIT 504 objects 335 ODM (Object Data Manager) descriptors 339 link 339 method 341 terminal 339 example code 346 adding objects 348 creating object classes 346 list of commands 345 list of subroutines 345	packaging device driver packaging 418 file names 419 fileset extension naming conventions 418 license agreement information 426 message catalogs 418 naming conventions 417 partitioning requirements 416 requisite information 422 pads creating 8 curses 5 deleting 8 refreshing 9 parser lexical analyzer 280 partial logical block 126 pbsearchsort 495 pclose subroutine 151 permissions directories 153 files 153 pipe subroutine 151 pipes child processes 152
N notify object class (SRC) creating a subsystem notification method 541 removing a subsystem notification method 541 NUM_SPAREVP environment variable 263 O O_DEFER 147 object classes 335 SMIT 504 objects 335 ODM (Object Data Manager) descriptors 339 link 339 method 341 terminal 339 example code 346 adding objects 348 creating object classes 346 list of commands 345 list of subroutines 345 object classes	packaging device driver packaging 418 file names 419 fileset extension naming conventions 418 license agreement information 426 message catalogs 418 naming conventions 417 partitioning requirements 416 requisite information 422 pads creating 8 curses 5 deleting 8 refreshing 9 parser lexical analyzer 280 partial logical block 126 pbsearchsort 495 pclose subroutine 151 permissions directories 153 files 153 pipe subroutine 151 pipes child processes 152 creating with mkfifo 146
N notify object class (SRC) creating a subsystem notification method 541 removing a subsystem notification method 541 NUM_SPAREVP environment variable 263 O O_DEFER 147 object classes 335 SMIT 504 objects 335 ODM (Object Data Manager) descriptors 339 link 339 method 341 terminal 339 example code 346 adding objects 348 creating object classes 346 list of commands 345 list of subroutines 345 object classes adding objects 337	packaging device driver packaging 418 file names 419 fileset extension naming conventions 418 license agreement information 426 message catalogs 418 naming conventions 417 partitioning requirements 416 requisite information 422 pads creating 8 curses 5 deleting 8 refreshing 9 parser lexical analyzer 280 partial logical block 126 pbsearchsort 495 pclose subroutine 151 permissions directories 153 files 153 pipe subroutine 151 pipes child processes 152
N notify object class (SRC) creating a subsystem notification method 541 removing a subsystem notification method 541 NUM_SPAREVP environment variable 263 O O_DEFER 147 object classes 335 SMIT 504 objects 335 ODM (Object Data Manager) descriptors 339 link 339 method 341 terminal 339 example code 346 adding objects 348 creating object classes 346 list of commands 345 list of subroutines 345 object classes adding objects 337 creating 337	packaging device driver packaging 418 file names 419 fileset extension naming conventions 418 license agreement information 426 message catalogs 418 naming conventions 417 partitioning requirements 416 requisite information 422 pads creating 8 curses 5 deleting 8 refreshing 9 parser lexical analyzer 280 partial logical block 126 pbsearchsort 495 pclose subroutine 151 permissions directories 153 files 153 pipe subroutine 151 pipes child processes 152 creating with mkfifo 146

plug-in, dbx <i>(continued)</i> event types 81	priority inversion 232 priority scheduling POSIX option 248
example 93	process
fds	properties 191
callback routine 83	process-shared attribute 250
file format 78	processes
file names 79	using pipes 151
get_pthread_context	processor
callback routine 88	example configurations 180
get_thread_context	numbers 179
callback routine 87	ODM names 179
header file 80	states 179
Interface 80	processors
loading 79	adding and removing 359
locate_symbol	protection protocol 233
callback routine 90	protocol attribute 233
location 79	protocols
modules	inheritance 233
callback routine 84	mutex 233
naming 78	priority inheritance 233
print	priority protection 233
callback routine 92	protection 233
process	ps command
callback routine 83	dynamic logical partitioning 355, 356
pthreads	pthread 196
callback routine 87	pthread_atfork subroutine 246
read_memory	pthread_attr_destroy subroutine 196
callback routine 90	pthread_attr_getdetachstate subroutine 197
regions	pthread_attr_getinheitsched subroutine 228
callback routine 85	pthread_attr_getsatckaddr subroutine 248
session	pthread_attr_getschedparam 230
callback routine 82	pthread_attr_getschedparam attribute 229
set_pthread_context	pthread_attr_getschedpolicy attribute 229
callback routine 89	pthread_attr_getscope subroutine 231
set_thread_context	pthread_attr_getstacksize subroutine 248
callback routine 88	pthread_attr_init subroutine 196
threads	pthread_attr_setdetachstate subroutine 197
callback routine 86	pthread_attr_setinheritsched subroutine 228
unloading 79	pthread_attr_setsatckaddr subroutine 248
version control 80	pthread_attr_setschedparam 230
what_function	pthread_attr_setschedparam subroutine 229
callback routine 91	pthread_attr_setschedpolicy subroutine 229
write_memory	pthread_attr_setscope subroutine 231
callback routine 90	pthread_attr_setstacksize subroutine 248
plug-in, dbx callback routines 82	pthread_attr_t data type 196
plug-in, dbx	pthread_cancel subroutine 200
overview 78	pthread_cleanup_pop subroutine 204
popen subroutine 151	pthread_cleanup_push subroutine 204
POSIX thread 196	pthread_cond_broadcast subroutine 215
postacquire command	pthread_cond_destroy subroutine 213
dynamic logical partitioning 357, 361	pthread_cond_init subroutine 213
postrelease command	PTHREAD_COND_INITIALIZER macro 213
dynamic logical partitioning 357, 360	pthread_cond_signal subroutine 215
preacquire command	pthread_cond_t data type 213
dynamic logical partitioning 357, 361	pthread_cond_timedwait subroutine 214
prerelease command	pthread_cond_wait subroutine 214
dynamic logical partitioning 357, 360	pthread_condattr_destroy subroutine 212
prioceiling attribute 233	pthread_condattr_init subroutine 212
priority	pthread_condattr_t data type 212
inheritance protocol 233	pthread_create subroutine 198
protection protocol 233	pthread equal subroutine 198

pthread_exit subroutine 199	reconfig_handler_complete subroutine (continued)
pthread_getschedparam 230	DR_MEM_REMOVE_CHECK event 363
pthread_getschedparam subroutine 230	DR_MEM_REMOVE_POST event 363
pthread_getspecific subroutine 238	DR_MEM_REMOVE_POST_ERROR event 363
pthread_join subroutine 225	DR_MEM_REMOVE_PRE event 363
pthread_key_create subroutine 236	event parameter 362
pthread_key_delete subroutine 237	h_arg parameter 362
pthread_key_t data type 235	req parameter 362
pthread_kill subroutine 243	resource_info parameter 363
pthread_mutex_destroy subroutine 207	reconfig_register subroutine 361
pthread_mutex_init subroutine 207	DR_CPU_ADD_CHECK event 361
pthread_mutex_lock subroutine 209	DR_CPU_ADD_POST event 361
pthread_mutex_t data type 207	DR_CPU_ADD_POST_ERROR event 361
pthread_mutex_trylock subroutine 209	DR_CPU_ADD_PRE event 361
pthread_mutex_unlock subroutine 209	DR_CPU_REMOVE_CHECK event 361
pthread_mutexattr_destroy subroutine 206	DR_CPU_REMOVE_POST event 361
pthread_mutexattr_getprioceiling subroutine 233	DR_CPU_REMOVE_POST_ERROR event 361
pthread_mutexattr_getprotocol subroutine 233	DR_CPU_REMOVE_PRE event 361
pthread_mutexattr_init subroutine 206	DR_MEM_ADD_CHECK event 361
pthread_mutexattr_setprioceiling subroutine 233	DR_MEM_ADD_POST event 362
pthread_mutexattr_setprotocol subroutine 233	DR_MEM_ADD_POST_ERROR event 362
pthread_mutexattr_t data type 206	DR_MEM_ADD_PRE event 361
t to the second of the second	
pthread_once subroutine 235	DR_MEM_REMOVE_CHECK event 362
PTHREAD_ONCE_INIT macro 235	DR_MEM_REMOVE_POST_ERROR avert 200
pthread_once_t data type 235	DR_MEM_REMOVE_POST_ERROR event 362
PTHREAD_PRIO_INHERIT 233	DR_MEM_REMOVE_PRE event 362
PTHREAD_PRIO_NONE 233	h_arg parameter 362
PTHREAD_PRIO_PROTECT 233	h_token parameter 362
pthread_self subroutine 198	name parameter 362
pthread_setcancelstate subroutine 200	reconfig_register function 362
pthread_setcanceltype subroutine 200	reconfig_unregister function 362
pthread_setschedparam subroutine 231	reconfiguration handlers
pthread_setspecific subroutine 238	dynamic logical partitioning 362
pthread_t data type 198	register command
pthread_testcancel 201	dynamic logical partitioning 356, 359
pthread_yield subroutine 215	regular expressions
	lex command 281
	REJECT directive 287
Q	remove subroutine 141
quotas 129	rmdir subroutine 141
400.00	round-robin scheduling policy 229
	routines
R	destructor 237
	reconfig_handler_complete 364
race condition 211	runtime linking 371
raise subroutine 243	Turning 67 1
read subroutine 147	
reconfig_handler_complete kernel service 364	S
reconfig_handler_complete routine 364	_
reconfig_handler_complete subroutine	SCCS
DR_CPU_ADD_CHECK event 362	commands
DR_CPU_ADD_POST event 362	list of 476
DR_CPU_ADD_POST_ERROR event 362	files
DR_CPU_ADD_PRE event 362	controlling 474
DR_CPU_REMOVE_CHECK event 362	creating 472
DR_CPU_REMOVE_POST event 363	detecting damage 475
DR_CPU_REMOVE_POST_ERROR event 363	editing 472
DR_CPU_REMOVE_PRE event 363	repairing damage 475
DR_MEM_ADD_CHECK event 363	tracking 474
	updating 472
DR_MEM_ADD_POST_EDDOR event 262	flags and parameters 472
DR_MEM_ADD_POST_ERROR event 363	sched_yield 230
DR MEM ADD PRE event 363	Sorica_yield 200

schedpolary attribute 229 scheduling parameters 228 policies 229 prority 229 synchronization of 232 screen types surchronization of 232 screen types SMIT 501 scriptino command descriptions or 356, 359 searching and sorting example program 495 sed command starting the editor 365 using string replacement 370 using text in commands 369 using the command summary 366 selectors SMIT 502 searphores (from OSF/1) 240 setimp subroutine 244 shared libraries creating 375 lazy loading 373 shared memory mnap comparison with shmat overview 379 shared objects 371 creating 373 ship licinase file 426 signaction subroutine 244 shared libraries or 370 using text included and summary 366 selectors specified 426 signaction dynamic logical partitioning 354 signalisms 243 signals 243 signals 360 short subroutine 244 shared libraries or 379 using text included and shared libraries or 370 using text include	sched_yield subroutine 230	SMIT (System Management Interface Tool) (continued)
scheduling parameters 228 priority 229 creating 513 understanding 513 information command descriptors comd to		
scheduling parameters 228 policies 229 priority 229 synchronization of 232 core places 259 synchronization of 232 core places 250 sore places		·
parameters 228 policies 229 priority 229 compositive 229 priority 229 priority 229 compositive 229 priority 229 compositive 232 compositive 232 compositive 232 compositive 232 compositive 232 compositive 233 compositive 234 compositive 234 producers 243 signals compositive 244 signal delivery 245 generation 243 handlers 243 signals Signace 244 signal composition 244 signal compositive 243 signals Signace 243 signals Signace 243 signals Signace 243 signals si		
policies 229 synchronization of 232 synchronization of 232 synchronization of 232 synchronization of 232 serent hypes SMIT 501 scriptific command dynamic logical partitioning 356, 359 searching and sorting example program 495 sed command starting the editor 365 using string replacement 370 using text in commands 369 using the command summary 366 selectors SMIT 502 semaphores (from OSF/1) 240 setiling subroutine 244 setiling subroutine 244 shardel libraries creating 375 lazy loading 373 shared objects 371 creating 373 ship license file 426 signation subroutine 243 signal delivery 245 generation 243 handlers 243, 244 maske 243 signals SIGRECONFIG 353, 354 dynamic logical partitioning 354 signer object definitions 551 removing subsystems to the SRC object class 551 int command 570 list of subroutines 552 modifying subsystems to the SRC object definitions 551 removing subsystems to the SRC object definitions 551 removing subsystem object definitions 551 sperations 538 removing subsystem object definitions 551 removing subsystem object	-	
priority 229 synchronization of 232 screen types SMIT 501 scriptinfo command dynamic logical partitioning 356, 359 searching and sorting example program 495 sed command starting the editor 365 using string replacement 370 using text in commands 369 using the command summary 366 selectors SMIT 502 semaphores (from OSF/1) 240 setting subroutine 244 shared libraries creating 375 lazy loading 373 shared memory mmap companison with shmat 380 overview 379 shared objects 371 creating 373 shared objects 371 creating 373 sipplicense file 26 sigaction dynamic logical partitioning 354 signal delivery 245 generation 243 handlers 243, 244 masks 243 signal delivery 245 generation 243 Signales SIGRECONFIG dynamic logical partitioning 354 siggred subroutine 244 signal delivery 245 generation 243 Signales SIGRECONFIG dynamic logical partitioning 354 siggred subroutine 243 signales 507 designing 503 executing 511 generating 510 example program 524 fast paths 507 generating commands with 510 menus designing 503 executing 511 generating 504 name selects designing 502 object classes dialog 518 dialog 5		•
synchronization of 232 screen types SMIT 501 scriptinfo command dynamic logical partitioning 356, 359 searching and sorting example program 495 sed command starting the editor 365 using string replacement 370 using text in commands 369 using the command summary 366 selectors SMIT 502 smaphores (from OSF/1) 240 selipms ubroutine 244 shared libraries creating 375 lazy loading 373 shared memory mmap comparison with shnat 380 overview 379 shared objects 371 creating 373 ship license file 426 sigaction dynamic logical partitioning 354 sigaction subroutine 243 signali delivery 245 generation 243 handlers 243, 244 masks 243 signal delivery 245 generation 243 handlers 243, 244 masks 243 signal signelipm subroutine 243 signals SIGRECONFIG 353, 354 dynamic logical partitioning 355 sigprocmask subroutine 243 signals SIGRECONFIG 353, 354 dynamic logical partitioning 355 sigprocals subroutine 243 signals subroutine 244 signal subroutine 245 signals subroutine 246 signal subroutine 551 modifying subsystem object definitions 551 removing subserver object definitions 551 removing subsystem object definitions 551 signogramming requirements 545 sockets overview 544 programming requirements 545 sockets overview 544 subserver type object overview 541 subsystem environment object overview 547 subsystem programming requirements 547	•	
screen types SMIT 501 scriptinfo command dynamic logical partitioning 356, 359 searching and sorting example program 495 sed command starting the editor 365 using string replacement 370 using text in commands 369 using the command summary 366 selectors SMIT 502 seleptors SMIT 503 seleptors creating 375 lazy loading 373 shared libraries creating 373 shared memory mmap comparison with shmat 380 overview 379 shared objects 371 creating 373 shared memory shared objects 371 creating 373 shared libraries divide-and-conquer 193 masterislave 193 spacino subroutine 243 siglocino dynamic logical partitioning 354 siglaction subroutine 243 siglonginp subroutine 244 siglonginp subroutine 243 siglonginp subroutine 244 signal delivery 245 generation 243 handlers 243, 244 masks 243 siglonging busbroutine 243 signals SIGRECONFIG dynamic logical partitioning 353, 354 dynamic logical partitioning 244 sigwait subroutine 243 signals subroutine 244 signals subroutine 243 signals subroutine 245 signals subroutine 246 signals producer object definitions 551 modifying subsystem object definitions 551 removing subsystem object definitions		
SMIT 501 scariptinfo command dynamic logical partitioning 356, 359 searching and sorting example program 495 sod command starting the editor 365 using string replacement 370 using text in commands 369 using the command summary 366 selectors SMIT 502 semaphores (from OSF/1) 240 setil program 375 lazy loading 373 shared memory mmap comparison with shnat 380 overview 379 shared objects 371 creating 373 shared memory mmap comparison with shnat 380 overview 379 shared objects 371 creating 373 ship license file 426 signation subroutine 243 signation subroutine 244 signal delivery 245 generation 243 handlers 243, 244 masks 243 signal SIGRECONFIG 353, 354 dynamic logical partitioning 355 sigprocmask subroutine 243 signals SIGRECONFIG 353, 354 dynamic logical partitioning 355 sigprocmask subroutine 243 signals subroutine 244 signal subroutine 245 signals subroutine 245 signals subroutine 246 signals subroutine 355 signorgmask subroutine 243 signals subroutine 244 signal subroutine 245 signals subroutine 246 signals subroutine 357 signals subroutine 247 signal subroutine 358 signals subroutine 248 signals subroutine 249 signals subroutine 249 signals subroutine 355 signals subroutine 244 signal subroutine 355 signals subroutine 245 signals subroutine 354 signals subroutine 355 signals subroutine 355 signals subroutine 355 signals subroutine 355 signals subroutine 358 signals subroutine 359 signals subroutine 350 security subroutine 351 signals subroutine 351 signals subroutine 351 signals subroutine 351 signals subroutine 353 signals subroutine 354 signals subroutine 355 signals subroutine 35	-	-
scriptinfo command dynamic logical partitioning 356, 359 searching and sorting example program 495 designing 502 object class for allases 515 object class sed command starting the editor 365 using string replacement 370 using text in commands 369 using the command summary 366 selectors SMIT 502 semaphores (from OSF/1) 240 setimp subroutine 244 sared libraries creating 375 lazy loading 373 smit errolear command 112 smit error command 560 SNA generic alert architecture 113 software models divide-and-conquer 193 masster/slave 193 producer/consumer 193 sigaction subroutine 243 signals delivery 245 generation 243 handlers 243, 244 masks 243 signals SIGRECONFIG dynamic logical partitioning 355 signoromask subroutine 243 signals SIGRECONFIG dynamic logical partitioning 355 signoromask subroutine 243 signals subroutine 245 signal delivery 245 generation 246 handlers 247 signal delivery 245 generation 248 handlers 248, 244 masks 243 signals subroutine 243 signals subroutine 243 signals subroutine 244 signal massing subroutine 245 signals subroutine 246 signals 542 programming requirements 546 signals 542 programming requirements 546 signals 542 programming requirements 546 signals 542 programming requirements 545 sockets overview 541 subserver type object overview 541 subserver type object overview 541 subserver programming requirements 545 sockets overview 547 subserver programming requirements 547		
dynamic logical partitioning 356, 359 searching and sorting example program 495 sed command starting the editor 365 using string replacement 370 using text in commands 369 using the command summary 366 selectors SMIT 502 semaphores (from OSF/1) 240 selector header 516 selector head		
searching and sorting example program 495 sed command starting the editor 365 using string replacement 370 using text in commands 369 using the command summary 366 selectors SMIT 502 semaphores (from OSF/1) 240 setimp subroutine 244 sate object and starting 373 shared memory mmap comparison with shmat 380 overview 379 shared objects 371 creating 373 shared memory mmap comparison with shmat 380 overview 379 shared objects 371 creating 373 ship license file 426 sigaction dynamic logical partitioning 354 sigaction subroutine 244 signal delivery 245 generation 243 handlers 243, 244 masks 243 signals SIGRECONFIG 353, 354 dynamic logical partitioning 355 signorism subroutine 244 sight adding 516 per signal signals SIGRECONFIG 353, 354 dynamic logical partitioning 355 signorism subroutine 244 sight adding 516 generation 244 sight adding 516 generation 245 signals SIGRECONFIG 353, 354 dynamic logical partitioning 355 signorism subroutine 244 sight adding 516 generation 245 signals SIGRECONFIG 353, 354 dynamic logical partitioning 355 signorism subroutine 244 sight adminishment 245 signals SIGRECONFIG 353, 354 dynamic logical partitioning 355 signorism subroutine 244 sight adminishment 245 signals SIGRECONFIG 353, 354 dynamic logical partitioning 355 signorism subroutine 244 sight adminishment 244 sight adminishment 245 signals 542 programming requirements 546 signals 542 programming requirements 545 sockets overview 544 programming requirements 545 sockets overview 544 programming requirements 545 sockets overview 541 subserver type object overview 541 subserver type object overview 541 subserver type object overview 547 fast paths 507 generating commands with 510		
example program 495 sed command starting the editor 365 using string replacement 370 using text in commands 369 using the command summary 366 selectors SMIT 502 semaphores (from OSF/1) 240 setimp subroutine 244 shared libraries creating 375 lazy loading 373 shared memory mana comparison with shmat 380 overview 379 shared objects 371 creating 373 shared objects 371 creating 373 shared objects 371 creating 374 ship license file 426 sigaction dynamic logical partitioning 354 signal delivery 245 generation 243 handlers 243, 244 masks 243 signals SIGRECONFIG 353, 354 dynamic logical partitioning 355 signoremask subroutine 243 signals SIGRECONFIG dynamic logical partitioning 355 signoremask subroutine 244 size 425 SMIT (System Management Interface Tool) aliases 507 dialogs designing 503 executing 511 generating 510 example program 524 fast paths 507 generating commands with 510 for allases 515 object classes dialog 518 dialog beader 522 menu 514 selector header 516 selector he		
sed command starting the editor 365 using string replacement 370 using text in commands 369 using the command summary 366 selectors SMIT 502 semaphores (from OSF/1) 240 setimp subroutine 244 shared libraries creating 375 slazy loading 373 shared memory mmap comparison with shmat 380 overview 379 shared objects 371 creating 373 ship license file 426 signal delivery 245 generation 243 handlers 243, 244 masks 243 signals SIGRECONFIG dynamic logical partitioning 353, 354 dynamic logical partitioning 353, 354 dynamic logical partitioning 353, 354 dynamic logical partitioning 244 signed signor 244 signed signor 243 SIGRECONFIG dynamic logical partitioning 353, 354 dynamic logical partitioning 244 signed signor 244 signed subroutine 244 signed partitioning 353, 354 dynamic logical partiti	-	•
starting the editor 365 using string replacement 370 using text in commands 369 using the command summary 366 selectors SMIT 502 semaphores (from OSF/1) 240 setimp subroutine 244 shared libraries creating 375 lazy loading 373 shared memory mmap comparison with shmat 380 overview 379 shared objects 371 creating 373 ship license file 426 sigaction dynamic logical partitioning 354 signongimp subroutine 244 sightneadmask subroutine 243 signoles SIGRECONFIG 353, 354 delivery 245 dynamic logical partitioning 355 signoremask subroutine 243 signoremask subroutine 243 signoremask subroutine 244 sigthreadmask subroutine 243 signostimp subroutine 244 sigthreadmask subroutine 245 signal 542 programming requirements 545 sockets sockets overview 544 programming requirements 545 sockets sockets sockets overview 541 subsystem object overview 541 subsystem programming requirements programming requirements sockets 547	to the second control of the second control	
using string replacement 370 using text in commands 369 using the commands 369 using the commands 369 using the commands 369 using the commands 369 selectors SMIT 502 semaphores (from OSF/1) 240 selector header 516 understanding 504 seks semaphores (from OSF/1) 240 soft tasks selector header 516 understanding 504 seks semaphores (from OSF/1) 240 soft tasks selector header 516 understanding 504 seks semaphores (from OSF/1) 240 soft tasks selector header 516 understanding 504 seks semaphores (from OSF/1) 240 soft tasks selector header 516 understanding 504 seks semaphores (from OSF/1) 240 software models debugging 513 smit errolear command 112 smit trace command 112 smit tra		•
using text in commands 369 using the command summary 366 selectors SMIT 502 semaphores (from OSF/1) 240 setimp subroutine 244 shared libraries creating 375 lazy loading 373 shared memory mmap comparison with shmat 380 overview 379 shared objects 371 creating 373 ship license file 426 sigaction dynamic logical partitioning 354 signals delivery 245 generation 243 signals SIGRECONFIG 353, 354 dynamic logical partitioning 355 signals SIGRECONFIG 353, 354 signals SIGRECONFIG 353, 354 sigwait subroutine 243 sigwait subroutine 244 sigther admask subroutine 243 sigwait subroutine 244 sigther admask subroutine 243 sigwait subroutine 243 sigvaling subroutine 243 sigvaling subroutine 243 sigvaling subroutine 243 signals SIGRECONFIG 353, 354 sigvaling subroutine 244 sigther admask subroutine 243 sigvaling subroutine 244 sigther admask subroutine 243 sigvaling subroutine 244 sigther admask subroutine 243 sigvaling subroutine 244 signals solvatine 245 sigvaling subroutine 246 sigvaling subroutine 247 sigvaling subroutine 248 sigvaling subroutine 249 sigvaling subroutine 240 sigvaling subroutine 241 sigvaling subroutine 243 sigvaling subroutine 243 sigvaling subroutine 244 sight subroutine 245 sigvaling subroutine 246 sigvaling subroutine 247 sigvaling subroutine 248 sigvaling subroutine 249 sigvaling subroutine 249 sigvaling subroutine 240 sigvaling subroutine 241 signals 542 programming requirements 545 sockets overview 544 programming requirements 545 SRC object classes descriptors 538 notify object overview 541 subserver type object overview 547 subserver type object overview 547 subserver type object overview 548 subserver type object overview 549 subserver sockets sockets overview 544 subserver type object overview 541 subserver type object overview 541 subserver type object overview 541 subsystem environment object overview 548 subserver type object overview 549 subserver type object overview 541 subsystem environme	•	
using the command summary 366 selectors SMIT 502 semaphores (from OSF/1) 240 signals 242 signals 243 signals 243 signals 244 signals 243 signals 244 signals 245 sockets overview 544 programming requirements 545 sockets overview 544 programming requirements 5		
selectors SMIT 502 semaphores (from OSF/1) 240 setimp subroutine 244 shared libraries creating 375 lazy loading 373 shared memory mmap comparison with shmat 380 overview 379 shared objects 371 creating 373 ship license file 426 sigaction dynamic logical partitioning 354 signals signals IGECONFIG 353, 354 dynamic logical partitioning 355 sigprocmask subroutine 243 signals SIGRECONFIG 353, 354 dynamic logical partitioning 353, 354 sigtime admask subroutine 243 sigue processing 503 executing 511 generating 501 example program 524 fast paths 507 generating commands with 510 understanding 501 screen types 501 tasks screen types 501 tasks adding 512 screen types 501 tasks adding 512 screen types 501 tasks adding 512 sdadking 512 sdadking 513 smit errolear command 112		
SMIT 502 semaphores (from OSF/1) 240 setimp subroutine 244 shared libraries creating 375 lazy loading 373 shared memory mmap comparison with shmat 380 overview 379 shared objects 371 creating 373 ship license file 426 sigaction dynamic logical partitioning 354 siglonging subroutine 243 signals SIGRECONFIG 353, 354 dynamic logical partitioning 355 signals SIGRECONFIG 353, 354 dynamic logical partitioning 355 sigprocreask subroutine 243 signals SIGRECONFIG dynamic logical partitioning 353, 354 sigsterion 243 signals SIGRECONFIG 353, 354 dynamic logical partitioning 353, 354 signals SIGRECONFIG 353, 354 dynamic logical partitioning 355 sigprocreask subroutine 243 signals SIGRECONFIG 353, 354 dynamic logical partitioning 355 sigprocreask subroutine 243 signals SIGRECONFIG 353, 354 dynamic logical partitioning 355, 354 signals subroutine 243 signals SIGRECONFIG 353, 354 dynamic logical partitioning 355, 354 signals subroutine 243 signals subroutine 243 signals SIGRECONFIG 353, 354 dynamic logical partitioning 355, 354 dynamic logical partitioning 355, 354 dynamic logical partitioning 355, 354 signals subroutine 243 signals subroutine 243 signals subroutine 244 signerating 510 executing 511 generating 510 example program 524 fast paths 507 generating commands with 510 sorterate removing subserver to the SRC object class 51 subsystem object definitions 551 subsystem subroutine 545 sockets		
semaphores (from OSF/1) 240 setimp subroutine 244 sarded libraries creating 375 lazy loading 373 shared memory mmap comparison with shmat 380 overview 379 shared objects 371 creating 373 ship license file 426 sigaction dynamic logical partitioning 354 signall delivery 245 generation 243 handlers 243, 244 masks 243 signals SIGRECONFIG 353, 354 dynamic logical partitioning 355 sigrorcmask subroutine 243 sigsetimp subroutine 243 signals SIGRECONFIG dynamic logical partitioning 355 sigrorcmask subroutine 243 sigtiffreadmask subroutine 243 sigenting 507 dialogs designing 503 executing 511 generating 510 example program 524 fast paths 507 generating commands with 510 tasks adding 512 debugging 513 smit errolear command 112 smit trace command 112 smit errolear command 112 smit trace command 12 smit errolear command 12 smit errolear command 112 smit trace command 12 smit errolear command 112 smit trace command 12 smit errolear centeritieter 113 software models divide-and-conquer 193 mater/slave 193 producer/consumer 193 software models divide-and-conquer 193 software models divid		•
setimp subroutine 244 shared libraries creating 375 lazy loading 373 shared memory mmap comparison with shmat 380 overview 379 shared objects 371 creating 373 ship license file 426 sigaction subroutine 243 siglongimp subroutine 243 siglongimp subroutine 243 siglongimp subroutine 243 handlers 243, 244 masks 243 SIGRECONFIG 353, 354 dynamic logical partitioning 355 sigprocrask subroutine 243 sigpromask subroutine 243 sigsetimp subroutine 244 sigthreadmask subroutine 243 sigsetimp subroutine 243 sigsetimp subroutine 243 signali subroutine 243 handlers 243, 244 masks 243 SIGRECONFIG 353, 354 dynamic logical partitioning 355 sigprocrask subroutine 243 sigprofile subroutine 244 sigthreadmask subroutine 245 SIGRECONFIG 353, 354 designing 503 executing 511 generating 510 example program 524 fast paths 507 generating commands with 510 search and single program 524 fast paths 507 senerating commands with 510 search and single program face under the face to solve the service solve the service of the service object definitions of 51 subserver object definitions 551 removing subsystem object definitions 551 removing subsystem object definitions 551 signals 542 programming requirements 546 signals 542 programming requirements 545 sockets overview 544 programming requirements 545 sockets sockets sockets sockets sockets overview 541 subsystem environment object overview 541 subsystem environment object overview 538 SRC subsystem programming requirements 545 special program for equirement object overview 538 specialing commands with 510		
shared libraries creating 375 lazy loading 373 shared memory mmap comparison with shmat 380 overview 379 shared objects 371 creating 373 ship license file 426 sigaction dynamic logical partitioning 354 signals signals SIGRECONFIG 353, 354 dynamic logical partitioning 355 signals SIGRECONFIG 353, 354 dynamic logical partitioning 355 sigprocmask subroutine 243 Sigartion 243 Signals SIGRECONFIG 353, 354 dynamic logical partitioning 355 Signals 542 programming requirements 546 signals 542 programming requirements 545 Signals 545 SMIT (System Management Interface Tool) aliases 507 dialogs designing 503 executing 511 generating 510 example program 524 fat spaths 507 generating command 112 smit trace command 1560 SNA generic alert architecture 113 software vitale Product Data (SWVPD) 416 SRC 537 capabilities 538 defining subservers to the SRC object class 551 init command 537 list of subroutines 552 init command 537 list of subroutines 551 modifying subserver object definitions 551 removing subseystem object definitions 551 removing requirements 546 signals 542 programming requirements 545 sockets overview 544 programming requirements 545 sockets overview 544 subserver type object overview 541 subserver type object overview 541 subserver type object overview 548 subsystem environment object overview 538 subsystem provironment object overview 538 subsystem provironment object overview 547		
creating 375 lazy loading 373 shared memory mmap comparison with shmat 380 overview 379 shared objects 371 creating 373 ship license file 426 sigaction dynamic logical partitioning 354 siglongimp subroutine 243 siglongimp subroutine 244 sighreadmask subroutine 243 sigsetimp subroutine 244 sighreadmask subroutine 243 sigwait subroutine 244 sighreadmask subroutine 244 sighreadmask subroutine 244 sighreadmask subroutine 243 sigwait subroutine 244 sighreadmask subroutine 244 sighreadmask subroutine 244 sighreadmask subroutine 244 sighreadmask subroutine 243 sigwait subroutine 244 sighreadmask subroutine 244 sighreadmask subroutine 244 sighreadmask subroutine 244 sighreadmask subroutine 243 sigwait subroutine 245 sigwait subroutine 246 signals 542 programming requirements 545 sockets overview 544 programming requirements 545 sockets overview 544 programming requirements 545 sockets overview 544 subsystem object overview 541 subsystem object overview 541 subsystem object overview 541 subsystem object overview 547 fat spaths 507 generating commands with 510		
lazy loading 373 shared memory mmap comparison with shmat 380 overview 379 shared objects 371 creating 373 ship license file 426 sigaction dynamic logical partitioning 354 signal delivery 245 generation 243 handlers 243, 244 masks 243 signals SIGRECONFIG 353, 354 dynamic logical partitioning 355 sigprocmask subroutine 243 signals SIGRECONFIG dynamic logical partitioning 355 sigprocmask subroutine 244 sigthreadmask subroutine 244 sigwait subroutine 244 sightreadmask subroutine 245 sigwait subroutine 246 signal for subroutine 247 sigwait subroutine 248 sigwait subroutine 249 sigwait subroutine 240 sigwait subroutine 241 sigwait subroutine 242 sigwait subroutine 243 sigwait subroutine 244 sightreadmask subroutine 245 sigwait subroutine 250 SMIT (System Management Interface Tool) aliases 507 dialogs designing 503 executing 511 generating 510 example program 524 fast paths 507 generating commands with 510 smit trace command 560 SNA generic alert architecture 113 software rotelecture 113 software rotelecture 113 software rotelecture 113 software models divide-and-conquer 193 master/slave 193 producer/consumer 193 software worlal Product Data (SWVPD) 416 SRC 537 capabilities 538 defining subsystems to the SRC object class 551 init command 537 list of subroutines 552 modifying subsystem object definitions 551 modifying subsystem object definitions 551 removing subsystem		
shared memory mmap comparison with shmat 380 overview 379 shared objects 371 oreating 373 ship license file 426 sigaction dynamic logical partitioning 354 siglongimp subroutine 243 siglongimp subroutine 244 signals delivery 245 generation 243 handlers 243, 244 masks 243 signals SIGRECONFIG 353, 354 dynamic logical partitioning 355 sigprocmask subroutine 243 SIGRECONFIG dynamic logical partitioning 355 sigperomask subroutine 244 sigetimp subroutine 244 sigetimp subroutine 243 SIGRECONFIG dynamic logical partitioning 353, 354 sigpestimp subroutine 244 sigstimp subroutine 243 sigwait subroutine 245 sigwait subroutine 245 sigwait subroutine 250 designing 503 executing 511 generating 510 example program 524 fatst paths 507 generating commands with 510 software models software models divide-and-conquer 193 master/slave 193 producer/consumer 193 Software Vital Product Data (SWVPD) 416 SRC 537 capabilities 538 defining subservers to the SRC object class 551 init command 537 list of subroutines 552 modifying subsystem object definitions 551 removing subsystem object definitions 551 space object defi		
mmap comparison with shmat 380 overview 379 software models shared objects 371 divide-and-conquer 193 master/slave 193 producer/consumer 193 master/slave 193 producer/consumer 193 sigaction dynamic logical partitioning 354 siglongimp subroutine 243 signals delivery 245 generation 243 handlers 243, 244 masks 243 signals SIGRECONFIG 353, 354 dynamic logical partitioning 355 sigrocmask subroutine 243 signals sIGRECONFIG 353, 354 dynamic logical partitioning 355 removing subsystem object definitions 551 removing subsystem object def	-	·
soltware models shared objects 371 creating 373 ship license file 426 sigaction dynamic logical partitioning 354 signal delivery 245 generation 243 handlers 243, 244 masks 243 signals SIGRECONFIG 353, 354 dynamic logical partitioning 355 sigprocramsk subroutine 243 sigprocramsk subroutine 243 sigprocramsk subroutine 244 sigestipm subroutine 355 signal delivery 245 generation 243 handlers 243, 244 masks 243 signals SIGRECONFIG 353, 354 dynamic logical partitioning 355 sigprocramsk subroutine 243 SIGRECONFIG 353, 354 dynamic logical partitioning 355 sigprocramsk subroutine 244 sigprocramsk subroutine 243 SIGRECONFIG dynamic logical partitioning 353, 354 signals subroutine 244 signals subroutine 245 signals subroutine 246 signals subroutine 247 signals removing subsystem object definitions 551 signals removing subsystem object definitions 551 signals removing subsystem object definitions 551 signals operations 538 removing subsystem object definitions 551 removing subsystem object def	•	
shared objects 371 creating 373 ship license file 426 sigaction dynamic logical partitioning 354 siglorigmp subroutine 243 signal delivery 245 generation 243 handlers 243, 244 masks 243 signals SIGRECONFIG 353, 354 dynamic logical partitioning 355 signals signals subserver volject definitions 551 removing subserver object definitions 551 remo		<u> </u>
creating 373 ship license file 426 sigaction dynamic logical partitioning 354 siglongjmp subroutine 243 siglongjmp subroutine 244 signal delivery 245 generation 243 handlers 243, 244 masks 243 signals SIGRECONFIG 353, 354 dynamic logical partitioning 355 sigprocases subroutine 243 sigprocases subroutine 243 signals SIGRECONFIG 353, 354 dynamic logical partitioning 355 sigprocases subroutine 243 SIGRECONFIG dynamic logical partitioning 355, 354 sigsetimp subroutine 243 SIGRECONFIG dynamic logical partitioning 353, 354 sigsetimp subroutine 243 SIGRECONFIG dynamic logical partitioning 353, 354 sigsetimp subroutine 243 SIGRECONFIG dynamic logical partitioning 355, 354 sigsetimp subroutine 243 sigsetimp subroutine 243 sigwait subroutine 243 sigwait subroutine 243 sigwait subroutine 215, 244 sigthreadmask subroutine 243 sigwait subroutine 215, 244 sightreadmask subroutine 243 sigwait subroutine 215, 244 sightreadmask subroutine 243 sightreadmask subroutine 244 sightreadmask subroutine 245 sightreadmask subroutine 246 sightreadmask subroutine 247 sightreadmask subroutine 248 sightreadmask subroutine 248 sightreadmask subroutine 248 sightreadm		
ship license file 426 sigaction Software Vital Product Data (SWVPD) 416 dynamic logical partitioning 354 sigaction subroutine 243 siglongimp subroutine 244 signal delivery 245 generation 243 handlers 243, 244 masks 243 signals SIGRECONFIG 353, 354 dynamic logical partitioning 355 signals SIGRECONFIG dynamic logical partitioning 355 sigprocmask subroutine 243 SIGRECONFIG dynamic logical partitioning 353, 354 sigsetimp subroutine 244 sigsetimp subroutine 244 sigsetimp subroutine 244 sigwait subroutine 243 sigwait subroutine 243 sigwait subroutine 244 sigwait subroutine 243 sigwait subroutine 243 sigwait subroutine 244 sigwait subroutine 245 SMIT (System Management Interface Tool) aliases 507 dialogs designing 503 executing 511 generating 510 example program 524 fast paths 507 generating commands with 510 producer/consumer 193 Software Vital Product Data (SWVPD) 416 SRC 537 capabilities 538 defining subserver to the SRC object class 551 defining subsystems to the SRC object class 551 init command 537 lefining subsystems object definitions 551 modifying subserver object definitions 551 modifying subserver object definitions 551 removing subsystem object definitions 551 removin		·
sigaction dynamic logical partitioning 354 sigaction subroutine 243 siglongimp subroutine 244 siglongimp subroutine 244 defining subservers to the SRC object class 551 defining subservers to the SRC object class 551 defining subsystems to the SRC object class 551 init command 537 generation 243 handlers 243, 244 masks 243 signals SIGRECONFIG 353, 354 dynamic logical partitioning 355 signals SIGRECONFIG 353, 354 dynamic logical partitioning 355 signals subroutine 243 SIGRECONFIG 353, 354 dynamic logical partitioning 355 signals subroutine 243 SIGRECONFIG subsystem object definitions 551 removing subsystem object definitions 551 removing subsystem object definitions 551 sigprocmask subroutine 243 sigwait subroutine 244 sigthreadmask subroutine 244 sigthreadmask subroutine 243 sigwait subroutine 244 sigthreadmask subroutine 243 sigwait subroutine 215, 244 size 425 SMIT (System Management Interface Tool) aliases 507 dialogs designing 503 executing 511 generating 510 example program 524 fast paths 507 senerating commands with 510 Senerating commands with 510		
dynamic logical partitioning 354 sigaction subroutine 243 siglongjmp subroutine 244 signal defining subservers to the SRC object class 551 delivery 245 generation 243 handlers 243, 244 masks 243 signals logical partitioning 355 signals signal subroutine 243 signal subroutine 243 signal subroutine 244 sightreadmask subroutine 244 sightreadmask subroutine 243 sigwait subroutine 215, 244 sigwait subroutine 215, 244 size 425 SMIT (System Management Interface Tool) aliases 507 dialogs designing 503 executing 511 generating 510 example program 524 fast paths 507 generating commands with 510 SRC 537 capabilities 538 defining subserver to the SRC object class 551 init command 537 list of subroutines 552 modifying subserver object definitions 551 removing subserver object definitio		·
sigaction subroutine 243 siglongimp subroutine 244 signal defining subservers to the SRC object class 551 defining subservers to the SRC object class 551 defining subsystems to the SRC object class 551 defining subsystems to the SRC object class 551 init command 537 list of subroutines 552 modifying subserver object definitions 551 modifying subsystem object definitions 551 modifying subsystem object definitions 551 modifying subserver object definitions 551 modifying subserver object definitions 551 signals SIGRECONFIG 353, 354 dynamic logical partitioning 355 sigprocrask subroutine 243 SIGRECONFIG dynamic logical partitioning 353, 354 sigsetjmp subroutine 244 sigstimeadmask subroutine 244 sigthreadmask subroutine 243 sigwait subroutine 244 sigwait subroutine 245 sigwait subroutine 215, 244 size 425 SMIT (System Management Interface Tool) aliases 507 dialogs designing 503 executing 511 generating 510 example program 524 fast paths 507 generating commands with 510 capabilities 538 defining subservers to the SRC object class 551 defining subservers to the SRC object class 551 interior subsystem endifying subserver object definitions 551 modifying subserver object definitions 551 speration 538 removing subserver object definitions 551 removing subserver object definition		
siglongimp subroutine 244 signal defining subservers to the SRC object class 551 defining subsystems to the SRC object class 551 defining subsystems to the SRC object class 551 defining subsystems to the SRC object class 551 init command 537 list of subroutines 552 handlers 243, 244 masks 243 modifying subserver object definitions 551 modifying subsystem object definitions 551 signals SIGRECONFIG 353, 354 dynamic logical partitioning 355 sigprocmask subroutine 243 SIGRECONFIG dynamic logical partitioning 353, 354 sigsetjmp subroutine 244 sigstimp subroutine 243 sigwait subroutine 243 sigwait subroutine 215, 244 size 425 SMIT (System Management Interface Tool) aliases 507 dialogs designing 503 executing 511 generating 510 example program 524 fast paths 507 generating commands with 510 defining subservers to the SRC object class 551 defining subsystems to the SRC object class 551 signals 342 modifying subserver object definitions 551 removing subsystem object definitions 551 sperations 538 removing subserver object definitions 551 semoving subsystem object definitions 551 removing subsystem object definitions 551 semoving subserver object definitions 551 semoving subserver object definitions 551 semoving subsystem object definitions 551 semoving subsystem object definitions 551 semoving subserver object definiti		
signal defining subsystems to the SRC object class 551 delivery 245 generation 243 list of subroutines 552 modifying subserver object definitions 551 modifying subserver object definitions 551 modifying subserver object definitions 551 operations 538 signals operations 538 removing subserver object definitions 551 removing subserver object definitions 551 removing subsystem object definitions 551 sigprocmask subroutine 243 SRC communication types message queues (IPC) overview 544 programming requirements 546 signals 542 programming requirements 545 sockets SMIT (System Management Interface Tool) aliases 507 dialogs designing 503 descriptors 538 executing 511 subserver type object overview 541 subsystem programm 524 subsystem programming requirements 545 sackets subsystem object overview 541 subsystem programming requirements 545 secrets fast paths 507 SRC subsystem programming requirements 545 secrets subsystem programming requirements 545	-	
delivery 245 generation 243 handlers 243, 244 masks 243 signals SIGRECONFIG 353, 354 dynamic logical partitioning 355 sigprocmask subroutine 243 SIGRECONFIG dynamic logical partitioning 353, 354 sigsetimp subroutine 244 sigsetimp subroutine 243 sigwait subroutine 243 sigwait subroutine 243 sigwait subroutine 243 sigwait subroutine 244 sigwait subroutine 245 sigwait subroutine 245 sigwait subroutine 246 signals SMIT (System Management Interface Tool) aliases 507 dialogs designing 503 executing 511 generating 507 generating 507 generating commands with 510 init command 537 list of subroutines 552 modifying subserver object definitions 551 removing subsystem object definitions 551 sperations 538 removing subsystem object definitions 551 semoving subsystem object definitions 551 removing subsystem object definitions 551 removing subsystem object definitions 551 sigprocmask subroutine 551 sermoving subsystem object definitions 551 removing subsystem object definitions 551 removing subsystem object definitions 551 sperations 538 removing subserver object definitions 551 removing subserver object definitions 551 sperations 538 removing subsystem object definitions 551 removing subsystem object definitions 551 sperations 538 removing subserver object definitions 551 sperations 538 removing subserver object definitions 551 removing subsystem object definitions 551 removing subsystem object definitions 551 sperations 538 removing subsystem object definitions 551 removing subsystem object definitions 551 removing subsystem object definitions 551 removing subsystem 538 serc communication types message queus (IPC) overview 544 programming requirements 546 signals 42 programming requirements 545 sockets socket		
generation 243 handlers 243, 244 masks 243 signals SIGRECONFIG 353, 354 dynamic logical partitioning 355 sigprocmask subroutine 243 SIGRECONFIG dynamic logical partitioning 355 sigprocmask subroutine 243 SIGRECONFIG dynamic logical partitioning 353, 354 sigsetimp subroutine 244 sigsetimp subroutine 244 sightreadmask subroutine 243 sigwait subroutine 215, 244 size 425 SMIT (System Management Interface Tool) aliases 507 dialogs designing 503 executing 511 generating 510 example program 524 fast paths 507 generating commands with 510 list of subroutines 552 modifying subserver object definitions 551 modifying subserver object definitions 551 sprocrations 538 removing subserver object definitions 551 sproving subserver object definition		
handlers 243, 244 masks 243 modifying subserver object definitions 551 masks 243 signals SIGRECONFIG 353, 354 dynamic logical partitioning 355 sigprocmask subroutine 243 SIGRECONFIG dynamic logical partitioning 353, 354 sigsetjmp subroutine 244 sigsetjmp subroutine 244 sigsetjmp subroutine 243 sigwait subroutine 215, 244 size 425 SMIT (System Management Interface Tool) aliases 507 dialogs designing 503 executing 511 generating 510 example program 524 fast paths 507 generating commands with 510 modifying subserver object definitions 551 modifying subsystem object definitions 551 sproducine 548 removing subsystem object definitions 551 sproducine 543 spreading 544 removing subsystem object definitions 551 spreading 548 removing subsystem object definitions 551 spreading 549 removing subsystem object definitions 551 spreading 540 overview 544 programming requirements 545 sockets overview 544 programming requirements 545 sockets sockets SPRC object classes descriptors 538 executing 511 generating 510 subserver type object overview 541 subsystem environment object overview 538 SPRC subsystem programming requirements processing SPRC request packets 547		
masks 243 signals SIGRECONFIG 353, 354 dynamic logical partitioning 355 sigprocmask subroutine 243 SIGRECONFIG dynamic logical partitioning 353, 354 sigsetjmp subroutine 244 sigsetjmp subroutine 243 sigwait subroutine 243 sigwait subroutine 245 sigmait subroutine 245 sigmait subroutine 245 sigmait subroutine 246 sigmait subroutine 247 size 425 SMIT (System Management Interface Tool) aliases 507 dialogs designing 503 executing 511 generating 510 example program 524 fast paths 507 generating commands with 510 modifying subsystem object definitions 551 space removing subserver object definitions 551 removing subsystem object definitions 551 space removing subsystem object definitions 551 space removing subserver object definitions 551 space removing subserver object definitions 551 removing subserver object definitions 551 space removin		
signals SIGRECONFIG 353, 354 dynamic logical partitioning 355 sigprocmask subroutine 243 SIGRECONFIG dynamic logical partitioning 355 SRC communication types SIGRECONFIG dynamic logical partitioning 353, 354 sigsetimp subroutine 244 sigsetimp subroutine 244 sigsetimp subroutine 243 sigwait subroutine 243 sigwait subroutine 215, 244 size 425 SMIT (System Management Interface Tool) aliases 507 dialogs designing 503 executing 511 generating 510 example program 524 fast paths 507 generating commands with 510 operations 538 removing subserver object definitions 551 secondary object definitions 551 removing subserver object definitions 551 removing subserver object definitions 551 removing subserver object definitions 551 secondary object definitions 551 removing subserver object definitions 551 removing subserver object definitions 551 secondary object definitions 551 removing subserver object definitions 551 secondary object definitions 551 subserver we subsystem object definitions 551 subserver we object definitions 551 secondary object definitions 551 secondary object definitions 551 secondary object definitions 551 subserver we object definitions 551 secondary object definitions 546 signals 542 signals 542 signals 542 signals 542 signals 54		
SIGRECONFIG 353, 354 dynamic logical partitioning 355 sigprocmask subroutine 243 SIGRECONFIG dynamic logical partitioning 353, 354 sigsetjmp subroutine 244 sigsetjmp subroutine 243 sigwait subroutine 244 size 425 SMIT (System Management Interface Tool) aliases 507 dialogs designing 503 executing 511 generating 510 example program 524 fast paths 507 generating commands with 510 SRC communication types message queues (IPC) overview 544 programming requirements 546 signals 542 programming requirements 545 sockets Sockets SRC object classes descriptors 538 notify object overview 541 subserver type object overview 541 subsystem environment object overview 538 SRC subsystem programming requirements processing SRC request packets 547		
dynamic logical partitioning 355 sigprocmask subroutine 243 SIGRECONFIG dynamic logical partitioning 353, 354 sigsetjmp subroutine 244 sigsetjmp subroutine 244 sigthreadmask subroutine 243 sigwait subroutine 215, 244 size 425 SMIT (System Management Interface Tool) aliases 507 dialogs designing 503 executing 511 generating 510 example program 524 fast paths 507 generating commands with 510 SRC communication types message queues (IPC) overview 544 programming requirements 546 signals 542 programming requirements 545 sockets Sockets SRC object classes descriptors 538 notify object overview 541 subserver type object overview 541 subsystem environment object overview 538 SRC subsystem programming requirements processing SRC request packets 547		•
sigprocmask subroutine 243 SRC communication types message queues (IPC) overview 544 sigsetjmp subroutine 244 sightreadmask subroutine 243 sigwait subroutine 215, 244 size 425 SMIT (System Management Interface Tool) aliases 507 dialogs designing 503 executing 511 generating 510 example program 524 fast paths 507 generating commands with 510 SRC communication types message queues (IPC) overview 544 programming requirements 546 signals 542 programming requirements 545 sockets Sockets SRC object classes descriptors 538 notify object overview 541 subserver type object overview 541 subsystem environment object overview 538 SRC subsystem programming requirements processing SRC request packets 547		
SIGRECONFIG dynamic logical partitioning 353, 354 sigsetjmp subroutine 244 sigsetjmp subroutine 243 sigwait subroutine 215, 244 size 425 SMIT (System Management Interface Tool) aliases 507 dialogs designing 503 executing 511 generating 510 example program 524 fast paths 507 generating commands with 510 message queues (IPC) overview 544 programming requirements 546 signals 542 programming requirements 545 sockets Sockets SRC object classes descriptors 538 notify object overview 541 subserver type object overview 541 subsystem environment object overview 538 SRC subsystem programming requirements processing SRC request packets 547		
dynamic logical partitioning 353, 354 sigsetjmp subroutine 244 sigsetjmp subroutine 244 sightreadmask subroutine 243 sigwait subroutine 215, 244 size 425 SMIT (System Management Interface Tool) aliases 507 dialogs designing 503 executing 511 generating 510 example program 524 fast paths 507 generating commands with 510 overview 544 programming requirements 545 sockets overview 541 subscription 545 sockets overview 541 subscription 545 socket		**
sigsetjmp subroutine 244 sigthreadmask subroutine 243 sigwait subroutine 215, 244 size 425 SMIT (System Management Interface Tool) aliases 507 dialogs designing 503 executing 511 generating 510 example program 524 fast paths 507 generating commands with 510 programming requirements 545 sockets overview 544 programming requirements 545 SRC object classes descriptors 538 notify object overview 541 subserver type object overview 541 subsystem environment object overview 538 SRC subsystem programming requirements processing SRC request packets 547		
sightreadmask subroutine 243 sigwait subroutine 215, 244 size 425 SMIT (System Management Interface Tool) aliases 507 dialogs designing 503 executing 511 generating 510 example program 524 fast paths 507 generating commands with 510 signals 542 programming requirements 545 sockets Overview 544 programming requirements 545 SRC object classes descriptors 538 notify object overview 541 subserver type object overview 541 subsystem environment object overview 538 SRC subsystem programming requirements processing SRC request packets 547		
sigwait subroutine 215, 244 size 425 SMIT (System Management Interface Tool) aliases 507 dialogs designing 503 executing 511 generating 510 example program 524 fast paths 507 generating commands with 510 programming requirements 545 SRC object classes descriptors 538 notify object overview 541 subserver type object overview 541 subsystem environment object overview 538 SRC subsystem programming requirements processing SRC request packets 547		
size 425 SMIT (System Management Interface Tool) aliases 507 dialogs designing 503 executing 511 generating 510 example program 524 fast paths 507 generating commands with 510 sockets overview 544 programming requirements 545 SRC object classes descriptors 538 notify object overview 541 subserver type object overview 541 subsystem environment object overview 538 SRC subsystem programming requirements processing SRC request packets 547		-
SMIT (System Management Interface Tool) aliases 507 dialogs designing 503 executing 511 generating 510 example program 524 fast paths 507 generating commands with 510 overview 544 programming requirements 545 SRC object classes descriptors 538 notify object overview 541 subserver type object overview 541 subsystem environment object overview 538 SRC subsystem programming requirements processing SRC request packets 547		
aliases 507 programming requirements 545 dialogs SRC object classes designing 503 descriptors 538 executing 511 notify object overview 541 generating 510 subserver type object overview 541 example program 524 subsystem environment object overview 538 fast paths 507 SRC subsystem programming requirements generating commands with 510 processing SRC request packets 547		
dialogs designing 503 executing 511 generating 510 example program 524 fast paths 507 generating commands with 510 SRC object classes descriptors 538 notify object overview 541 subserver type object overview 541 subsystem environment object overview 538 SRC subsystem programming requirements processing SRC request packets 547		
designing 503 executing 511 generating 510 example program 524 fast paths 507 generating commands with 510 descriptors 538 notify object overview 541 subserver type object overview 541 subsystem environment object overview 538 SRC subsystem programming requirements processing SRC request packets 547		
executing 511 notify object overview 541 generating 510 subserver type object overview 541 example program 524 subsystem environment object overview 538 fast paths 507 SRC subsystem programming requirements generating commands with 510 processing SRC request packets 547	· · · · · · · · · · · · · · · · · · ·	
generating 510 subserver type object overview 541 example program 524 subsystem environment object overview 538 fast paths 507 SRC subsystem programming requirements generating commands with 510 processing SRC request packets 547		
example program 524 subsystem environment object overview 538 fast paths 507 SRC subsystem programming requirements generating commands with 510 processing SRC request packets 547		
fast paths 507 SRC subsystem programming requirements generating commands with 510 processing SRC request packets 547		
generating commands with 510 processing SRC request packets 547	to the second control of the second control	
	·	

SRC subsystem programming requirements (continued) receiving SRC request packets using message queue (IPC) communication 546 using signals communication 545 using sockets communication 545 returning continuation packets 549 returning error packets 550 returning subsystem reply packets 549 srcmstr 538	system calls bindprocessor 354 bind CPU ID 354 dr_reconfig 354 mycpu 352 sigaction 354 sysconfig 352 system environment Dynamic Processor Deallocation 183
stack address POSIX option 248	system file tables 142
stackaddr attribute 248	system resource controller 538, 542, 545, 551, 552
stacksize attribute 248	
status 141	Т
directories 141	-
status subroutines	termcap file 21
overview 153	terminal devices 563
stop subcommand 67	tty subsystem overview 563
string manipulation	terminals
using sed command 365	multiple 18
strings	setting input and output modes 19
adding to windows	terminating a thread 199
curses 14	terminfo file 21
minicurses 14	thread
structures	attributes
dri_cpu 363	contention-scope 231
dri_mem 363	creating 196
sub-windows	destroying 196
refreshing 9	detachstate 197
subroutines	inheritsched 228
reconfig_handler_complete 362 alloca 391	object 196
calloc 391	schedparam 229
chmod 154	schedpolicy 229
chown 154	stackaddr 248
controlling files and directories	stacksize 248
list of 481	binding 182
errlog 114	Compiler Invocation 262
free 391	concurrency level 231
hardware-specific 499	contention scope 231 data types 250
mallinfo 391	default values 251
malloc 391	definition 191
mallopt 391	detached state 225
pthread_getspecific 238	ID 198
pthread_setspecific 238	initial 192
realloc 391	join 225
reconfig_register 361	kernel 193
valloc 391	libpthreads_compat.a library 196
subsystems	libpthreads.a library 196
using the system resource controller 545	library 193
super block 131	limits 250
SWVPD 416	models 194
symbolic debug program, dbx 61	POSIX thread 196, 262
synchronization	properties 192
mutex 211	pthread 196
synchronization scheduling 232	supported 251
definition 232	user 193
sysconf subroutine 249	thread-safe
sysconf system call	SRC subroutines 552
dynamic logical partitioning 352	thread-safe code 256
syslog facility 359	thread-specific data 236
	concurrent creation 236

thread-specific data <i>(continued)</i> definition 235	tty subsystem 563
destroying data 238	
destructor 237	U
destructor routine 237	
key 235	undopreacquire command dynamic logical partitioning 357, 361
setting values 238	undoprerelease command
swapping data 238	dynamic logical partitioning 357, 360
usage 238	unlink subroutine 141
threads	unmount command 156
creating 196	usage command
object-oriented interface 194	dynamic logical partitioning 356, 359
synchronization 206	user thread 193
terminating 199	utimes subroutine 153
threads library 193	
cancelation 203	
cleanup 204	V
condition variable	var.v ncpus field
attributes creation and destruction 212	dynamic logical partitioning 352
synchronization point 217	var.v_ncpus_cfg variable
timed wait 214, 216	dynamic logical partitioning 352
dynamic initialization 234	variables
thread-safe 235	_system_configuration.max_ncpus 352
exiting a thread 199	_system_configuration.ncpus 354
freeing returned data 227	_system_configuration.original_ncpus 352
join 226	var.v_ncpus_cfg 352
mutex	video attributes
attributes creation and destruction 206	setting 25
read/write locks (from POSIX) 241	virtual memory
thread creation 264	addressing
timestamp	overview 377
changing 153	virtual processor 194
trace	vital product data (VPD) 107
configuring 560	VP 194
controlling 555	VPD (vital product data) 107
generating reports 561	VI D (Vital product data)
recording trace event data 560	
starting 560	W
starting a trace 560	windows
stopping 561	clearing 16
using generic trace channels 560	copying 10
trace command 560	curses 5
configuring 560	drawing box around 10
trace facility	moving 10
overview of 555	overlapping 10
trace hook identifiers 560	refreshing 9
tracing	refreshing multiple 9
configuring 560	scrolling 15
starting 560	winsize structure 566
translation requirements 426	Workload Manager
trcrpt command 560, 561	class definitions
trostop command 561	dynamic logical partitioning 355
truncate subroutine 149	dynamic logical partitioning dependencies 355
tty	Software Partitioning 355
current modes 23	write subroutine 148
restoring modes 23	
saving modes 23	
tty (terminal type)	Υ
definition 563	yacc command 279
examples 563	actions 297
tty driver 564	adilotto 201

yacc command (continued)
ambiguous rules 302
creating a parser 290
declarations 294
error handling 299
grammar file 291
rules 296
turning on debug mode 303
yacc program 280
yyleng external variable 287
yyless Subroutine 285
yymore Subroutine 285
yywleng external variable 287
yywleng external variable 287
yywrap Subroutine 285

Readers' Comments — We'd Like to Hear from You

AIX 5L Version 5.2 General Programming Concepts: Writing and Debugging Programs

Phone No.

Publication No. SC23-4	128-08				
Overall, how satisfied a	re you with the inf	ormation in this	book?		
	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction					
How satisfied are you th	nat 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 ca	an improve this bo	ook:			
Thank you for your respon	nses. May we conta	act you? 🗌 Ye	es 🗌 No		
When you send comment	o to IDM you grout	IDM a nanavalus	sive right to use o	r diatributa vaur a	ammanta in any
way it believes appropriat			-	i distribute your c	omments in any
may it believes appropriat	o manour mouning (arry conganori to	you.		
Name		Ad	ldress		
Company or Organization					
company or organization					

Readers' Comments — We'd Like to Hear from You SC23-4128-08



Cut or Fold Along Line

Fold and Tape

Please do not staple

Fold and Tape



NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation Information Development Department H6DS-905-6C006 11501 Burnet Road Austin, TX 78758-3493



Hadlaldadddalalalalddalldadlalal

Fold and Tape

Please do not staple

Fold and Tape

IBM

Printed in U.S.A.

SC23-4128-08

