

IBM

**OS Open
User's Guide**

Sixth Edition (September 1997)

This edition of *IBM OS Open User's Guide* applies to IBM OS Open Version 1.6.1 and to subsequent versions of the OS Open Real-Time Operating System until otherwise indicated in new versions or technical newsletters.

The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS MANUAL "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND 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.

IBM does not warrant that the contents of this publication or the accompanying source code examples, whether individually or as one or more groups, will meet your requirements or that the publication or the accompanying source code examples are error-free.

This publication could contain 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 program(s) described in this publication at any time.

It is possible that this publication may contain references to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country. Any reference to an IBM licensed program in this publication is not intended to state or imply that you can use only IBM's licensed program. You can use any functionally equivalent program instead.

No part of this publication may be reproduced or distributed in any form or by any means, or stored in a data base or retrieval system, without the written permission of IBM.

Requests for copies of this publication and for technical information about IBM products should be made to your IBM Authorized Dealer or your IBM Marketing Representative.

Address comments about this publication to:

IBM Corporation
Department 0H83A
P.O. Box 12195
Research Triangle Park, NC 27709

IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

©Copyright International Business Machines Corporation 1993, 1997. All rights reserved.

Notice to U.S. Government Users—Documentation Related to Restricted Rights—Use, duplication, or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corporation.

Patents, Trademarks, and Acknowledgments

IBM may have patents or pending patent applications covering the subject matter in this publication. The furnishing of this publication does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 208 Harbor Drive, Stamford, CT 06904, United States of America.

The following terms are trademarks of IBM Corporation:

IBM	POWER Architecture
AIX	PowerPC Architecture
OS Open	RISC System/6000
PowerPC	RS/6000
RISCWatch	

The following term is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited:

UNIX

Other terms which are trademarks are the property of their respective owners.

The OS Open software and documentation are based in part on the Fourth Berkeley Software Distribution under license from the Regents of the University of California. We acknowledge the following institution for its role in the development: the Electrical Engineering and Computer Sciences Department at the Berkeley Campus.

Portions of the code and documentation described in this book were derived from code and documentation developed under the auspices of the Regents of the University of California and have been acquired and modified under the provisions that the following copyright notice and permission notice appear:

©Copyright Regents of the University of California, 1986, 1987. All rights reserved.

Redistribution and use in source and binary forms are permitted provided that this notice is preserved and that due credit is given the University of California at Berkeley. The name of the University may not be used to endorse or promote products derived from this software without specific prior written permission. This software is provided "as is" without express or implied warranty.

Some of the C language functions described herein use function names and prototypes derived from the following standards and draft standards from the American National Standards Institute (ANSI) and the Institute of Electrical and Electronic Engineers (IEEE):

- ANSI X3.159-1989 (ANSI C)
- IEEE Std 1003.1b-1993 (POSIX)
- IEEE Draft Standard 1003.1c/D9
- IEEE Draft Standard 1003.12/D1
- IEEE Draft Standard 1003.13/D5

Contents

About This Book	xvii
Who Should Use This Book	xvii
How To Use This Book	xviii
Conventions Used In This Book.....	xix
Numeric Notation and Input Conventions	xix
Highlighting Conventions	xx
Syntax Diagram Conventions.....	xx
Character Sets Used In This Book.....	xxi
Decimal Digits	xxi
Graphics Characters	xxi
Hexadecimal Digits	xxi
Lowercase Letters.....	xxi
Uppercase Letters.....	xxi
Portable File Name Characters.....	xxii
Where to Find More Information	xxii
Product Support	xxii
Related IBM Publications.....	xxii
Related non-IBM Publications.....	xxiv
Standards.....	xxiv
Benchmark Papers.....	xxiv
Chapter 1. About the OS Open Operating System.....	1-1
File Naming Changes	1-1
Embedded System Software Development	1-2
Programming Languages.....	1-2
Host System Requirements	1-2
Target System Requirements	1-3
Features.....	1-3
Functions	1-4
Performance Optimization	1-6
Chapter 2. OS Open Support for Industry Standards.....	2-1
The ANSI C Standard	2-1
OS Open Support for the ANSI C Standard.....	2-2
The POSIX Standard and Draft Standards	2-2
OS Open Support for the POSIX Standard and Draft Standards.....	2-3
Chapter 3. Understanding the OS Open Architecture	3-1
The Real-time Executive.....	3-2
System Management	3-3

Thread Management	3-4
Storage Management	3-4
Signals	3-5
Clocks and Timers	3-6
Interrupt and Fault Handling	3-6
Message Queues	3-6
Semaphores	3-7
Trace Buffer Support	3-7
Miscellaneous Services	3-8
The ANSI C Libraries	3-9
Character Handling	3-9
Nonlocal Jumps	3-9
Variable Arguments	3-10
Input/Output	3-10
General Programming Utilities	3-11
Storage Allocation	3-11
String Handling	3-11
Time and Date	3-12
Device and File Support Library	3-13
Low-Level POSIX I/O Support	3-13
Supporting Regular Files on a Logical File System	3-14
Integrated Device Drivers	3-14
Storage Locking	3-15
Terminal Support Library	3-15
Ring Buffer Library	3-16
Queue Library	3-16
DOS File System Support Library	3-16
RAM Disk Library	3-16
PCMCIA Support Libraries	3-17
Network Support Library	3-17
TCP/IP Protocol Support Library	3-17
NFS Client Support Library	3-17
Run Library	3-17
RPC/XDR Protocol Support Library	3-18
File Transfer Protocol Support Library	3-18
Telnet Support Libraries	3-18
6xx Alignment Exception Support Library	3-18
Debugging Support Library	3-19
OpenShell	3-19
Remote Source Level Debugger Library	3-19

Kernel Abstract Data Types Library	3-20
Symbol Support Library	3-20
Dynamic Loader Library.....	3-20
XL C Compiler Support Library (xcoff only)	3-20
C++ Runtime Support (C Set ++ Support) Library	3-21
The Floating Point Emulation Library.....	3-21
Rate Monotonic Scheduling (RMS) Library.....	3-21
Chapter 4. Configuring the OS Open Operating System.....	4-1
Kernel Configuration Block Fields.....	4-1
The Panic Function	4-2
Chapter 5. Using OS Open Features in Hard Real-Time Systems.....	5-1
RMS Analysis.....	5-1
Deterministic Execution	5-2
Bounded Execution.....	5-2
Chapter 6. Understanding OS Open Threads.....	6-1
Working with OS Open Threads	6-1
Creating Threads	6-1
Terminating Threads.....	6-2
Canceling Threads	6-2
Working with the Cancelability State of a Thread	6-2
Using Cleanup Routines	6-3
Using Cancel Safe Functions.....	6-3
Scheduling Threads	6-4
Serializing Threads	6-5
Using Mutexes	6-6
Handling Priority Inversions	6-6
Using Mutex Attributes Objects.....	6-7
Synchronizing Threads	6-7
Understanding Condition Variables.....	6-8
Using Condition Variables.....	6-8
Using Semaphores.....	6-9
Using Thread Attributes Objects	6-9
Working with Thread-Specific Data.....	6-11
Chapter 7. Writing OS Open Interrupt and Fault Handlers.....	7-1
Using OS Open Interrupt Handlers	7-1
Running Interrupt Handlers.....	7-3
Interrupt Handler Safe Functions.....	7-4
Chapter 8. Writing OS Open Device Drivers.....	8-1

Device Driver Types.....	8-1
Implementing OS Open Device Drivers	8-2
Installing and Initializing Device Drivers.....	8-2
Uninstalling a Device.....	8-3
Understanding File Blocks	8-3
Writing Character Device Driver Functions	8-5
Device-Specific Open.....	8-5
Device-Specific Close	8-5
Device-Specific Read	8-6
Device-Specific Write	8-6
Device-Specific I/O Control	8-6
Device-Specific Select.....	8-7
Writing Block Device Driver Functions	8-7
Device-Specific Open.....	8-8
Device-Specific Close	8-8
Device-Specific I/O Control	8-9
Device-Specific Strategy	8-9
Performance Considerations.....	8-9
Chapter 9. Writing Logical File Systems	9-1
Implementing OS Open Logical File Systems.....	9-1
Installing Logical File Systems	9-1
Writing Logical File System Functions	9-2
Driver-Specific Open	9-2
Driver-Specific Close.....	9-2
Driver-Specific Read	9-3
Driver-Specific Write.....	9-3
Driver-Specific I/O Control.....	9-3
Driver-Specific Select.....	9-13
Chapter 10. Using the DOS Logical File System.....	10-1
Installing the DOS File System Driver.....	10-1
Installing Devices for DOS File Systems.....	10-1
Uninstalling Devices for DOS File Systems	10-2
Opening DOS Files	10-2
Reading and Writing DOS Files	10-3
DOS File System I/O Control Commands.....	10-3
Interfacing a DOS File System to a Block Device Driver	10-4
Using the OS Open RAM Disk.....	10-5
Installing the RAM Disk Device Driver.....	10-5
Installing Devices for RAM Disk File Systems.....	10-5

Opening RAM Disk Files	10-7
Reading and Writing RAM Disk Files	10-7
Using rambuild to Preload the RAM Disk.....	10-7
Using OS Open SCSI Support.....	10-8
SCSI Device Driver	10-8
Device Driver Installation	10-8
Device Installation	10-9
Uninstalling a SCSI Device	10-9
Opening a SCSI Device	10-9
Closing a SCSI Device.....	10-10
Reading and Writing.....	10-10
I/O Control.....	10-11
SCSI Adapter Device Drivers.....	10-11
The scsiadpt SCSI Adapter Device Driver	10-12
Device Driver Installation	10-12
Device Installation	10-12
Uninstalling a scsiadpt Device	10-13
Opening a scsiadpt Device	10-14
Closing a scsiadpt Device	10-14
Required I/O Control Commands.....	10-15
Using SCSI Devices with the DOS File System.....	10-18
Chapter 11. OS Open PCMCIA Support	11-1
Introduction to PCMCIA	11-1
PC Cards.....	11-2
Socket Controllers.....	11-3
Socket Services	11-4
Card Services and Enablers	11-5
Card Services/Enabler Software Layer.....	11-5
Initializing the Card Services/Enabler Software Layer	11-6
Installing and Querying Shared Interrupts.....	11-8
Allocating Socket Services Windows	11-8
Determining the Insertion State of a Card.....	11-9
Advanced Card Services/Enabler Card Matching.....	11-9
Socket Services	11-12
Initializing the Socket Services Library.....	11-12
Socket Services Functions.....	11-13
Non-Specific Function	11-13
Adapter Functions	11-14
Socket Functions.....	11-16
Window Functions.....	11-18

PCMCIA ATA/IDE Device Driver	11-20
Device Driver Installation.....	11-21
Device Installation	11-21
Uninstalling a PCMCIA ATA/IDE Device.....	11-22
Opening a PCMCIA ATA/IDE Device.....	11-22
Closing a PCMCIA ATA/IDE Device	11-22
Reading and Writing.....	11-22
I/O Control.....	11-23
Using the PCMCIA ATA/IDE Device Enabler.....	11-24
Putting it All Together.....	11-25
Chapter 12. OS Open Networking Features	12-1
Introducing the TCP/IP Library.....	12-1
Corequisite Libraries	12-3
Introducing the Network Library	12-3
Defining the _Tcpi_services Array.....	12-4
Defining the _Tcpi_resolv Array	12-5
Defining the _Tcpi_protocols Array	12-5
Defining the _Tcpi_hosts Array	12-5
Defining the _Tcpi_networks Array	12-6
Corequisite Libraries	12-6
Introducing the NFS Client Library	12-6
Defining the _Rpc_rpc Array	12-7
Chapter 13. Writing OS Open Network Interface Drivers	13-1
Introducing Network Interface Concepts	13-1
Working with Memory Buffers	13-1
Serializing Network Interface Control Structures.....	13-1
Scheduling Network Software Interrupts.....	13-1
Reading an Annotated Network Interface Driver.....	13-2
Attaching the Network Interface Driver.....	13-2
Sending a Packet	13-2
Receiving a Packet.....	13-4
I/O Control Processing	13-6
ARP Processing	13-7
Chapter 14. Using File Transfer Protocols (FTP and TFTP).....	14-1
Chapter 15. Using Telnet.....	15-1
Chapter 16. OS Open with Virtual Memory.....	16-1
OS Open with Virtual Memory Features	16-2
Read-Only Code.....	16-2

Stack Protection	16-2
Improved Memory Manager	16-3
Protected I/O Areas	16-3
Private Allocated Memory	16-3
Protection Options for Loaded Objects	16-3
Thread Groups	16-4
The Initial Thread Group	16-4
Creating and Destroying Thread Groups	16-4
Thread Group Keys	16-4
Virtual Memory Manager	16-7
Create and Destroy virtual address spaces	16-7
Changing attributes of virtual memory regions	16-8
Querying a virtual memory region	16-8
Adding and removing real mappings to a virtual address space	16-8
Removing real pages from a virtual mapping	16-9
Virtual memory statistics	16-9
Moving information between virtual address regions	16-9
Obtaining the real address of a specified virtual address	16-9
Virtual Memory Manager Fault handling	16-9
Miscellaneous System Issues	16-10
Symbol Table Management	16-11
Supervisor function privilege	16-11
Loader Support	16-11
First level Interrupt Handlers	16-11
Device Drivers	16-12
Critical Exception Processing	16-12
Application Thread Group Access to Kernel Thread Group	16-12
Chapter 17. Developing OS Open Applications	17-1
Introducing Host Development Tools	17-1
Creating OS Open XCOFF Object Files	17-2
Compiling C Source Programs (XCOFF Object)	17-2
Assembling Assembler Language Source Programs (XCOFF Object)	17-2
Working with XCOFF Object Files	17-3
Working with Archive Files	17-3
Linking XCOFF Object Files	17-4
Creating OS Open ELF Object Files	17-6
Compiling C Source Programs (ELF Object)	17-6
Assembling Assembler Language Source Programs (ELF Object)	17-7
Working with ELF Object Files	17-7
Working with Archive Files	17-7

Linking ELF Object Files	17-8
Working with Makefiles	17-9
Working with the OS Open Sample Programs.....	17-10
Introducing the Sample Programs.....	17-10
Introducing the Source Programs.....	17-11
Introducing the Sample Kernel Configuration Block.....	17-12
Introducing the Sample panic() Module.....	17-12
Building and Running the OS Open Sample Programs	17-12
Compiling the Sample Programs.....	17-13
Loading the OS Open Image	17-13
Booting the Sample Images over a TCP/IP Connection.	17-13
Building the Diskettes.....	17-14
Building a Loadable Module	17-14
Chapter 18. Using OpenShell.....	18-1
Preparing OS Open for OpenShell.....	18-1
Starting OpenShell	18-2
Entering and Working with OpenShell Commands	18-2
Editing Commands	18-2
Using Command History	18-3
Redirecting Input and Output	18-4
Using the Special File /more	18-4
Changing the Command Line Prompt	18-5
Using the OpenShell C Interpreter	18-5
Understanding C Interpreter Restrictions and Differences.....	18-6
Constants	18-6
String Literals	18-6
Identifiers.....	18-7
Expression Operators.....	18-7
Unary Operators.....	18-7
Binary Operators	18-7
Storage Class Specifiers	18-8
Type Specifiers.....	18-8
Structure Specifier	18-9
Control Specifiers	18-9
User-Defined Functions	18-10
Shell Commands.....	18-11
C Interpreter Examples	18-11
Defining and Printing an Integer Variable.....	18-11
Declaring a Character Pointer.....	18-12
Defining a Character Variable	18-12

Defining a Function	18-12
Working with Shell Symbols.....	18-13
Examples	18-14
Displaying Currently Defined Kernel Objects	18-14
Displaying Thread Information	18-15
Disassembling Instructions	18-15
Displaying Memory.....	18-16
Testing Functions.....	18-16
Printing Variable Information.....	18-16
Using OpenShell Debugging Features	18-16
Creating a Debugging Environment.....	18-17
OpenShell Variables	18-17
Using Local Debugger Commands in OpenShell	18-18
Disassembling Object Code.....	18-18
Displaying Memory Contents	18-19
Setting Breakpoints	18-19
Clearing Breakpoints.....	18-20
Listing Breakpoint Status	18-21
Continuing after a Breakpoint.....	18-21
Instruction Step/Trace.....	18-21
Performing Stack Trace-Backs	18-22
Debugging a Thread	18-23
Taking Kernel Trace Snapshots.....	18-24
Appendix A. OS Open Functions Listed by Library.....	A-1
Functions in rtxLib.a.....	A-1
System Management	A-1
Thread Management.....	A-1
Heap Management.....	A-2
Buffer Pool Management	A-2
Signal Handling	A-2
Clocks and Timers	A-2
Interrupt and Fault Handling.....	A-2
Message Queues	A-2
Semaphores.....	A-2
Trace Buffer Support.....	A-2
Miscellaneous Services.....	A-2
Functions in cLib.a	A-2
Functions in mathLib.a	A-3
Functions in fsLib.a	A-4
Functions in rngLib.a.....	A-4

Functions in queLib.a	A-4
Functions in devLib.a	A-4
Functions in ttyLib.a	A-5
Functions in netLib.a	A-5
Functions in tcpipLib.a	A-5
Functions in rpcLib.a	A-6
Functions in ftpLib.a	A-7
Functions in dbLib.a	A-7
Functions in kadtLib .a	A-7
Functions in symLib.a	A-7
Functions in ldrLib.a	A-7
Functions in mwdctor.o and mwdctorl.o	A-8
Functions in rmsLib.a	A-8
Functions in nfsLib.a	A-8
Functions in runLib.a	A-8
Functions in ramdLib.a	A-8
Functions in rsldLib.a	A-8
Functions in tftp.o	A-8
Functions in tnetdLib.a	A-8
Functions in alignLib.a	A-8
Functions in cpprtLib.a	A-8
Functions in csLib.a	A-8
Functions in pataLib.a	A-8
Functions in ssLib.a	A-8
Appendix B. Sample Device Drivers	B-1
Sample Asynchronous Device Driver	B-1
Sample Diskette Device Driver	B-20
Sample SCSI Device Driver	B-39
Include File for Skeleton SCSI Adapter Device Driver	B-39
Sample Skeleton SCSI Adapter Device Driver	B-39

Tables

Table 1-1.	Registers Saved (PowerPC 601).....	1-6
Table 2-1.	Supported ANSI and IEEE Standards and Draft Standards.....	2-1
Table 3-1.	OS Open Optional Libraries	3-1
Table 5-1.	Real-Time Executive Functions Having Unbounded Execution Times	5-3
Table 6-1.	Mutex Attributes Summary	6-7
Table 6-2.	Thread Attributes Summary.....	6-10
Table 7-1.	PowerPC Interrupt Types	7-2
Table 10-1.	I/O Control Commands for DOS File Systems	10-3
Table 12-1.	I/O Control Commands for Socket File Descriptors.....	12-2
Table 18-1.	Command Editor Control Mode Commands.....	18-3

About This Book

This book describes the IBM™ OS Open™ real-time operating system and its development environment for embedded systems based on PowerPC™ processors.

The OS Open operating system is a scalable real-time operating system and development environment. No other real-time operating system exploits the power of the IBM PowerPC Architecture™ so effectively.

To ensure that applications can be ported across a wide variety of platforms, the OS Open operating system supports the Portable Operating System Interface (POSIX) programming standards as well as the American National Standards Institute (ANSI) C programming standards.

The OS Open operating system features:

- Hard real-time support, including deterministic execution, priority inheritance protocols, and priority ceiling protocols
- Board support packages for plug-and-play operation of popular board-level products
- Support for existing American National Standards Institute (ANSI) C and emerging POSIX standards
- Open network interfaces to support embedded systems in heterogeneous environments
- Scalable implementations to meet the requirements and constraints of a variety of embedded systems

Who Should Use This Book

This book is for:

- Programmers who use the OS Open operating system to develop applications for embedded systems
- System designers who plan to move the OS Open operating system to a custom hardware platform
- Hardware designers who have a need to understand the OS Open operating environment

Users should understand:

- Their host systems' functions, architecture, and features
- The PowerPC Architecture and assembler programming
- C programming

How To Use This Book

This book is divided into three parts to meet the needs of the various types of users.

Part 1, “Introducing the OS Open Operating System,” contains general overview material and should be read by all OS Open users. This part contains the following chapters:

- Chapter 1, “About the OS Open Operating System,” describes major OS Open functions and features and lists host hardware and software requirements.
- Chapter 2, “OS Open Support for Industry Standards,” lists and briefly describes the major supported operating system, programming, and networking standards.
- Chapter 3, “Understanding the OS Open Architecture,” describes the OS Open components.

Part 2, “Configuring the OS Open Operating System,” is primarily of interest to those who install OS Open. This part contains the following chapter:

- Chapter 4, “Configuring the OS Open Operating System,” describes how to configure the OS Open operating system on the target system.

Part 3, “Writing OS Open Applications,” should be read by all OS Open application developers, including those who extend the functionality of the operating system using additional device drivers and network interfaces. Part 3 contains the following chapters:

- Chapter 5, “Using OS Open Features in Hard Real-Time Systems,” describes the programming features provided by the OS Open operating system to support hard real-time applications.
- Chapter 6, “Understanding OS Open Threads,” describes OS Open threads, thread control features, and interthread communication methods.
- Chapter 7, “Writing OS Open Interrupt and Fault Handlers,” describes interrupt handlers and their use in the OS Open operating system, and provides examples of interrupt handler code.
- Chapter 8, “Writing OS Open Device Drivers,” describes device drivers and their use in the OS Open operating system, and provides examples of device driver code.
- Chapter 9, “Writing Logical File Systems,” describes file systems and their use in the OS Open operating system, and provides examples of file system code.
- Chapter 10, “Using the DOS Logical File System,” describes the installation and use of the DOS logical file system.

- Chapter 11, "OS Open PCMCIA Support," describes the PCMCIA Card Services and Socket Services provided by the OS Open operating system.
- Chapter 12, "OS Open Networking Features," describes the networking support provided by the OS Open operating system.
- Chapter 13, "Writing OS Open Network Interface Drivers," describes network interfaces and their use in the OS Open operating system, and provides examples of network interface code.
- Chapter 14, "Using File Transfer Protocols (FTP and TFTP)," describes the network utilities that allow files to be transferred between systems.
- Chapter 15, "Using Telnet," describes the network utility that provides interactive terminal services between systems.
- Chapter 16, "OS Open with Virtual Memory," describes the enhanced functionality of OS Open with Virtual Memory and the system concepts relevant to the OS Open VM environment.
- Chapter 17, "Developing OS Open Applications," provides an overview of application development for a particular OS Open target platform.
- Chapter 18, "Using OpenShell," describes OpenShell and its use in testing and debugging OS Open application programs.
- Appendix A, "OS Open Functions Listed by Library," lists the OS Open functions by library.
- Appendix B, "Sample Device Drivers," lists sample character and block device drivers.

An index follows the appendices.

Conventions Used In This Book

This book follows numeric and highlighting notation conventions based on those used in the RISC System/6000™ and Advanced Interactive Executive (AIX™) publications.

Numeric Notation and Input Conventions

In general, numbers are used exactly as shown. Unless noted otherwise, all numbers are in decimal, and, if entered as part of a command, are entered without format information.

In text, hexadecimal numbers are preceded by X followed by the number enclosed in single quotes, for example:

X'1A7'

In code examples and messages, hexadecimal numbers are preceded by "0x" or "x," and may be followed by the number enclosed in single quotes, for example:

0x1a7, x'd1a7'

In text, the hexadecimal digits A through F appear in uppercase. In programs, these digits are typically entered in lowercase.

Highlighting Conventions

This book uses the following highlighting conventions:

- The names of invariant objects known to the OS Open operating system appear in bold type. In some text, however, such as in lists, no special typographic treatment is used. Examples of such objects include:





- Function and macro names
- Data types and structures
- Constants and flags

Names of objects known to the OS Open operating system must be entered exactly as shown.

- Variable names supplied by user programs appear in italic type. In some text, however, such as in lists, no special typographic treatment is used. Examples of these objects include arguments and other parameters.
- No highlighting appears in code examples.

Syntax Diagram Conventions

Throughout this book, diagrams illustrate the syntax for string formats and commands. The following list shows how to read these diagrams:

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.
- A  symbol begins a diagram.
- A  symbol indicates continuation of a diagram on the next line.
- A  symbol indicates continuation of a diagram from the previous line.
- A  symbol terminates a diagram.
- Keywords are in regular type, and variables are in italics. Keywords must be typed exactly as shown.
- Keywords or variables on the main path of a diagram are required.

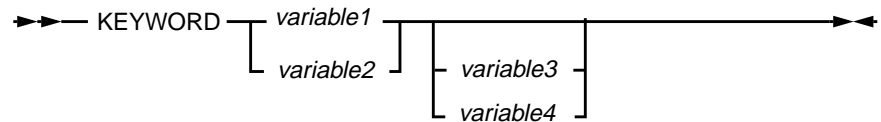
 keyword — *variable1* — *variable2* 

- Keywords or variables shown on branches below the main path are optional.

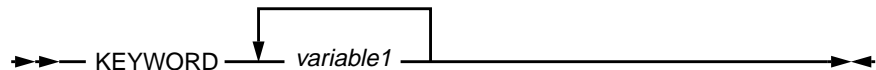
 keyword 

- Keywords or variables can appear in a stack, indicating that only one item in a stack can be chosen. If an item in a stack is on the main path, you must choose an item from the stack. If all items in a stack are below the main path, you may choose an item from the stack.

For example, in the following syntax diagram, you must choose either *variable1* or *variable2*. However, because *variable3* and *variable4* are below the main path, neither is required.



- A repeat separator is a returning arrow that surrounds a syntax element or group and shows that the element or group can be repeated.



Character Sets Used In This Book

Decimal Digits

The following 10 characters form the set of decimal digits:

0 1 2 3 4 5 6 7 8 9

Graphics Characters

The following 29 characters form the set of graphics characters:

! " # % & ' () * + , - . / : ; < = > ? [\] ^ _ { | } ~

Hexadecimal Digits

The following 22 characters form the set of hexadecimal digits:

0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f

Lowercase Letters

The following 26 characters form the set of lowercase letters:

a b c d e f g h i j k l m n o p q r s t u v w x y z

Uppercase Letters

The following 26 characters form the set of uppercase letters:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Portable File Name Characters

The set of characters from which portable file names can be constructed contains:

- All uppercase letters
- All lowercase letters
- All decimal digits
- The period (.)
- The underscore (_)
- The hyphen (-)

The slash (/) can also be used in a portable path name.

Where to Find More Information

Sources of information about or related to the OS Open operating system are listed under the following headings.

Product Support

If you have questions or experience problems while using the OS Open operating system, call the PowerPC Embedded Systems Solutions Center at (919) 254-1810. Please send any comments about this document to the following Internet address: ppc400pubs@vnet.ibm.com

Related IBM Publications

This book refers to the following publications. To order, contact your IBM Microelectronics Division sales representative or call the PowerPC Embedded Systems Solutions Center at (919) 254-1810.

OS Open Library

The following list includes all the books in the OS Open library:

IBM OS Open User's Guide, 92G6897

IBM OS Open Programmer's Reference, Volume 1, 92G6911

IBM OS Open Programmer's Reference, Volume 2, 92G6912

Using OS Open on the 403EVB, 92G6932

Using OS Open on Sandalfort, 92G6933

User's Manuals

PPC403GA Embedded Controller User's Manual, 13H6960

PPC403GB Embedded Controller User's Manual, 13H6985

PowerPC 601 RISC Microprocessor User's Manual, 52G7484

PowerPC 603 RISC Microprocessor User's Manual, MPR603UMU-01

PowerPC 604 RISC Microprocessor User's Manual, MPR604UMU-01

RISCWatch™ User's Guides

RISCWatch 400 Debugger User's Guide, 13H6964

RISCWatch 601 User's Guide and Reference, 13H6969

RISCWatch 603 User's Guide and Reference, 13H6967

RISCWatch 604 User's Guide and Reference, 13H6968

RISC System/6000 Publications

This book refers to the following RISC System/6000 publications:

IBM RISC System/6000: POWERstation and POWERserver Hardware Technical Information General Architectures, SA23-2643

AIX Publications

This book refers to the following AIX publications. The words “IBM AIX Version 3.2 for RISC System/6000” are actually part of the title of each book; however, in all references to these books, those words are omitted.

Assembler Language Reference, SC23-2197

Commands Reference, Volume 1, SC23-2376

Commands Reference, Volume 2, SC23-2366

Commands Reference, Volume 3, SC23-2367

Commands Reference, Volume 4, SC23-2393

Editing Concepts and Procedures, GC23-2212

Files Reference, GC23-2200

XL C Compiler/6000 Publications

XL C Language Reference, SC23-1260

XL C User's Guide, SC23-1259

IBM High C/C++ Publications

The following list includes the books in the IBM High C/C++ library:

IBM High C/C++ Programmer's Guide for PowerPC, 92G6920

IBM High C/C++ Language Reference for PowerPC, 92G6923

IBM ELF Assembler User's Guide for PowerPC, 92G6921

IBM ELF Linker User's Guide for PowerPC, 92G6922

PowerPC Embedded Application Binary Interface

To receive a copy of the EABI specification, send an email to eabi@goth.sps.mot.com, and include the word “eabi” or “EABI” in the subject line. The EABI PostScript file will be sent to you.

Related non-IBM Publications

This book refers to the following non-IBM publication:

Comer, Douglas E., and Stevens, David L., *Internetworking with TCP/IP*, Volumes 1–3. Prentice-Hall, NJ, 1991.

Standards

This book refers to the following standards:

American National Standard for Information Systems - Programming Language - C, ANSI X3.159, 1989, American National Standards Institute.

Information Technology - Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) [C Language], IEEE Std. 1003.1b: 1993, Institute of Electrical and Electronics Engineers, Inc.

Draft Standard for Information Technology - Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) - Amendment 2: Threads Extension [C Language], P1003.1c Draft 9, Institute of Electrical and Electronics Engineers, Inc.

Draft Standard for Information Technology - Standardized Application Environment Profile - POSIX Realtime Application Support (AEP), P1003.13 Draft 5, Institute of Electrical and Electronics Engineers, Inc.

Benchmark Papers

This book refers to the following benchmark papers:

Kar, R. P. “Implementing the Rhealstone Benchmark.” *Dr. Dobb's Journal*, Vol. 15 No. 4, April 1990, pp. 46–52.

Liu, C. L. and Layland, J. W. “Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment.” *J.ACM*, Vol. 20, No. 1, January 1973, pp. 46–61.

Sha, L., and Goodenough, J. B. “Real-Time Scheduling Theory and ADA.” *Computer*, Vol. 23, No. 4, pp. 53–62.

Part 1. Introducing the OS Open Operating System

Chapter 1. About the OS Open Operating System

The IBM OS Open Real-time Operating System provides a scalable run-time and development environment for embedded systems running on PowerPC processors.

OS Open supports important industry programming and operating system standards as well as advanced real-time programming tools and techniques. This support provides a solid platform for the efficient development of powerful real-time applications.

File Naming Changes

To support OS Open development under DOS, OS Open header files and libraries that had more than eight characters in their base names (Release 1.3 and earlier) were renamed in Release 1.4.

One-line stub files having the Release 1.3 and earlier file names are supplied to support existing code. For example, **kerneladtLib.h** was renamed to **kadtLib.h**; the stub file **kerneladtLib.h** consists of the line:

```
#include <kadtLib.h>
```

The documentation reflects the new names.

The following files have changed (all of the files may not exist on all platforms):

Old name	New name
biosenetLib.h	benetLib.h
kerneladtLib.h	kadtLib.h
ramdskLib.h	ramdLib.h
semaphore.h	semaphor.h
machine/machparam.h	machine/machparm.h
netinet/tcp_debug.h	netinet/tcp_debu.h
netinet/tcp_timer.h	netinet/tcp_time.h
rpc/auth_unix.h	rpc/auth_uni.h
rpc/pmap_clnt.h	rpc/pmap_cln.h
rpc/pmap_prot.h	rpc/pmap_pro.h
sys/devDriver.h	sys/devDrivr.h
sys/socketvar.h	sys/socketva.h
sys/telnetdLib.h	sys/tnetdLib.h

Libraries with names of more than eight characters were also renamed:

Old name	New name
biosenetLib.a	benetLib.a
kerneladtLib.a	kadtLib.a
ramdiskLib.a	ramdLib.a
telnetdLib.a	tnetdLib.a

You should update your Makefiles to reflect these changes.

Embedded System Software Development

Embedded systems are typically developed in a cross-development environment consisting of host computers and target systems. The host computers provide software and project management tools for embedded system application developers. The developers are not restricted to the limited computing resources typically available on the target embedded system.

Developers write, compile, and debug embedded system application programs on the host computers. When appropriate, the application programs are loaded on the target embedded system, where they can run and be tested in the target operating environment.

Embedded system development is an iterative process; the application programs are refined on the host computers and tested on the target system until the programs meet their requirements. Eventually, the application programs are shipped as part of an embedded system.

Programming Languages

OS Open application programs are typically written in C and assembler language.

Host System Requirements

The OS Open cross-development environment supports either eXtended Common Object File Format (XCOFF) or Extended Link Format (ELF) object file format, depending on specific version and release levels of the OS Open libraries. IBM and other vendors provide numerous optional software development tools which can be used in the OS Open development environment:

- Computer-aided software engineering (CASE)
- Structured analysis and design
- Program understanding
- Code management and version control

System requirements for XCOFF and ELF versions on their respective platforms are described below.

XCOFF

The XCOFF-based OS Open cross-development environment is hosted on IBM RISC System/6000 (RS/6000) computers running the IBM AIX operating system, Version 3.2.5 or higher. AIX tools used to develop XCOFF-based OS Open applications include:

- XL C compiler, for C programs
- AIX assembler, for AIX assembler language programs
- nimgbld, to build an OS Open load image
- AIX linker/binder, to build OS Open applications for a target system

ELF

The ELF-based OS Open cross-development environment can be hosted on the following systems:

- IBM RISC System/6000 (RS/6000) computers running the IBM AIX operating system, Version 3.2.5 or higher
- Sun SPARCstation running SunOS 4.1.3 or Solaris 2.3
- IBM PC or compatible, running Windows 3.1

Tools used to develop ELF-based OS Open applications include:

- High C/C++ compiler, for C programs
- asppc assembler, for assembler language programs
- eimgbld, to build an OS Open load image
- ELF linker/binder, to build OS Open applications for a target system

Target System Requirements

The OS Open operating system is appropriate for virtually any embedded application running on a PowerPC processor. For supported board-level products, OS Open provides a turnkey environment for application development. Assistance is available to port OS Open to custom board products.

Features

The OS Open operating system is a complete, flexible platform for embedded applications. The OS Open operating system provides the following features to enable application designers to concentrate on their programs:

- Open programming interfaces
- Support for POSIX and ANSI C standards provides portability of application software across many platforms, which enables software developed for other platforms to be reused.

- Open network interfaces

OS Open libraries support standard networking interfaces, including TCP/IP, Telnet, file transfer protocol (FTP), sockets, and others. This support allows an embedded system to participate in heterogeneous computing environments.

- Support for the PowerPC Architecture

The OS Open operating system is fully optimized for PowerPC processors.

- Flexible, modular embedded computing solutions

OS Open configurations are as varied as the systems in which they are used. Only the libraries that provide the functions needed in a specific environment need to be included in the application.

- Support for hard real-time, high-throughput applications

The OS Open operating system provides the mechanisms necessary to achieve high throughput and deterministic execution, including constant time implementations of most major operating system functions, priority inheritance protocols, and priority ceiling protocols.

- Board-support libraries for turnkey operations

Application designers are primarily interested in getting the application running on the hardware and environment that will be used in the embedded application. With the goal of quick, confidence-building installation and use, the OS Open operating system provides sample code, device drivers, and other support for board-level products featuring PowerPC processors.

Functions

OS Open provides the following major classes of functions for the embedded programming environment:

- Thread management

The thread, as defined by POSIX standards, is the OS Open execution context unit. OS Open functions to create threads having various scheduling and execution attributes are provided. Serialization and synchronization primitives are provided to manage the execution environment. The OS Open operating system also provides functions to associate data with specific threads.

- Storage management

OS Open functions provide variable storage block allocation using a heap. These functions extend the heap, query heap usage, and allocate storage to meet alignment constraints. OS Open also provides an independent storage management mechanism to allocate fixed blocks of storage in constant time.

- Interrupt and fault support

OS Open functions can attach user-written code to any processor exception or interrupt. Most OS Open functions can be used in these interrupt handlers, except for those that suspend execution or are valid only in the context of a running thread. When underlying hardware provides support, OS Open platform-specific libraries provide additional functions to attach user-written code to platform-specific external interrupts.

- Clock and timer management

OS Open functions provide time-of-day clock support and the ability to create, use, and destroy timers. These timers can be one-time or periodic.

- Device support

OS Open functions support the installation of user-written device drivers to provide character special files, block special files, and logical file systems. Low-level POSIX I/O (read, write) as well as ANSI C stream (fget, fput) functions are provided for device and regular file access.

- ANSI C library support

The OS Open operating system provides a comprehensive set of ANSI C functions, providing support for string manipulation, memory management, string-to-number conversion, input/output, nonlocal jumps, and variable arguments.

- Pseudo device driver support

The OS Open operating system provides several functions that are installed and managed like device drivers, but they do not manipulate actual hardware nor do they have platform or device dependencies.

These pseudo drivers include:

- A tty driver to add additional functions to a character serial device
- A shared memory driver to implement shared memory functions
- A null device driver to provide **/dev/null** support
- A logical file system to manage a block device driver as a PC DOS 5.0 file system.
- Networking support

The OS Open operating system provides functions that create and manage TCP/IP sockets. Network interface functions for Token Ring, Ethernet, and Serial Line IP (SLIP) are also provided. With the TCP/IP protocol stack and network interfaces, additional functions are provided that implement several popular networking utilities, such as ping, ifconfig, ftp, and telnet.

- Debug functions and kernel abstract data types

The OS Open operating system provides functions that set, clear, and query breakpoints. The OS Open operating system features an internal circular trace buffer for operating system and user events. Functions are also provided to dump kernel data objects in a readable form.

Performance Optimization

OS Open employs a number of techniques to minimize delays introduced by system calls. These techniques include:

- Optimized abstract data types (ADTs)

Many ADTs have two source language versions. The C versions provide portability across processors. Assembler language versions are optimized for a target processor.

The assembler language ADTs:

- Use machine instructions not generated by the C compiler
- Achieve zero-cycle branching for “most travelled” paths
- Avoid duplicate computation of complex basing expressions

- Minimal thread state information

OS Open threads carry a minimal amount of state information. Only enough information to describe the processor state is routinely put into the threads. These extra-light threads allow for optimally fast context switching.

- Selective state saving

OS Open’s real-time executive could save the entire state of the currently executing thread on each entry to the real-time executive. This approach would simplify the design of the real-time executive, but because of the large number of registers in the PowerPC Architecture, this approach would have a strong negative impact on performance.

The OS Open operating system saves the minimal number of registers on each entry to the real-time executive. As shown in Table 1-1, on a non-blocking system call, only three registers are saved on entry to the real-time executive. On a blocking system call, 25 registers are saved; on an external interrupt, 20 registers are saved.

Table 1-1. Registers Saved (PowerPC 601)

Entry Type	General Purpose Registers Saved	Special Purpose Registers Saved
Non-Blocking System Call	1	2
Blocking System Call	22	3
External Interrupt	13	7

Chapter 2. OS Open Support for Industry Standards

The OS Open application programming interface (API) supports the following standards and draft standards from the American National Standards Institute (ANSI) and the Institute of Electrical and Electronic Engineers (IEEE).

Table 2-1 shows the major standards and draft standards supported by OS Open. The rightmost column in the table gives the abbreviated title used in the OS Open publications to refer to the standard or draft standard.

Table 2-1. Supported ANSI and IEEE Standards and Draft Standards

Common Name	Standard Title	Referenced Title
ANSI C	American National Standard for Information Systems, X3.159-1989, <i>Programming Language – C</i>	ANSI C
POSIX 1	<i>Information Technology–Portable Operating System Interface (POSIX)–Part 1: System Application Programming Interface (API) [C Language]</i>	IEEE Std 1003.1b: 1993
POSIX 1c/D9	<i>Draft Standard for Information Technology–Portable Operating System Interface (POSIX)–Part 1: System Application Programming Interface (API)–Amendment 2: Threads Extension [C Language]</i>	IEEE Draft Std 1003.1c/D9
POSIX 13/D5	<i>Draft Standard for Information Technology–Standardized Application Environment Profile–POSIX Realtime Application Support (AEP)</i>	IEEE Draft Std 1003.13/D5

The ANSI C Standard

The ANSI C standard specifies the form and establishes the interpretation of programs written in the C programming language. The ANSI C standard provides a clear, machine-independent language definition and supports and promotes the development of portable C programs.

The ANSI C standard specifies the:

- Representation of C programs
- Syntax and constraints of the C language
- Semantic rules for interpreting C programs.
- Representation of input and output data processed by C programs
- Restrictions and limits imposed on conforming C implementations

The ANSI C standard does not specify:

- How C programs are transformed for use by a system
- How C programs are invoked for use by a system
- How a C program handles input and output
- The size or complexity of a C program

OS Open Support for the ANSI C Standard

OS Open implements the ANSI C Standard.

The POSIX Standard and Draft Standards

POSIX (an acronym for Portable Operating System Interface, with an X added to indicate an association with UNIX® operating systems) names a family of standards sponsored by the IEEE Computer Society Technical Committee on Operating Systems and Application Environments:

- IEEE Std 1003.1b:1993

This international standard, often called POSIX 1, defines a standard operating system interface and environment to support application portability at the source-code level.

This standard defines and specifies terminology, system service interfaces and subroutines, including system services for the C programming language, and interface issues such as portability and error handling.

The system calls defined in IEEE Std 1003.1b:1993 comprise a portable operating system interface. Supported system calls are provided as OS Open functions, such as **open()**, **close()**, **read()**, and **write()**.

As amended, this standard defines interfaces and functionality appropriate for real-time systems. Specific functional areas in the standard include:

- Semaphores
- Process memory locking
- Memory-mapped files and shared memory
- Priority scheduling
- Real-time signal extensions
- Timers
- Interprocess communication
- Synchronized input and output (I/O)
- Asynchronous I/O
- Real-time files

- IEEE Draft Std 1003.1c/D9

This draft standard defines the System Application Programming Interface (API) for Threads Extensions for the C language.

- IEEE Draft Std 1003.13/D5

This draft standard defines four real-time application environment profiles (AEPs) based on the POSIX standard and draft standards.

OS Open Support for the POSIX Standard and Draft Standards

IEEE prohibits claims of conformance to draft standards, so IBM cannot claim that the OS Open operating system conforms to the POSIX draft standards. However, the intent is for the OS Open operating system to conform to the Minimal Realtime System Profile and the Realtime Controller System Profile, as defined in IEEE Draft Std 1003.13/D5, at the earliest possible opportunity.

The OS Open operating system defines the following configuration variables for each of the standard and draft levels. Applications can use C preprocessor **#ifdef** constructs to test configuration variables to determine the presence or absence of various operating system features.

- IEEE Standard 1003.1

```
_POSIX_SINGLE_PROCESS
_POSIX_DEVICE_IO
_POSIX_NO_TRUNC
_POSIX_VERSION
```

```
_POSIX_FSYNC
_POSIX_MAPPED_FILES
_POSIX_SHARED_MEMORY_OBJECTS
_POSIX_SEMAPHORES
_POSIX_MEMLOCK
```

- _POSIX_MEMLOCK_RANGE
- _POSIX_SHARED_MEMORY_OBJECTS
- _POSIX_TIMERS
- _POSIX_MESSAGE_PASSING
- _POSIX_SYNCHRONIZED_IO
- IEEE Draft Standard 1003.1c/D9
 - _POSIX_THREADS
 - _POSIX_REENTRANT_FUNCTIONS
 - _POSIX_THREAD_ATTR_STACKSIZE
 - _POSIX_THREAD_ATTR_STACKADDR
 - _POSIX_THREAD_PRIORITY_SCHEDULING
 - _POSIX_THREAD_PRIO_INHERIT
 - _POSIX_THREADS_PRIO_PROTECT

Chapter 3. Understanding the OS Open Architecture

The OS Open operating system comprises a real-time executive and several optional libraries, each supplying a class of functions and macros.

The real-time executive provides an operating system core for embedded applications. Depending on an application's requirements, an embedded application may also incorporate one or more optional libraries.

This modular approach enables embedded system developers to scale an OS Open operating system image to closely match application requirements. Because unused features are not included, an OS Open configuration can provide savings in system hardware, initialization and reset time, and program size when compared with a general-purpose operating system.

Table 3-1 summarizes the OS Open optional libraries, which are described in this chapter. For detailed descriptions of the OS Open functions and macros mentioned in this chapter, refer to *OS Open Programmer's Reference*.

Appendix A, "OS Open Functions Listed by Library," lists OS Open functions by library.

Table 3-1. OS Open Optional Libraries

Library	File Name
6xx alignment exception support library	alignLib.a
ANSI C library	cLib.a
ANSI C I/O library	fsLib.a
ANSI C Math library	mathLib.a
Block buffer library	bbuffLib.a
C++ runtime support (C Set ++ [™] support) Library	cpprtLib.a
Card services (PCMCIA support) library	csLib.a
Debugger support library	dbLib.a
Device and file support library	devLib.a
DOS file system support library	fatLib.a
Dynamic loader library	ldrLib.a
File Transfer Protocol (FTP) support library	ftpLib.a
Floating-point emulation library	fpeLib.a
Kernel abstract data types library	kadtLib.a
Network file system (NFS) client library	nfsLib.a
Network support library	netLib.a
OpenShell	shell.o
OS Open minimal kernel	rtxmin.o

Table 3-1. OS Open Optional Libraries

Library	File Name
OS Open kernel extensions	rtxext.o
PCMCIA ATA/IDE support library	pataLib.a
Queue library	queLib.a
RAM disk library	ramdLib.a
Rate monotonic scheduling (RMS) library	rmsLib.a
Remote source level debugger library	rsldLib.a
Ring buffer library	rngLib.a
RPC/XDR protocol support library	rpcLib.a
Run library	runLib.a
Socket services support library	ssLib.a
Symbol support library	symLib.a
TCP/IP protocol support library	tcpipLib.a
Telnet client support library	telnet.o
Telnet server support library	tnetdLib.a
Trivial file transfer protocol	tftp.o
TTY support library	ttyLib.a
XL C compiler support library (xcoff only)	xlclib.a

The OS Open operating system also provides several platform-specific libraries, which are described in the platform-specific user's guides.

The Real-time Executive

The real-time executive, the only required component in an OS Open operating system, provides a full set of basic operating system services:

- System management
- Thread management
- Storage management
- Signals
- Clocks and timers
- Interrupt and fault handling
- Message queues
- Semaphores
- Trace buffer support
- Miscellaneous services

The C functions for the real-time executive functions are in two files, **rtx.o** and **rtxLib.a**. The **rtx.o** object module contains the OS Open kernel. The **rtxLib.a** library contains interface routines to OS Open functions, and is linked with application programs to resolve calls to the real-time executive.

The OS Open minimal kernel is appropriate for deeply embedded applications, where memory resources are extremely constrained and all the OS Open functions are not required. The OS Open minimal kernel, **rtxmin.o** is used instead of the full OS Open kernel **rtx.o**.

The following functions are available in **rtxmin.o**:

int_faultfield(), int_getid(), int_install(), int_query(), kill(), memheap_alloc(), memheap_extend(), memheap_free(), memheap_query(), pthread_create(), pthread_equal(), pthread_errno_np(), pthread_exit(), pthread_getschedparam(), pthread_getspecific(), pthread_getstackaddr_np(), pthread_key_create(), pthread_key_delete(), pthread_kill(), pthread_self(), pthread_setschedparam(), pthread_setspecific(), pthread_resume_np(), pthread_suspend_np(), sched_yield(), sem_init(), sem_post(), sem_getvalue(), sem_destroy(), sem_close(), sem_timedwait(), sem_trywait(), sem_wait(), sigwait(), sysconfig(), timertick_notify(), trace_get(), trace_write()

Additional OS Open functions can be added to the minimal kernel using the **sysconfig()** function. When functions are added to the minimal kernel, the functions being added need to be specified with the same names as user functions except that the first letter of each added function name needs to be capitalized. For example, to add the **pthread_mutex_lock()** function to the minimal kernel, the following code can be executed:

```
svc_number=pthread_mutex_lock_call;
sysconfig(SVC_INSTALL, Pthread_mutex_lock, &svc_number)
```

Note: When the OS Open minimal kernel is used, additional OS Open libraries cannot be used, except for **cLib.a**, **ppcLib.a**, **fpeLib.a** and **xlcLib.a**. Most of the functions in the above libraries do not require functions other than these provided in the minimal kernel. If an OS Open function is called that is not available in the minimal kernel and this function was not installed using **sysconfig()**, an invalid call/invalid operation error will be generated.

The OS Open minimal kernel is about 55% to 60% the size of the full function OS Open kernel.

System Management

The real-time executive provides functions, listed in Appendix A, to control system operation.

Thread Management

A thread, the basic unit of schedulable execution in the OS Open operating system, consists of a stack and an execution context. An execution context is an operating system object that stores the machine state associated with a thread while the thread is not running.

The OS Open operating system consists of only one process executing in one address space. All threads run in the system process, and the running threads share all system resources. Threads can create additional threads. OS Open provides functions to create threads, control thread execution, and synchronize the execution of concurrent threads.

For a thorough discussion of threads and thread functions, see Chapter 6, "Understanding OS Open Threads."

Storage Management

OS Open supports two storage management mechanisms that solve two important problems in embedded application programming:

- Efficient allocation of working storage for application programs

Working storage requirements are difficult to predict. Working storage is usually acquired for relatively long periods and is often needed throughout the execution of an application program.

- High-performance management of fixed-size storage blocks, such as those used for communications and transaction buffers

Such blocks are acquired and released frequently, and are used for short periods. These blocks must be managed for low operating system overhead.

OS Open heaps solve the problems of allocating working storage for application programs. Buffer pools solve the problems of managing high-performance, deterministic allocation of fixed-size storage blocks.

OS Open Heap Management

The OS Open heap implementation contains features, not typically found in traditional heap designs, that increase the usefulness of the heap implementation for embedded applications.

The OS Open system can be configured to use one or two heaps. Heap usage can be set in the kernel configuration block at initialization, or extended during system operation.

A one-heap system is simplest to configure. One heap attempts to satisfy storage requests both from the OS Open kernel and from application programs. Storage requests to the kernel or the application heap are internally redirected to the single heap.

A two-heap system provides improved system storage protection at the cost of increased complexity. Such a system has a system heap separate from the application heap. Because the heaps typically are not contiguous, storage exceptions in an application program are less likely to affect the operating system. In embedded systems using hardware-shared storage, the use of two heaps allows kernel data to be stored separately, if necessary, to conserve the shared-storage resource.

See Appendix A for a list of the heap management functions in **rtxLib.a**.

OS Open Buffer Pool Management

An OS Open configuration can have an arbitrary number of buffer pools. Buffer pool control block information is stored, in system storage, separately from the managed pool. For systems using hardware shared buffer pools, this feature provides the most efficient use of hardware shared storage.

See Appendix A for a list of the buffer pool management functions in **rtxLib.a**.

Signals

Signals are generated using **kill()** and **pthread_kill()**, which direct signals to the OS Open process and to threads, respectively.

An unblocked signal causes any synchronization primitive, such as **sem_wait()**, to terminate prematurely with a return code of EINTR. This is useful when terminating blocking functions, such as **mq_receive()**, during exception conditions. Also, a signal satisfies and is consumed by a **sigwait()** function specifying the signal. This method is recommended for signals controlling thread-specific behavior.

Thread signal masks control thread behavior upon signal reception. Appendix A lists functions contained in **rtxLib.a** used to set, clear, and test thread signal masks.

No signals are automatically connected to processor exception-handling mechanisms. The interpretation of exceptions varies among PowerPC processors and board-level products. User-written fault handlers can easily use **kill()** to assert process-level signals.

OS Open defines all required POSIX signals. Some signals are not supported.

Note: The OS Open operating system does not provide **sigaction()**; its utility is limited compared to the system complexity and performance degradation caused by its implementation.

See Appendix A for a list of the signal-handling functions in **rtxLib.a**.

Clocks and Timers

OS Open supports a POSIX real-time clock, **CLOCK_REALTIME**, defined in the file **<time.h>**. The real-time clock is maintained by timer ticks from an external source supplied to the real-time executive.

CLOCK_REALTIME specifies the current time is defined as the elapsed time in seconds and nanoseconds since 00:00:00 January 1, 1970 coordinated universal time (UTC). Actual clock resolution depends on the external source, which is typically resolved to milliseconds.

See Appendix A for a list of functions in **rtxLib.a** used to retrieve information from or set **CLOCK_REALTIME**.

A thread can use a timer to send a signal to the creating thread. Such timers can be configured to send a signal periodically, or to issue a signal only once.

If a timer expires and a signal is pending on a thread counter overrun, the real-time executive increments a counter in the timer. **timer_getoverrun()** can retrieve and automatically reset this overrun count.

See Appendix A for a list of functions in **rtxLib.a** used to work with timers.

Interrupt and Fault Handling

Application developers can write code using an OS Open mechanism to handle interrupts and processor faults. These terms differ from those used in PowerPC processor publications such as *PowerPC 601 RISC Microprocessor User Manual*.

In this book, an interrupt is an action taken as the result of an event. The causing event may be initiated by the real-time executive, or may result from one of a specific set of events termed exceptions. A processor fault is an exception resulting from the attempted execution of an instruction. The OS Open mechanism handles both types of interrupts.

In OS Open applications, user-written C code to handle a specific event is contained in a first-level interrupt handler (FLIH). FLIHs may contain OS Open functions, except for those that require a thread context, such as **pthread_yield()**, or that may block, such as **sem_wait()**. OS Open functions available to FLIHs are denoted as “Interrupt Handler Safe” in their function descriptions in *OS Open Programmer's Reference*.

For more information about OS Open interrupt handlers, see Chapter 7, “Writing OS Open Interrupt and Fault Handlers.”

See Appendix A for a list of the interrupt- and fault-handling functions in **rtxLib.a**.

Message Queues

A message queue passes messages between threads. In many operating systems, message queues are used for interprocess communication (IPC); in the

OS Open operating system, which has only one process, the real-time executive enables an interthread communication mechanism.

The real-time executive uses a message queue descriptor to establish a named connection between the system process and a message queue. Threads use this connection to communicate with other threads. Several threads may open a message queue by name. Message queues can be created in various sizes to hold varying numbers and sizes of messages.

Ordinarily, message queues block if a thread attempts to send a message to a full message queue or if a thread attempts to receive a message from an empty queue. This behavior can be changed dynamically using **mq_setattr()** with the appropriate message queue attribute object.

See Appendix A for a list of functions contained in **rtxLib.a** used to work with message queues.

Semaphores

The OS Open operating system provides named and unnamed counting semaphores, which are generalized binary semaphores.

While the count value is positive, **sem_wait()** returns immediately to the calling thread, decrementing the count; a **sem_wait()** against a semaphore having a count value less than or equal to 0 or less causes the calling thread to block. Therefore, if the count value is positive, **sem_wait()** calls can be issued before a calling thread is suspended. If the count value is negative, the absolute value of the count represents the number of threads blocked on that semaphore.

In the real-time executive, named and unnamed semaphores differ only in how they are created and destroyed. Named semaphores are provided for compatibility with applications that share named semaphores among processes.

See Appendix A for a list of functions in **rtxLib.a** used to work with semaphores.

Trace Buffer Support

OS Open supports an optional circular trace buffer for system- and user-defined events.

To take a snapshot of the OS Open trace buffer, use **trace_snapshot()** to copy the trace buffer into a buffer. The information copied to the buffer can be written to a file and moved to an RS/6000 system for processing.

There are several ways to transfer the file from the OS Open system to a RISC System/6000 computer.

- Write the file to a DOS FAT formatted diskette on a diskette drive attached to the OS Open system. Then, read the diskette on the RISC System/6000 using the AIX **dosread** command.

- Write the file to an OS Open file system. Then, use FTP to send it over a TCP/IP connection to the RISC System/6000.
- Rather than writing directly to a file, capture the standard output of **trace_snapshot()** on the OS Open console using the screen capture feature of the terminal emulator. Then, write the captured output to a file.

After the file is on the RISC System/6000, run the AIX **tracefmt** program (located in `/usr/osopen/bin`), passing the file name, for example:

```
tracefmt trace.buf
```

tracefmt produces the output file `trace.out`.

When running on AIX 4.1 **trc41** must be run before **trcrpt**. **trc41** produces and output file named `trace.out4.1`.

```
trc41 trace.out
```

The AIX trace formatting command **trcrpt** can format `trace.out`. You will need to use the template file **trcfmt.templ** appropriate for your OS Open platform. The file is located in `/usr/osopen/<platform>/samples`.

The AIX command:

```
trcrpt -t /usr/osopen/<platform>/samples/trcfmt.templ -o trace.fmt trace.out
```

The AIX 4.1 command:

```
trcrpt -t /usr/osopen/<platform>/samples/trcfmt.templ -o trace.fmt trace.out4.1
```

writes the formatted trace output to the AIX file `trace.fmt`. See *AIX Commands Reference* (GC23-2393) for information about the **trcrpt** command.

The trace buffer is established and configured at system initialization. If a trace buffer is provided, the real-time executive writes information about all system calls and processor faults to the trace buffer. **trace_write()** writes a user-defined entry into the trace buffer.

For more information about initialization, see Chapter 4, "Configuring the OS Open Operating System."

Miscellaneous Services

Like many operating system kernels, the real-time executive supplies numerous utility functions. For example, **setuname()** allows an application to set certain fields to be returned by a subsequent call to **uname()**.

See Appendix A for a list of miscellaneous functions contained in **rtxLib.a**.

The ANSI C Libraries

The ANSI C libraries contain more than 125 functions providing source code portability for applications using ANSI C libraries.

In the OS Open implementation, the following libraries contain the ANSI C functions and the ANSI C reentrant functions required by POSIX 1003.4a/D8:

Library	Description
cLib.a	Contains basic ANSI C functions, except for math and file handling functions
mathLib.a	Contains ANSI C math functions
fsLib.a	Contains ANSI C functions requiring underlying device support (provided in devLib.a) to implement the file descriptors stdin , stdout , and stderr

The ANSI C libraries can be linked to by an application program as needed.

Appendix A lists the functions contained in **mathLib.a** and **fsLib.a**.

ANSI C functions fall into the broad classes described under the following headings.

Character Handling

The character handling functions test and map printable and control characters in the seven-bit ASCII character set, also referred to as the “C” locale.

OS Open supports only the “C” locale; printable characters are those having values from X'20' (space) through X'7E' (tilde). Control characters are those having values from X'00' (NULL) through X'1F' (US) and X'7F' (DEL).

The following character handling functions are provided for ANSI C compatibility, but because of limited locale support perform minimal functions: **mblen()**, **mbtowc()**, and **wctomb()**.

Nonlocal Jumps

Nonlocal jumps bypass the normal C function call and return model, allowing control to jump from one function to another.

The real-time executive provides POSIX 1 functions, **sigsetjmp()** and **siglongjmp()**, that perform the same basic functions as **setjmp()** and **longjmp()**. These functions save and restore the current signal mask and the execution context.

Variable Arguments

A function may be called using a varying number of arguments of varying types. Declared in a prototype using ellipses, a variable argument is a function called with an unknown number or type of arguments; the necessary information is received when the function is translated.

A function that accepts a variable list of arguments must have at least one known argument type, which must be the first argument or arguments specified. In the following example, the variable *format* is the known argument type.

```
int printf(const char *format, ...)
```

The file **<stdarg.h>** defines several macros for handling variable arguments, along with a data type, **va_list**. The data type stores information used by the macros to access the variable argument list.

Once initialized, the **va_list** type may be passed as an argument to another function. The value of the **va_list** type is undetermined when the function returns.

Both **va_start()** and **va_end()** should be invoked within the function receiving the variable list of arguments.

Input/Output

High-level input/output (I/O) is implemented by the operating system rather than by thread.

Before a C library I/O function can be used, the file system must be initialized using **fs_init()**. The **fs_init()** call assumes that the low-level device subsystem was previously initialized and that three successful **open()** function calls established the file descriptors used by C library I/O functions:

0	stdin (standard input)
1	stdout (standard output)
2	stderr (standard error)

The application initializing the device driver subsystem determines the maximum number of open files. The **devLib.a** function **dev_io_init()** initializes the I/O device subsystem and establishes open file and device driver count limits.

flockfile() and **ftrylockfile()** ensure exclusive access to a file. **funlockfile()** relinquishes exclusive access. All functions that access a file using a pointer to type **FILE** for reading, writing, or changing any file attributes lock the file before accessing it. The only exceptions are **getc_unlocked()**, **getchar_unlocked()**, **putc_unlocked()**, and **putchar_unlocked()**.

A thread blocks on locking a file held locked by another thread. The thread holding the lock on the file does not block on subsequent attempts to lock the same file. A thread may nest **flockfile()** and **funlockfile()** calls; each **flockfile()** must have a matching **funlockfile()**. **ftrylockfile()** is a non-blocking version of **flockfile()**.

ANSI C supports the following types of I/O buffering:

- Full buffering, when data is read from and written to a stream, in buffer-sized blocks, after the buffer becomes full or is flushed.
- Line buffering, which occurs when one of the following events occurs:
 - Data is read from the stream into an empty buffer.
 - A newline character (`\n`) is read from or written to the stream.
 - The buffer is flushed.

No buffering occurs when data is moved to and from a stream a byte at a time.

By default, newly opened files are line-buffered using a buffer size of **BUFSIZ**, which is defined in the file `<stdio.h>`. **stdin**, **stdout**, and **stderr** are line-buffered, using a buffer of **BUFSIZ**.

setbuf() and **setvbuf()** change buffering characteristics such as type and size of the buffer, and whether a system or user-defined buffer is used. However, default buffering characteristics should be changed only before attempts to access a file. An attempt to change the buffer after a file is accessed result in an error.

General Programming Utilities

The ANSI C libraries provide a rich set of general programming utilities for string and number conversions, pseudo-random number generation, environment variable handling, searching and sorting, and integer math calculations.

Storage Allocation

The storage allocation functions include **free()**, **malloc()**, **calloc()**, and **realloc()**. The OS Open implementation allocates storage from the application heap. In a single-heap system, the application heap and the kernel heap share the same address space.

A successful storage allocation request returns a pointer to the beginning (lowest byte address) of the allocated storage. The pointer is suitably aligned to be assigned to a pointer that can be used to access any type of object, or an array of objects, in the allocated storage until it is explicitly freed or reallocated.

If the storage size requested is 0, each function returns an error. If no storage can be allocated, each function returns a NULL pointer.

String Handling

The string handling functions work with arrays of characters and other objects treated as strings. Various methods are used to determine array sizes. A pointer to type **char** or **void** always points to the character at the lowest byte address in the array.

String handling functions are provided to copy, compare, and concatenate strings; search for the first or last occurrence of characters within strings; split a string into tokens; and search for substrings within a string.

Functions that work with storage specified by a starting address and length are classed with the string functions.

Time and Date

Before the ANSI C time and date functions can be used, a FLIH that periodically calls **timertick_notify()** must be installed and the application program must set the system clock using **clock_settime()**. The time and date functions assume that the software clock is set to the correct number of seconds since 00:00:00 January 1, 1970 UTC.

The C library provides several functions for manipulating expressions of time and date. These functions can express time in either of two types:

Calendar time	Indicates that the current date and time are represented by the Gregorian calendar as UTC (formerly Greenwich mean time (GMT))
Local time	Indicates that the calendar time is expressed for a specific time zone and with local daylight saving time (DST) corrections added

OS Open uses the environment variable **TZ** to specify the local time zone.

The **TZ** variable is defined as follows. Brackets ([]) enclose optional items; the brackets are not part of the value definition. Defaults are supplied for unspecified items.

TZ=stdoffset[dst[offset],rule]

The fields in the value are defined as:

std	Standard time zone abbreviation containing at least three characters; cannot include numeric characters, plus sign (+), minus sign (–), or comma (,)
offset	The offset of time from UTC in the time zone A negative value indicates that the time zone is east of the prime meridian (0° longitude); a positive value indicates that the time zone is west of the prime meridian. For example, United States eastern standard time (EST), <i>offset</i> is expressed as “[+] <i>5</i> ”.
dst	Daylight savings time zone abbreviation containing at least three characters; cannot include numeric characters, plus sign (+), minus sign (–), or comma (,). An offset of time from UTC may be given.

rule	The “rule” for the start and end of DST; specified as follows:		
	start[/time],end[/time]		
	start, end	Mm.n.d	
	M	M (literal)	
	m	month (1–12)	
	n	Week of the month (1–5; 5 represents the last week of the month)	
	d	Day of the week (0–6, 0 = Sunday)	
	time	hh[:mm[:ss]], defined as:	
		hh = hours (0–23)	
		mm = minutes (0–59)	
		ss = seconds (0–59)	

By default **TZ** is defined for in the United States, expressed as:

TZ=EST5EDT,M4.1.0/02:00:00,M10.5.0/02:00:00

To change the default definition, use **setenv_np()**.

Device and File Support Library

The device and file support library provides the POSIX 1 file and device access model to application programs that have devices or manage regular files. This library, contained in the file **devLib.a**, supports device drivers and the high-level I/O calls such as **open()** and **read()**.

The device library supports drivers for character devices, block devices, logical file systems. The device library includes integrated drivers for a null device and for shared memory.

Device drivers are installed at runtime and are easily added to the OS Open environment at any time, even while the operating system is running.

The device and file support library must be initialized, using **dev_io_init()**, before the low-level I/O functions can be used. This function establishes the maximum number of open files and simultaneously installed device drivers supported in an OS Open configuration.

Appendix A lists functions in the **devLib.a** library.

Low-Level POSIX I/O Support

The low-level POSIX 1 I/O support functions include **open()**, **close()**, **read()**, **write()**, **ioctl()**, and **strategy()**.

For information about device driver processing for these functions, see Chapter 8, "Writing OS Open Device Drivers."

For device-specific information about **ioctl()**, see device documentation and the platform-specific user's guides.

Supporting Regular Files on a Logical File System

To support regular files on a logical file system, **open()**, **read()**, **write()**, and **close()** are extended to support directory trees, working directories, and file status.

Some logical file systems do not completely support the POSIX file model. For example, the supplied DOS logical file system does not completely support status time and permission mode fields; a DOS directory entry does not contain the required information.

IEEE 1003.1b-1993 states that working directory information shall be maintained at the process level. OS Open maintains a directory prefix that is prepended to all pathnames that do not start with / (slash).

The POSIX 1 file model maintains several fields that track file access, creation, and update times, along with information about file permissions.

Integrated Device Drivers

The device library support includes two integrated device drivers, the **/dev/null** device driver and the shared memory device driver.

Null Device Driver

The null device, **/dev/null**, provides a handy way for output to be ignored. The null device driver establishes the **/dev/null** device in the device I/O library root directory space.

open() can open files against the null device. **write()** calls to **/dev/null** always succeed, and **read()** calls succeed with 0 bytes returned.

Shared Memory Device Driver

IEEE 1003.1b-1993 specifies an interface allowing the creation of named regions of storage. These regions enable processes to share data stored in the regions. Because the OS Open operating system executes one process, the utility of its shared storage implementation is limited.

Storage regions are named according to POSIX conventions. In the OS Open implementation, names must be preceded by the prefix **/shm/**.

To use shared memory objects to name specific regions of storage, **ioctl()** is used with a shared memory file descriptor. The syntax is:

```
void *addr, size_t extent;
```

```
rc = ioctl(shm_fd, SET_SHM, addr, extent);
```

If **ioctl()** is successful, the return code (rc) is 0. **SET_SHM** cannot refer to defined shared memory regions. If a region is already defined, the return code is **-1** and *errno* is set to **EEXIST**.

The following example opens a named memory region, associating it with a memory-mapped video device at address **X'A000'** for **X'1000'** bytes:

```
int rc;
int shm_fd;

shm_fd = shm_open("/shm/video_ram", O_CREAT | O_RDWR, 0);
if (shm_fd == -1) {
    perror("problem creating shared memory file");
    return(-1); }
rc = ioctl(shm_fd, SET_SHM, (void *) 0xA000, 0x1000);
if (rc != 0) {
    perror("SET_SHM ioctl failed");
    return(-1); }
```

Storage Locking

Functions for locking and unlocking storage pages are included in the device library support. Because all storage is inherently locked in the OS Open operating environment, these functions simply check parameter validity and return.

Appendix A lists storage-locking functions contained in the **devLib.a** library.

Terminal Support Library

The **ttyLib.a** library contains POSIX 1 terminal (tty) support functions.

In the OS Open operating system, POSIX 1 terminal support is implemented as a device driver interface. A device driver, such as the asynchronous port device driver, is required to complete the terminal support implementation.

The terminal support library provides functions for accessing terminal attributes through the **termios** structure defined in header file **<termios.h>**.

OS Open extensions to the POSIX 1 terminal functions include:

- Retrieval of previously entered lines.
- Programming of up to 12 function keys.
- Mapping of newline character to carriage-return, new-line.
- Output processing of tab characters.

The extensions are activated by setting the **IEXTEN** local mode mask and **OPOST** output mode mask.

For more information about the POSIX 1 general terminal interface, refer to IEEE 1003.1b-1993.

Appendix A lists functions contained in the **ttyLib.a** library.

Ring Buffer Library

The ring buffer library implements a circular ring buffer as an abstract data type. Ring buffers work with byte serial devices that supply or request data in multibyte bursts, such as a serial port that transfers a multibyte packet while the application processes another transaction.

Appendix A lists the functions in the **rngLib.a** library.

Queue Library

The queue library implements a doubly-linked queue as an abstract data type. Doubly-linked queues are common in embedded applications.

Appendix A lists the functions in the **queLib.a** library.

DOS File System Support Library

An OS Open operating system can be configured with a logical file system that processes files on media formatted to DOS 5.0 conventions. The DOS logical file system implements a POSIX 1 file system, within the constraints of the DOS directory and file structure.

The **fatLib.a** library, which contains the DOS file support functions, serves as an intermediary between the OS Open high-level I/O calls and an underlying block device driver for DOS-formatted media.

For more information about DOS file system support, see Chapter 10, "Using the DOS Logical File System."

RAM Disk Library

An OS Open RAM disk can be used with the DOS Logical File System Support Library to implement a DOS file system in system RAM.

The **ramdLib.a** library can be used to create and initialize a RAM disk, after which the DOS (FAT) Logical File System is loaded and a DOS directory created.

For more information, see Chapter 10, "Using the DOS Logical File System."

PCMCIA Support Libraries

PCMCIA support functions are provided in three OS Open libraries: card services in **csLib.a**, ATA/IDE initialization in **pataLib.a**, and socket services in **ssLib.a**.

For more information, see Chapter 11, "OS Open PCMCIA Support."

Network Support Library

OS Open network support functions, contained in the **netLib.a** library, provide the networking utilities **ping()** and **ifconfig()**, network interface drivers for standard Ethernet and Serial Line Interface Protocol (SLIP), and functions to search network databases, such as **gethostbyaddr()** and **gethostbyname()**.

For more information, see Chapter 12, "OS Open Networking Features."

Appendix A lists networking support functions contained in the **netLib.a** library.

TCP/IP Protocol Support Library

The TCP/IP support library, contained in the **tcplib.a** library, implements the TCP/IP protocol stack and socket interfaces defined in UNIX Berkeley Standard Distribution (BSD), version 4.3.

For more information, see Chapter 12, "OS Open Networking Features."

Appendix A lists TCP/IP support functions contained in the **tcplib.a** library.

NFS Client Support Library

The network file system (NFS) client support functions, contained in the **nfsLib.a** library, provides a client implementation of NFS.

For more information, see Chapter 12, "OS Open Networking Features."

Appendix A lists NFS support functions contained in the **nfsLib.a** library.

Run Library

The run library (**runLib.a**) parses a command line, loads the program, and executes it with the parameters that were entered. The run library can be called with a single command, or called to continue prompting for commands until "exit" or Ctrl-D is entered. **run()** will search all the directories in the environment variable "PATH" for a file with the command name and suffix ".ld". It will load it, create a thread, and call the entry point, passing the standard *argc* and *argv* parameters.

For more information on the run library, see the *OS Open Programmer's Reference* for descriptions of the **run()** and **getopt()** functions.

RPC/XDR Protocol Support Library

The remote procedure call (RPC)/external data representation (XDR) protocol support library, contained in the **rpcLib.a** library, supports data communications.

Appendix A lists RPC/XDR support functions contained in the **rpcLib.a** library.

File Transfer Protocol Support Library

The **ftpLib.a** library supports File Transfer Protocol (FTP) client and server operations, as defined by UNIX BSD, version 4.3.2, over TCP/IP networks.

The library is primarily intended for interactive use from an OpenShell or equivalent interface.

For more information, see Chapter 14, "Using File Transfer Protocols (FTP and TFTP)."

The **ftpLib.a** library contains the **ftp()** and **ftpd_start()** functions.

Telnet Support Libraries

The object file **telnet.o** contains the Telnet client function **tn()**. This function, ported from BSD 4.3, is another optional OS Open networking utility.

tn() provides interactive terminal services over TCP/IP connections to other hosts supporting Telnet.

The **telnetd_start()** function in **tnetdLib.a** starts the telnetd server.

For more information, see Chapter 15, "Using Telnet."

6xx Alignment Exception Support Library

The **alignLib.a** library contains support for processing alignment exceptions that can occur when using a PowerPC processor. Architecturally, a PowerPC processor can raise an alignment exception whenever a load or store operation accesses an address not on the "natural" boundary for the instruction. PowerPC 6xx processors silently handle the vast majority of these cases in hardware. Some of the cases where the software must emulate the access include

- Misaligned access crosses page (4K) boundary with Data Relocation active
- Floating point access not word aligned

The alignment handler provided in `alignLib.a` is a general purpose alignment handler implementing all of the exception cases for Big Endian 6xx operation. You may want to provide your own handler to reduce space/increase performance for specific operating environments. For example, those applications not utilizing Data Relocation and Floating Point typically do not require any alignment handler.

Appendix A lists the functions contained in the **alignLib.a** library.

Debugging Support Library

The **dbLib.a** library contains debugging functions for handling software breakpoints, displaying storage and execution information, and performing miscellaneous debugging tasks.

The debugging functions are primarily used by OpenShell to implement its native debug environment, but they can also be used by applications.

Appendix A lists the functions contained in the **dbLib.a** library.

OpenShell

The object file **shell.o** implements OpenShell, a powerful native system debugging and testing tool for embedded application programs. OpenShell executes on the target system, requiring only an ASCII display and keyboard for communication with the target system.

OpenShell features a C interpreter. Most C functions can be called from the OpenShell command line, and developers can interactively define C functions to perform application-specific debugging tasks.

To aid embedded system debugging, developers can interactively call OS Open functions to decode and display OS Open control blocks. Supplied OpenShell commands can set thread breakpoints, disassemble instructions, and display storage.

For more information, see Chapter 18, "Using OpenShell."

Remote Source Level Debugger Library

The **rsldLib.a** library supports remote debugging and contains only the function **rsld_start()**.

The **rsldLib.a** library allows software running in the target system to communicate with the RISCWatch Debugger to support hardware and software debugging in a flexible windowed development environment.

Kernel Abstract Data Types Library

The **kadtLib.a** library, primarily used by OpenShell and associated debugging functions, provides functions to access objects in the OS Open kernel.

Formatting kernel data objects into readable characters, even without a complete understanding of the OS Open implementation, helps in problem determination. Kernel abstract data types enable OpenShell, debuggers, and applications to handle kernel data using minimal information of the kernel implementation.

The functions provide a wide variety of formatted information about various kernel data object types.

Appendix A lists the functions contained in the **kadtLib.a** library.

Symbol Support Library

When an OS Open application is compiled, symbol information is incorporated into the load image unless specific steps are taken. The OS Open static symbol table library provides functions to look up the address of external functions and variables using their respective names.

Appendix A lists the symbol support functions contained in the **symLib.a** library.

Dynamic Loader Library

OS Open provides a dynamic loader in the library **ldrLib.a**. A file to be loaded must contain a loader section. By convention, the file name should end with ".ld," but this convention is not a requirement.

The XCOFF-based OS Open loader library functions load executable files in extended common object file format (XCOFF) format into storage and link the files with previously loaded code. The AIX linker produces XCOFF format executable files.

The ELF-based OS Open loader library functions load executable files in ELF format into storage and link the files with previously loaded code. The ELF linker/binder produces ELF-format executable files.

Appendix A lists the dynamic loader functions contained in the **ldrLib.a** library.

XL C Compiler Support Library (xcoff only)

The **xlclib.a** library contains the OS Open implementation of XL C compiler-intrinsic functions. This library is available only in an OS Open xcoff release.

The XL C compiler for OS Open applications generates internal function calls to optimize appropriate operations for a specific processor target. Such operations include filling storage regions, comparing storage regions, and performing block moves of storage.

The compiler support library allows the compiler-intrinsic functions to be optimized for various processor architecture implementations without requiring compiler modifications.

C++ Runtime Support (C Set ++ Support) Library

The **cpprtLib.a** library contains support for using C++ applications under OS Open with the C Set ++ compiler. Functions are provided to initialize and terminate the C++ runtime environment, the most important part of which is to invoke and initialize static constructors or destructors defined in the application load image.

Also, the library contains the default **new** and **delete** functions, which are implemented using the services of **malloc()** and **free()**.

Appendix A lists the functions contained in the **cpprtLib.a** library.

The Floating Point Emulation Library

The **fpeLib.a** library contains floating point emulation functions, which are used by OpenShell and the C library.

Rate Monotonic Scheduling (RMS) Library

The **rmsLib.a** library contains rate monotonic scheduling (RMS) functions, which are listed in Appendix A .

Part 2. Configuring the OS Open Operating System

Chapter 4. Configuring the OS Open Operating System

OS Open requires information about the system environment at initialization. This information is contained in an object called a kernel configuration block, which must be provided by an application designer.

The configuration block address is passed in general purpose register 3 (GPR3) when the system bootstrap code calls the kernel entry point. If XCOFF object format is being used, the TOC pointer must be set correctly in GPR2.

A configuration block type definition, **conf_t** in the file **<config.h>**, is provided to declare the kernel configuration block data.

If bootstrap code supplied with a supported OS Open platform is used, the mem0 starting address and mem0 size will be adjusted to allow the maximum use of heap storage.

Kernel Configuration Block Fields

The following list defines the fields in the configuration block:

Main function	Points to an initial function which receives control as the first thread started by the real-time executive.
Panic function	Points to a user-supplied function called when an unrecoverable system error occurs. For more information, see “The Panic Function” on page 4-2.
Main argc	Points to the first parameter of the main function.
Initial thread stack size	The stack size of the initial thread; this value becomes the stacksize attribute in the thread attributes object for the internal pthread_create() establishing the initial thread.
mem0 starting address	The starting address of the required kernel heap; this address is typically initialized to zero as it will be modified by the bootstrap code.
mem0 size	The size, in bytes, of the kernel heap; this field is typically set to the memory size as it will be modified by the bootstrap code.
mem1 starting address	The starting address of the optional application heap.
mem1 size	The size, in bytes, of the application heap. If a two-heap system is not used, this value is 0.

Trace buffer size	The size, in bytes, of the buffer used by OS Open trace functions. This value must equal 0 or be greater than 24.
Tick resolution	The number of nanoseconds in a tick, ranging from 50000 and 1000 million, inclusive. This value is added to the nanoseconds portion of the OS Open POSIX clock when timertick_notify() is called.
Time slice	The time slice for a thread when round robin scheduling is in use; this value must be a multiple of the tick resolution field.

The following example shows a typical kernel configuration block declaration that has one heap that uses whatever memory is left of the 4Meg storage after bootstrap. In this example tracing is turned on by the compile option “-DKERNELTRACE”.

```
#include <config.h>

void thread0();
void panic();
const conf_t _Kernel_Config_Block = {
    thread0,          /* First thread function */
    panic,            /* Panic function */
    0x00000000,       /* First thread argument */
    0x00040000,       /* First thread stack size */
    (void *)0x00000000, /* Memory heap one start address */
    0x00400000,       /* Memory heap one size */
    0x00000000,       /* Memory heap two start address */
    0x00000000,       /* Memory heap two size */
#ifdef KERNELTRACE
    0x000004b0,       /* Trace table size */
#else
    0x00000000,       /* Trace table size */
#endif
    0x00989680,       /* Tick rate */
    0x09896800,       /* Round robin time slice */
};
```

The Panic Function

The panic function enables an application to gain control when a fault occurs that has no fault handler installed or when **abort()** is called. The panic function is also called if a first-level interrupt handler (FLIH) attempts to perform a floating point operation.

Two parameters are passed to the panic function; their values are defined in **<panic.h>**.

- The source of the panic request

If this value is 0, the panic occurred during the execution of a FLIH.

Two values, defined in **<panic.h>**, may be passed. One indicates a system initialization error; the other indicates an application request to use **abort()** to abnormally terminate OS Open.

Any other value is interpreted as the thread ID of the thread executing when the problem occurred.

- The cause of the panic request

An example value is **Panic_invalid_instruction**, which indicates that an invalid instruction was encountered when no invalid instruction handler was installed.

Part 3. Writing OS Open Applications

Chapter 5. Using OS Open Features in Hard Real-Time Systems

Several OS Open programming features support the development of hard real-time systems.

A fundamental performance measurement for a real-time system is the ability to meet specified task deadlines; when a result is produced is as important as its correctness. In a hard real-time system, failing to meet deadlines can be catastrophic. Every effort must be made to prevent failure or, when a failure occurs, to minimize its negative effects.

OS Open supports the following features for hard real-time applications:

- Rate monotonic scheduling (RMS) analysis

The scheduling policy is optimal for fixed-priority uniprocessor systems.

- Deterministic execution

Non-deterministic behavior can be minimized or eliminated.

- Bounded execution

Most functions in the real-time are implemented using constant-time algorithms.

The OS Open programming features supporting hard real-time systems can be applied to real-time systems having less stringent requirements, and to high-performance embedded systems.

RMS Analysis

The real-time executive implements the fixed-priority scheduling algorithms defined in POSIX 4a.

Priorities are assigned so that the thread having the shortest period is assigned the highest priority, and threads having longer periods are assigned correspondingly lower priorities. This scheduling policy, called rate monotonic scheduling (RMS), is described in a seminal paper on real-time scheduling[†].

A least upper bound on processor utilization can be computed for RMS systems. The exact schedulability of an RMS system can be determined analytically if the periodicity and worst-case run times for each thread in the system are known[‡].

[†] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment," *J.ACM*, Vol. 20, No. 1, January 1973, pp. 46–61.

[‡] Sha, L. and Goodenough, J.B. "Real-Time Scheduling Theory and Ada," *Computer*, Vol. 23, No. 4, April 1990, pp. 53-62.

Deterministic Execution

The real-time executive minimizes the effects of certain features of the PowerPC Architecture that can introduce non-deterministic program timing. The following architecture features complicate the calculation of worst-case thread run times:

- Translation look-aside buffer (TLB) misses
- Page faults
- Cache misses

OS Open threads execute with address translation disabled; TLB misses and page faults do not affect thread execution.

Some PowerPC processors provide mechanisms to disable caching. For such processors, the real-time executive can enable or disable caching under program control. Hard real-time applications can run with caching disabled, while high-throughput applications can run with caching enabled.

Bounded Execution

RMS analysis requires the determination of worst-case execution times. Such a calculation is possible only if the tasks performed by the threads have bounded execution times.

If the execution time of an algorithm is a function of input size ($O(n)$ and $O(\log(n))$ algorithms), worst-case execution times must be determined using the maximum input size.

If the execution time of an algorithm is independent of input size ($O(1)$ algorithms), worst-case, average case, and best-case times are equal. Worst-case execution times of $O(1)$ algorithms, called constant-time algorithms, can be determined at any input size.

Constant-time algorithms simplify the determination of worst-case thread execution times, which simplifies RMS analysis.

The real-time executive provides constant-time functions whenever possible. Interthread communication, thread synchronization, and thread serialization functions are constant-time, as are functions for allocating and freeing fixed-length buffers.

Table 5-1 on page 5-3 lists the real-time executive functions that have unbounded execution times:

Table 5-1. Real-Time Executive Functions Having Unbounded Execution Times

Function	Cause of Unboundedness
alarm()	Number of armed timers
memheap_alloc()	Searching for free blocks
memheap_alloc_aligned()	Searching for free blocks
memheap_alloc_pages()	Searching for free blocks
memheap_query()	Number of free blocks
memheap_realloc()	Searching for free blocks
mempool_init()	Number of blocks to initialize
mq_open()	Name search, memory allocation
mq_close()	Name search
mq_unlink()	Name search
mq_send()	Message size
mq_receive()	Message size
nanosleep()	Number of timers
package_install()	Name search
pthread_cond_broadcast()	Number of waiting threads
pthread_cond_timedwait()	Number of timers
pthread_create()	Memory allocation
pthread_mutex_lock()	Priority inheritance chain caused by successive priority inversions
pthread_mutex_setprioceiling()	Priority inheritance chain caused by successive priority inversions
pthread_setschedparam()	Priority inheritance chain caused by successive priority inversions
sem_init()	Memory allocation
sem_destroy()	Number of blocked threads on the semaphore
sem_unlink()	Name search
sem_open()	Name search, memory allocation
sem_close()	Name search

Table 5-1. Real-Time Executive Functions Having Unbounded Execution Times

Function	Cause of Unboundedness
timer_settime()	Number of armed timers
timertick_notify()	Number of armed timers

Chapter 6. Understanding OS Open Threads

In the OS Open operating system, the thread is the basic schedulable unit of execution. To the operating system, a thread appears as a stack and an execution context. An execution context is a system object that stores state information about a thread.

The default stack size for a thread is defined in **pthread.h**. For I/O-intensive applications, the stack size may need to be increased. For programs using remote procedure calls (RPCs), the stack size should be set to 16 KB or larger as required for normal operation.

Threads, which provide multiple flows of control within a program, support the requirements of tightly-coupled real-time applications. Multiple threads running in the single OS Open process can run tasks independently or in concert. Each thread competes for the processor based on its priority.

A thread has a unique thread ID, scheduling priority and policy, timer state, *errno* value, thread-specific key/value bindings, and system resources required to support control flow.

The OS Open thread implementation supports explicit parallelism. Multiple schedulable threads are explicitly created and terminated. The suspension of one thread does not suspend other threads.

Working with OS Open Threads

OS Open thread functions support tasks associated with working with multiple threads, including:

- Creating threads
- Terminating threads
- Scheduling threads
- Serializing threads
- Synchronizing threads
- Using thread attributes objects to establish classes of threads
- Maintaining thread-specific data

Creating Threads

pthread_create() creates and initializes a new thread and stores its identifier.

The thread is initialized using default thread attributes or attributes in a specified thread attributes object, as described in “Using Thread Attributes Objects” on page 7-9.

When a thread is created, it can run a specified routine.

Terminating Threads

pthread_exit terminates the calling thread and makes its return code available to a successful join with the terminating thread.

An implicit call to **pthread_exit()** is made when a thread returns from the start routine that created the thread. The function's return value serves as the thread's exit status.

Any cancelation cleanup handlers on the cleanup routine stack are popped and executed. If the thread has thread-specific data, appropriate destructor functions are called in an unspecified order after all cleanup handlers have run.

Thread termination does not release resources visible to the application, including (but not limited to) mutexes, condition variables, attributes objects and storage.

If the **detachstate** attribute of the thread is **PTHREAD_CREATE_DETACHED**, the terminated thread and its stack are freed. Otherwise, the terminated thread and its stack are not freed unless **pthread_join()** is called when the thread is terminated.

The behavior of **pthread_exit()** is undefined if it is called from a cleanup handler or key destructor.

Canceling Threads

Thread cancelation enables a thread to terminate any other thread in a controlled manner. The thread being terminated may hold cancelation requests pending and perform cleanup routines when a cancelation request is acted upon.

pthread_cancel() issues requests to cancel a thread.

A thread can be canceled only at a cancelation point or when it is asynchronously cancelable. These terms are defined under subsequent headings.

Working with the Cancelability State of a Thread

The ability of a thread to be canceled is its cancelability state. A thread maintains its own cancelability state, which is either enabled or disabled. The cancelability state is set using **pthread_setcancelstate()**.

One of the following constants, **PTHREAD_CANCEL_ENABLE** or **PTHREAD_CANCEL_DISABLE**, specifies the cancelability state. These constants are defined in the file **<pthread.h>**. The default cancelability state is **PTHREAD_CANCEL_ENABLE**.

If the cancelation state is disabled, cancelation requests are held pending for that thread. If the cancelability state is enabled, there are two types of cancelability:

Asynchronous	A cancelation request may be acted upon at any time.
Deferred	Cancelation requests are held pending until a cancelation point is reached.

If the cancelability state is disabled, the cancelability type has no meaning.

A cancelation point occurs when an operation may result in an unbounded wait. For example, the following conditions are cancelation points:

- A thread waits for a condition variable
- A thread waits for another thread to terminate
- A thread waits for a semaphore

pthread_testcancel() creates a cancelation point.

pthread_setcanceltype() sets the cancelability type, which is specified by **PTHREAD_CANCEL_ASYNCHRONOUS** or **PTHREAD_CANCEL_DEFERRED**. These constants are defined in the file `<pthread.h>`. The default cancelability type is **PTHREAD_CANCEL_DEFERRED**.

Using Cleanup Routines

A thread maintains a list of cleanup routines to execute when a cancelation request is acted upon or when the thread terminates normally. The list is in a stack; the cleanup routines are executed in last-in-first-out (LIFO) order.

pthread_cleanup_push() and **pthread_cleanup_pop()** add and remove routines from the list, respectively.

For example, a cleanup routine could free acquired dynamic storage before a thread terminates.

Using Cancel Safe Functions

The cancel safe attribute is meaningful only for threads running in deferred cancelability mode (the cancel state is enabled; the cancel type is deferred).

If a thread running in deferred cancelability mode calls a cancel safe function, the calling thread cannot be canceled before the cancel safe function returns. If the thread calls a non-cancel safe function, the calling thread can be canceled before the non-cancel safe function returns.

To determine whether a function is cancel safe, refer to its description in *OS Open Programmer's Reference*.

Scheduling Threads

Scheduling refers to how an operating system allocates the processor to execution contexts (threads, in OS Open). OS Open uses the local thread scheduling model, in which a single process contains multiple threads. This scheduling model implements the following basic rules:

- A thread runs only when no thread at a higher priority is ready to run.
- When a thread at a higher priority than the running thread is ready to run, the running thread is preempted.
- Once a thread starts to run, it retains control of the processor until the thread switches execution context, the thread blocks, or the thread is preempted by a thread having a higher priority.

A thread has a priority and a scheduling policy that together determine the scheduling of the thread. Priority determines the position in the ready queue of a runnable thread. Scheduling policy determines when and for how long a thread runs when contending for the processor with another thread at the same priority.

Thread priority is represented by an integer value in the inclusive range of **PTHREAD_PRIO_MIN_NP** through **PTHREAD_PRIO_MAX_NP**; these constants are defined in the file `<pthread.h>`. Higher values have higher priorities. The thread in the ready queue having the highest priority receives control of the processor.

pthread_setschedparam() sets the priority and scheduling policy of a thread; **pthread_attr_setschedparam()** and **pthread_attr_setschedpolicy()** set the priority and scheduling policy attributes, respectively, in a thread attributes object. **pthread_getschedparam()** gets the priority and scheduling policy of a thread; **pthread_attr_getschedparam()** and **pthread_attr_getschedpolicy()** get the priority and scheduling policy attributes, respectively, in a thread attributes object.

The OS Open scheduling policies handle processor contention. The policies are priority scheduling, also called first-in-first-out (FIFO), and round-robin scheduling. Scheduling policy is defined at thread creation, either explicitly or by default.

The scheduling policy of a thread is one of the constants **SCHED_FIFO** (FIFO), **SCHED_RR** (round-robin), or **SCHED_OTHER** (reserved for future use; currently set to the same value as **SCHED_FIFO**). These constants are defined in the file `<sched.h>`. **SCHED_RR** and **SCHED_FIFO** scheduling policies can be modified by ORing with the **SCHED_NOPREEMPT** flag. This flag will prevent a running thread from being preempted by a higher priority thread. When **SCHED_RR** and **SCHED_NOPREEMPT** are specified, the only preemption that can occur is when the time quantum of the running thread expires. The **SCHED_NOPREEMPT** option does not affect scheduling decisions in cases when a running thread blocks or voluntarily relinquishes the processor.

If multiple runnable FIFO threads are at the same priority, they are scheduled in the order in which they arrived in the ready queue. Assuming that the threads are not preempted and do not block, each thread runs to completion in the order of arrival.

If multiple runnable round-robin threads are at the same priority, each thread runs for a period (time slice) and is then preempted. Assuming that the threads are not preempted by a thread having a higher priority and do not block, the threads receive control of the processor in turn. Over time, the threads each receive an approximately equal amount of processor time.

The time slice allocated to round-robin threads is configured at OS Open initialization. See Chapter 4, "Configuring the OS Open Operating System," for information about configuring the round-robin time slice.

If a runnable FIFO thread and a runnable round-robin thread are at the same priority, their scheduling is initially determined by their order of arrival in the ready queue. If the FIFO thread arrived first, it runs until it is preempted or blocked. If the round-robin thread arrived first, it runs until it is preempted, blocked, or its time slice expires. If no higher-priority thread arrives, the FIFO thread regains control of the processor.

Serializing Threads

Thread serialization solves the problem caused when multiple threads modify shared data. If a thread reads data that may be concurrently written by another thread, or when a thread tries to write data that another thread may concurrently read, indeterminate results occur.

If each thread sharing a data object can obtain exclusive access to the data when needed, the problem can be solved. A thread can exclude all other threads from modifying a shared data object until such exclusive access is no longer required. Such exclusion is called mutual exclusion.

The real-time executive enforces mutual exclusion using a mechanism called a mutex. A mutex serializes access to shared data by multiple threads, ensuring data integrity when used properly. A thread requiring exclusive access to shared data locks a mutex, which typically remains locked while a small routine accesses the shared data. After the data access, the mutex is unlocked, and other threads can access the data.

The portion of the thread requiring exclusive access is called a critical section. A thread executing its critical section receives exclusive access to the shared data. Any other threads requiring access to the same data must wait, so critical sections should be programmed carefully to ensure quick execution. A thread must not block within a critical section, and the thread must relinquish exclusive access as soon as the data access is complete.

Using Mutexes

A mutex is created using **pthread_mutex_init()**. The new mutex is initialized using default attributes or attributes specified in a mutex attributes object.

The mutex is available to multiple threads to serialize access to a shared data object. In the following example, a thread, called Thread1, requires exclusive access to the shared data.

Thread1 attempts to lock the mutex using **pthread_mutex_lock()**, **pthread_mutex_timedlock()**, or **pthread_mutex_trylock()**. If successful, Thread1 owns the mutex. If another thread already owns the mutex, Thread1 is blocked until the owning thread unlocks the mutex. Thread1 locks the mutex, runs code accessing the shared data, and unlocks the mutex using **pthread_mutex_unlock()**.

When the mutex is no longer needed, **pthread_mutex_destroy()** can be used to destroy it.

Handling Priority Inversions

A priority inversion occurs when a thread is blocked for an unbounded time because a mutex is held by a thread having a lower priority.

Mutex attributes, contained in mutex attributes objects, specify the protocol associated with a mutex to handle priority inversions at mutex creation. The real-time executive implements the following protocols:

Priority inheritance Raises the priority of the lower-priority thread to the priority of the higher-priority thread until the lower-priority thread unlocks the mutex. The priority of the lower-priority thread is then returned to its original value. Priority inversions are bounded by the duration of the critical sections.

Priority ceiling Specifies a priority ceiling, which is typically set to the priority of the highest-priority thread using the mutex. When a thread locks a mutex, the priority of the thread is set to the highest priority ceiling of the mutexes locked by the thread. A lower-priority thread can lock a mutex for no longer than the duration of a critical section.

To avoid priority inversions, the priority ceiling of the mutex should be set to a value greater than or equal to the highest priority of any thread that may lock the mutex.

pthread_mutexattr_setprotocol() and **pthread_mutexattr_getprotocol()** set and get the priority inversion protocol value, respectively, of a mutex attributes object. The protocol is one of the following types; the constants are defined in the file **<pthread.h>**:

NO_PRIO_INHERIT	No priority inversion protocol is used
PRIO_INHERIT	Specifies a mutex supporting priority inheritance
PRIO_PROTECT	Specifies a mutex supporting priority ceilings

If **PRIO_PROTECT** is specified for a mutex, a priority ceiling attribute must be given. A priority ceiling is represented by an integer value in the inclusive range of **PTHREAD_PRIO_MIN_NP** through **PTHREAD_PRIO_MAX_NP**; these constants are defined in the file **<pthread.h>**.

pthread_mutexattr_setprioceiling() and **pthread_mutexattr_getprioceiling()** set and get the priority ceiling value, respectively, of a new mutex attributes object. **pthread_mutex_setprioceiling()** and **pthread_mutex_getprioceiling()** set and get the priority ceiling value, respectively, of an existing mutex attributes object.

Using Mutex Attributes Objects

Mutex attributes are specified using a mutex attributes object, which serves as a template to create a new mutex. Table 6-1 summarizes the mutex attributes.

pthread_mutexattr_init() creates and initializes a new mutex attributes object. The new mutex attribute object has the system default values for the attributes, which are defined in the file **<pthread.h>**.

When a mutex attribute object is no longer needed, it can be destroyed by using **pthread_mutexattr_destroy()**.

Table 6-1. Mutex Attributes Summary

Attribute	Description	Related Functions
Protocol	Specifies the mutex protocol: NO_PRIO_INHERIT , PRIO_INHERIT , or PRIO_PROTECT	pthread_mutexattr_getprotocol() pthread_mutexattr_setprotocol()
Priority Ceiling	Specifies the priority ceiling if PRIO_PROTECT is given; from PTHREAD_PRIO_MIN_NP through PTHREAD_PRIO_MAX_NP	pthread_mutex_getprioceiling() pthread_mutex_setprioceiling() pthread_mutexattr_getprioceiling() pthread_mutexattr_setprioceiling()

Synchronizing Threads

Thread synchronization can prevent unbounded waits for resources and ensure a certain order of thread execution.

The real-time executive implements condition variables and semaphores as mechanisms for thread synchronization.

Understanding Condition Variables

Condition variables are useful when resource contention may result in unbounded waits.

A condition variable is associated with a mutex guarding the critical section containing the condition variable. Together, the condition variable and the mutex ensure exclusive access to a resource in the same way a mutex protects exclusive access to data.

A condition variable blocks a thread unless some condition is met. Typically, the condition depends on the availability of a resource. Using the condition variable ensures that the resource is in a particular state before execution continues. The thread locks the mutex and examines the state of the resource. If the thread cannot continue execution, it blocks on the condition variable. Otherwise, execution of the critical section completes and mutex is unlocked.

Waiting on the condition variable unlocks the mutex. Other threads can compete for the mutex. If a thread changes the state of the resource to satisfy the condition variable, the thread can signal the condition variable to unblock threads blocked on the condition variable.

A condition variable may be associated with a time-out value to prevent unbounded blocking on the condition variable.

Using Condition Variables

pthread_cond_init() creates a condition variable. The mutex associated with the condition variable must exist before subsequent condition variable operations can occur.

pthread_cond_wait() blocks the calling thread on a condition variable.

pthread_cond_timedwait() performs the same function and specifies a time-out value to prevent unbounded blocking on the condition variable.

pthread_cond_signal() unblocks a thread blocked on a condition variable. If more than one thread is blocked, the scheduling policy determines which thread is unblocked. If no threads are blocked, **pthread_cond_signal()** has no effect.

pthread_cond_broadcast() unblocks all threads blocked on a condition variable. If more than one thread is blocked, the scheduling policy determines the order in which the threads are unblocked. An unblocked thread resumes only after it acquires the mutex with which the thread called **pthread_cond_wait()** or **pthread_cond_timedwait()**. Unblocked threads contend for the mutex according to the scheduling policy.

pthread_cond_destroy() destroys a condition variable.

The real-time executive provides two functions for working with condition attributes objects: **pthread_condattr_init()**, which creates and initializes a

condition attributes object, and **pthread_condattr_destroy()**, which destroys a condition attributes object.

Using Semaphores

The real-time executive implements counting semaphores for thread synchronization.

Semaphores are appropriate when the time-out feature of condition variables is not required and no conditional expression is needed to determine the availability of a resource.

For example, in device drivers, a semaphore is often used in the interrupt handler to signal the device driver that some operation may proceed, based on the receipt of an interrupt.

Semaphores are named or unnamed. A semaphore is referred to by name when its descriptor is unknown; once the descriptor is known, the semaphore can be referred to by that descriptor.

sem_open() establishes a connection between a named semaphore and the real-time executive; **sem_init()** initializes an unnamed semaphore. The semaphores may be used on subsequent calls to **sem_post()**, **sem_timedwait()**, **sem_wait()**, and **sem_trywait()**.

Using Thread Attributes Objects

A thread attributes object serves as a template to create a class of threads having identical attributes.

pthread_create() uses a thread attributes object to specify a thread's attributes when the thread is created. Threads created using the same thread attributes object have identical thread attributes.

Before being used, a thread attributes object must be created and initialized using **pthread_attr_init()**. If no value is given for an attribute when **pthread_create()** is called, the real-time executive supplies a system default. Creating a thread using a NULL attributes object applies system defaults to all thread attributes.

Changing a thread attributes object does not affect threads previously created using the object.

pthread_attr_destroy() destroys a thread attributes object.

Table 6-2 summarizes the thread attributes. Constants in the table are defined in the file `<pthread.h>`.

Table 6-2. Thread Attributes Summary

Attribute	Description	Related Functions
stacksize	The size, in bytes, of a thread's stack; must be greater than PTHREAD_STACK_MIN . The system default is 2048.	pthread_attr_getstacksize() pthread_attr_setstacksize()
stackaddr	Points to the space used for the thread stack. The system default is 0.	pthread_attr_getstackaddr() pthread_attr_setstackaddr()
detachstate	Specifies whether a thread can participate in a pthread_join() ; can be set to the values of PTHREAD_CREATE_DETACHED or PTHREAD_CREATE_JOINABLE (the system default)	pthread_attr_getdetachstate() pthread_attr_setdetachstate()
floating point availability	Specifies whether a thread can use floating point registers; can be set to the values of PTHREAD_FP_AVAILABLE_NP and PTHREAD_FP_NOTAVAILABLE_NP (the system default)	pthread_attr_getfp_np() pthread_attr_setfp_np()
priority	Specifies the initial priority of a thread; can be an integer in the inclusive range of PTHREAD_PRIO_MIN_NP through PTHREAD_PRIO_MAX_NP ; the system default is 15	pthread_attr_getschedparam() pthread_attr_setschedparam()
contentionscope	Provided for portability only; can be set to the values of PTHREAD_SCOPE_SYSTEM and PTHREAD_SCOPE_PROCESS (the system default)	pthread_attr_getscope() pthread_attr_setscope()
scheduling policy	Determines the order in which threads receive control of the processor; can be set to the values of SCHED_RR , SCHED_FIFO (the system default), or SCHED_OTHER (same as SCHED_FIFO)	pthread_attr_getschedpolicy() pthread_attr_setschedpolicy()
inherit schedule	Specifies whether a thread inherits the scheduling policy of its parent thread or uses the default scheduling policy; can be set to the value of PTHREAD_INHERIT_SCHED or PTHREAD_EXPLICIT_SCHED (the system default)	pthread_attr_getinheritsched() pthread_attr_setinheritsched()

Working with Thread-Specific Data

The real-time executive provides keys to maintain thread-specific data. Keys are bound to key values, which are opaque objects serving as identifiers for thread-specific data. A key is available to all threads in the system; values are bound to the key on a per-thread basis and persist for the life of a thread.

For example, an application may provide a common buffer pool to several threads. Each thread uses one key to maintain a list of buffers allocated by the thread. The key has several values, each associated with the active buffer list for a particular thread.

pthread_key_create() creates a new key that is available to all threads in the system. After a key is created, **pthread_setspecific()** sets the thread-specific value, which is retrieved using **pthread_getspecific()**.

An optional destructor function can be associated with the key at key creation. The destructor function is called at thread termination if the key value is non-NULL. A typical use for a destructor function is the freeing of system resources allocated to the calling thread.

pthread_key_delete() deletes a thread-specific key previously returned by **pthread_key_create()**.

The maximum number of keys that can be created is obtained using **sysconf()**. After the maximum is reached, **pthread_key_create()** calls fail until one or more keys are deleted.

Chapter 7. Writing OS Open Interrupt and Fault Handlers

In OS Open applications, interrupt and fault handlers perform some processing in response to events that interrupt normal program execution.

In this chapter, the term “interrupt handler” refers to code that responds to interrupts from peripheral devices; such interrupts are often called “external interrupts.” The term “fault handler” refers to code that responds to processor interrupts, such as those listed in Table 7-1 on page 8-2. Unless this distinction is important, “interrupt handler” refers to both interrupt and fault handlers.

The OS Open operating system performs most of the housekeeping tasks required for interrupt processing. For example, the operating system saves registers and starts execution of an interrupt handler. When the interrupt handler finishes, the operating systems restores registers and schedules the thread to run next.

To specify an interrupt handler, an **flih_t** structure, defined in the file **<flih.h>**, is initialized with the function pointer of the interrupt handler, a user-defined stack, and a value passed as the first parameter to the interrupt handler. This value can be used to anchor control information needed in subsequent processing and is useful when the defining logic and the interrupt handler are not in the same scope.

Two OS Open functions, **int_install()** and **int_query()**, support interrupt handlers:

int_install() Installs a new interrupt handler. **int_install()** optionally returns the address of an **flih_t** structure already installed for the interrupt, allowing an application to cascade interrupt handlers. To remove an interrupt handler, **int_install()** is given a NULL argument for the **flih_t** structure.

int_query() Returns the **flih_t** address of the FLIH for a specified event.

Application developers specify the action to be taken on receipt of an interrupt in a first-level interrupt handler (FLIH). The operating system calls a FLIH when an interrupt occurs, if one is provided for the received interrupt. Interrupt handlers can be written in C and assembler language.

OS Open interrupt handler support minimizes the path length between interrupt receipt and interrupt handler execution.

Using OS Open Interrupt Handlers

Interrupt handlers provide a mechanism for sensibly dealing with events occurring outside of the direct control of an application. Such events can be as varied as the arrival of data over a communications channel, the failure of a device, or a processor interrupt such as those listed in Table 7-1 on page 8-2.

A FLIH for an external interrupt is tightly coupled to the peripheral device generating the interrupt. For many devices, FLIHs are part of the device driver. The behavior and structure of a FLIH for an external interrupt are typically dictated by the peripheral device generating the interrupt.

For a specific processor, application developers can provide a FLIH for all processor interrupts. Refer to appropriate PowerPC processor publications for descriptions of the processor interrupt set, the contents of special-purpose registers (SPRs) during a specific interrupt, and appropriate application action in response to a specific interrupt

Table 7-1. PowerPC Interrupt Types

Interrupt Type	Description
System reset	Occurs when a reset pin on the processor is asserted.
Machine check	Generally unrecoverable; causes are implementation-dependent.
Data Storage	Caused by address translation or protection errors when accessing data.
Instruction Storage	Occurs when address translation fails while fetching instructions.
External	Occurs when the external interrupt pin on the processor chip is asserted.
Alignment	Caused by unaligned storage access, or an attempt to use the PowerPC dcbz instruction against a noncached region.
Program (floating point)	Caused by invalid floating point operations, such as division of 0 by 0.
Program (illegal opcode)	Taken when the processor executes an instruction containing an illegal instruction opcode or combination of opcode and extended opcode.
Program (privileged instruction)	Taken when the processor executes a privileged instruction while the processor is in user mode (MSR[PR] is 1)
Program (trap)	Caused by a successful trap instruction.
Floating Point Unavailable	Caused by an attempt to perform a floating-point operation when the floating-point enable bit (MSR[FP]) is 0.
Decrementer	Caused by the transition of the most significant bit of the decrementer from 0 to 1; must be enabled using the MSR[EE] bit.

Note: In the PowerPC Architecture, the machine check and system reset interrupts are unordered and imprecise. Software cannot usually recover from these interrupts. If an FLIH for one of these interrupts returns, normal processing may not be possible.

FLIHs cannot be debugged using the supplied breakpoint functions or the shell.

Running Interrupt Handlers

All OS Open events that can be installed to handle processor exceptions are divided in two categories: external interrupts and faults. Handlers for both categories of events are installed using `int_install()` function. Include file `<flih.h>` lists all the events that are available and specifies if the event is treated as an external interrupt or fault.

When the OS Open real-time executive passes control to an interrupt handler, external and decremter interrupts are disabled, and the first argument passed to the interrupt handler is set to the value specified when the interrupt handler was installed using `int_install()`. Values returned from the interrupt handler are not checked.

In a fault handler, similarly, the first parameter is the user-specified value. The second parameter is set to the address of a register image defined in the file `<dbLib.h>`. This allows fault handlers to examine and alter the processor state before returning.

The return value of a fault handler determines subsequent processing. If the return value is non-zero (the usual case), the machine state is restored from the register image and control returns to the point of interruption.

If the return value is 0, the oldest, highest-priority thread is dispatched. This allows the system to run while retaining the ability to examine the state of a thread encountering a fatal fault or exception. For example, a trap handler can easily suspend a thread that has encountered a breakpoint.

,During system calls (For those processors that have critical exceptions) OS Open disables critical exceptions to preserve consistency of kernel data areas. If application requires that critical exceptions are processed without any delay system call entry point can be changed from `__System_call` to `__System_call_nocritical`, so that critical exceptions are not disabled during system call processing. This can be accomplished by writing branch instruction at system call exception vector that will branch to `__System_call_no critical` function instead of `__System_call`. If system calls are handled by `__System_call_nocritical` function critical exception routines can not make any system calls.

Note: The OS Open operating system does not support reentrant FLIHs. Do not reenale external interrupts inside an FLIH by direct machine state register (MSR) manipulation.

Note: Critical exceptions on some processors (such as 403GA) must be handled differently from other exceptions. If such an exception occurs when external interrupts are disabled (this can be verified by examining MSR in the register

image that is passed to the fault handler) then the fault handler must return value non-zero so that machine state is restored from register image. If nested external interrupts are being used, MSR check is not sufficient to determine if the critical interrupt occurred during processing of exception. In this case, in addition to checking MSR `int_getid()`, function must be called to verify that external interrupts are not running.

Interrupt Handler Safe Functions

An interrupt handler safe function may be called by a FLIH. OS Open functions used in an FLIH must not suspend execution or be valid only in the context of a running thread.

To determine whether a function is interrupt handler safe, refer to the description of the function in *OS Open Programmer's Reference*.

Chapter 8. Writing OS Open Device Drivers

A device driver is some code that supports a device. OS Open device drivers hide the details of device operation from application programs. Programs can read and write disk files, for example, without issuing disk controller commands.

OS Open device drivers are not part of the OS Open real-time executive but are treated as applications. OS Open device drivers are integrated into the OS Open file system as special files, and can be installed onto a running OS Open operating system.

Each device available to an application program has a special file associated with it that is a system interface with which the device driver is invoked. This special file can be accessed as an ordinary file; it can be opened, read, written, and manipulated using OS Open functions.

One OS Open device driver can support multiple devices. For example, a device driver for a diskette controller may support two diskette drives. Although one device driver controls all drive operations, each drive appears as a separate device to an application program.

Appendix B, “Sample Device Drivers,” lists a sample character device driver (asynchronous device) and a sample block device driver (diskette). Appendix B also contains the skeleton source code for a SCSI adapter device driver.

Device Driver Types

The OS Open operating system supports the following types of device drivers:

- Character, supporting devices that manage byte streams, such as serial ports, keyboards, and printers.

For detailed information about writing character device drivers, see “Writing Character Device Driver Functions” on page 8-5.

- Block, supporting random access storage devices that manage fixed-size addressable data blocks, such as fixed disk drives.

For detailed information about writing block device drivers, see “Writing Block Device Driver Functions” on page 8-7.

- Logical file system, supporting interfaces between applications and underlying media (real or simulated) containing regular files, such as the OS Open DOS logical file system.

For detailed information about logical file systems, see Chapter 9, “Writing Logical File Systems.”

- Terminal device drivers, which are not described in this chapter, are internal to the OS Open operating system.

Implementing OS Open Device Drivers

To implement device drivers, developers provide the following device-specific functions to support device operations:

<code>dev_init</code>	Initializes a device
<code>dev_config</code>	Configures a device
<code>dev_open</code>	Opens a file descriptor for a device
<code>dev_close</code>	Disassociates a file descriptor and a device
<code>dev_read</code>	Reads from the device
<code>dev_write</code>	Writes data to the device
<code>dev_ioctl</code>	Performs device-specific I/O commands
<code>dev_select</code>	Queries a device for read, write, or exceptions
<code>dev_strategy</code>	Performs block-oriented transactions (block devices only)

Each device driver provides an initialization function that implements the installation of the device driver. This function should provide entry points for the device-specific functions to the real-time executive when a device is installed.

Each type of device driver implements device-specific functions slightly differently.

Installing and Initializing Device Drivers

driver_install(), which installs all device driver types, takes as its arguments a pointer to a device handle, the driver initialization function, and any driver-specific parameters unique to the device.

OS Open device support then calls the user-supplied driver initialization function *dev_init()* using the following prototype:

```
int dev_init(driver_t *dsw, va_list vargs);
```

dsw points to a device switch table, which must be filled with the entry points of the device-specific functions listed under the preceding heading.

vargs passes, in the variable argument list, optional parameters supplied to **driver_install()**.

At the least, *dev_init()* must fill the device switch table with the device-specific function pointers. **dev_init()** may also perform any device-wide initialization or configuration at this time. When **dev_init()** returns, device support assigns a device handle, which is returned to the thread calling **driver_install()**.

device_install() binds a device driver to a named special file. **device_install()** parameters are the device handle returned by a corresponding **driver_install()**, the special file name of the device, and device-specific optional parameters, if any.

OS Open device support then calls a user-supplied device driver configuration function *dev_config()* (from the device switch table) using the following prototype:

```
int dev_config(void **dds, va_list vargs);
```

dds anchors device-specific information. Its value is retained in the file system and is supplied, using the device-specific file block, to subsequent device driver calls.

vargs passes in the variable argument list optional parameters supplied to **device_install()**. When **device_install()** completes, a subsequent **open()** can access the device.

A simple device driver supporting only one physical device may not require any processing in *dev_config()*.

Note for character and block devices:

If the device driver supports multiple devices per driver instance (that is, multiple **device_install()** functions can be issued using the same device handle returned from **driver_install()**), the device-specific anchor set during device configuration must be unique for each device.

Uninstalling a Device

device_uninstall() removes an installed device from usage. The parameter passed to **device_uninstall()** is the special file name of the device. OS Open device support then calls a user-supplied device driver configuration function with the first variable argument set equal to -1. This function is the same function that is called in response to a **device_install()** call and has the following prototype:

```
int dev_config(void **dds, va_list vargs);
```

dds points to the device-specific anchor that was filled in during the device install. If the *dev_config()* function determines that the first variable argument is -1, it should determine if the device is in use by comparing the number of *dev_open()* calls to the number of *dev_close()* calls against the device. If they are equal, the *dev_config()* function should destroy all objects (such as semaphores and mutexes), uninstall all interrupt handlers, and free all memory associated with the device.

Since **device_install()** and **device_uninstall()** both call the *dev_config()* function, the **device_install()** call must include at least one optional parameter which will never be set equal to -1.

Understanding File Blocks

The file block, defined in **<sys/devDriver.h>**, provides the basic transaction structure for many device driver functions:

```

/* file block */

typedef struct file_block
{
    struct file_block *f_filef;      /* pointer to next open file */
    struct file_block **f_fileb;     /* pointer to previous open file */
    unsigned short file_mode;        /* mode of this file */
    unsigned short file_type;        /* type of the file */
    struct driver_table *dt_ptr;      /* pointer to driver table */
    off_t file_offset;               /* current file position */
    int ref_count;                   /* number of descriptors */
    void *driver_specific;           /* driver specific anchor */
    int fblk_busy;                   /* busy count for this file block */
    int fblk_state;                  /* state flags for file block */
    short f_msgcount;                /* references from message queue */
    int *seltest_mask;               /* mask for mutex locked select retry */
} file_block_t;

```

The number of file blocks available is set during OS Open initialization. Opening a file in a logical file system uses a file block instantiation, up to the number of file blocks set during initialization.

Most file block fields are used by OS Open device support for internal file management. The following fields are of particular interest to device driver developers:

file_mode The mode in which a file is opened (**O_RDWR**, **O_RDONLY**, **O_WRONLY**)

file_type File type from those listed in the following **#define** statements, which precede the file block structure:

```

/* file types */
#define REGTYPE '0'                /* regular file */
#define LNKTYPE '1'                /* line */
#define SYMTYPE '2'                /* reserved */
#define CHRTYPE '3'                /* character special */
#define BLKTYPE '4'                /* block special */
#define DIRTYPE '5'                /* directory */
#define FIFOTYPE '6'                /* FIFO special */
#define CONTTYPE '7'                /* reserved */
#define LFSTYPE 'A'                /* logical file system */
#define SHMOTYPE 'B'                /* shared memory object file */
#define DTYPE_SOCKET 'C'           /* socket */
#define DTYPE_TTY 'D'              /* Terminal device */

```

file_offset For regular files, the current file offset

ref_count The number of descriptors pointing to the file block

driver_specific

The *dds* value provided by *dev_config()*.

Writing Character Device Driver Functions

User-supplied functions for character device drivers support devices that produce and consume data as byte streams, such as serial ports, printers, keyboards, and terminals.

Descriptions of the user-supplied functions for character device drivers follow. The function descriptions include a prototype and descriptions of the associated parameters.

Portions of a serial port driver follow in Appendix B, "Sample Device Drivers." The portions provide examples of user-supplied functions. Note that the sample code is device-specific, and may not meet any user requirements.

Device-Specific Open

A call to the OS Open function **open()** results in a call to a device-specific function having the following prototype:

```
int dev_open(file_block_t *fb, int oflag, va_list vars);
```

fb points to the initialized file block.

oflag is passed unchanged from **open()**.

vars passes additional parameters supplied to the **open()** variable argument list.

If successful, *dev_open()* returns 0. Otherwise, *dev_open()* returns -1 and sets *errno* if appropriate.

There is no required processing for this function, although many drivers use this function to initialize the associated device.

OS Open device subsystem assigns only O_ACCMODE flags to the *file_mode* field in the *file_block_t* pointer during execution of the *open()* function. Other open mode flags such as O_NONBLOCK, O_TRUNC must be assigned to the *file_mode* field by the device driver if these flags are used by the device driver.

Device-Specific Close

A call to the OS Open function **close()** results in a call to a device-specific function, if no other descriptors are open against the device. The device-specific function has the following prototype:

```
int dev_close(file_block_t *fb);
```

fb points to the initialized file block.

If successful, *dev_close()* returns 0. Otherwise, *dev_close()* returns -1.

There is no required processing for this function, although many drivers use this function to disable the associated device.

Device-Specific Read

A call to the OS Open function **read()** results in a call to a device-specific function having the following prototype:

```
int dev_read(file_block_t *fb, void *buffer, size_t length);
```

fb points to the initialized file block.

buffer points to the application buffer supplied to **read()**.

length is passed unchanged from **read()**.

If successful, *dev_read()* returns the number of bytes transferred. Otherwise, *dev_read()* returns `-1` and sets *errno* if appropriate.

dev_read() moves *length* bytes of device data into the buffer pointed to by *buffer*. If the buffer is not filled, typically *dev_read()* blocks, unless **O_NONBLOCK** is supported by the device driver and was specified at device open. To block, semaphores are recommended, but condition variables can be used.

Device-Specific Write

A call to the OS Open function **write()** results in a call to a device-specific function having the following prototype:

```
int dev_write(file_block_t *fb, void *buffer, size_t length);
```

fb points to the initialized file block.

buffer points to the application buffer supplied to **write()**.

length is passed unchanged from **write()**.

If successful, *dev_write()* returns the number of bytes transferred. Otherwise, *dev_write()* returns `-1` and sets *errno* if appropriate.

dev_write() moves *length* bytes of data from the area pointed to by *buffer* to the device. If the device cannot immediately accept *length* bytes, typically *dev_write()* blocks, unless **O_NONBLOCK** is supported by the device driver and was specified at device open. To block, semaphores are recommended, but condition variables can be used.

Device-Specific I/O Control

A call to the OS Open function **ioctl()** results in a call to a device-specific function having the following prototype:

```
int dev_ioctl(file_block_t *fb, int cmd, va_list vargs);
```

fb points to the initialized file block.

cmd is passed unchanged from **ioctl()**.

vargs passes additional parameters supplied to the **ioctl()** variable argument list.

If successful, *dev_ioctl()* returns 0. Otherwise, *dev_ioctl()* returns `-1` and sets *errno* if appropriate.

ioctl() provides flexible device-specific I/O processing. *cmd* specifies a device-specific command, which can be passed optional parameters. For example, an **ioctl()** command can query modem status or send a Break character on a serial port.

Device-Specific Select

A call to the OS Open function **select()** results in a call to a device-specific function having the following prototype:

```
int dev_select(file_block_t *fb, int *flags);
```

fb points to the initialized file block.

flags is set by OS Open device support to any combination of the following flags: **SEL_ASYNC**, **SEL_READ**, **SEL_WRITE**, **SEL_EXCP**.

If successful, *dev_select()* returns 0. Otherwise, *dev_select()* returns `-1`.

dev_select() must examine the flags and return the conditions that are true at the time of the call. For example:

- If **SEL_READ** is set, and characters are available to be read, **SEL_READ** remains set. Otherwise, the flag is cleared.
- If **SEL_WRITE** is set and the device can accept characters, **SEL_WRITE** remains set. Otherwise, the flag is cleared.
- If the **SEL_EXCP** flag is set and there are outstanding exceptions, the flag remains set.
- If no specified conditions are true at the time of the call, and **SEL_ASYNC** is set, the device driver must note that an asynchronous select request was made. If the specified condition becomes true, the device driver calls **select_notify()**, supplying the file block pointer of the initial **select()** and the updated flag value.

dev_select() must not call **select()**. If **select()** processing is needed within *dev_select()*, *dev_select()* should call **select_redrive()** instead. For more information on **select_redrive**, see the *OS Open Programmer's Reference*.

Writing Block Device Driver Functions

User-supplied functions for block device drivers support random access storage devices having fixed-length blocks, such as diskette and fixed disk drives.

Descriptions of the user-supplied functions for block device drivers follow. The function descriptions include a prototype and descriptions of the associated parameters.

Portions of a diskette driver follow in Appendix B, "Sample Device Drivers." The portions provide examples of user-supplied functions. Note that the sample code is device-specific, and may not meet any user requirements.

The major difference between character and block device drivers is the use of **strategy()**, coupled with the lack of use of **read()**, **write()** and **select()**. **strategy()** specifies a block number for a transaction. **read()**, **write()** and **select()** should return `-1` and set *errno* to `EINVAL`.

Device-Specific Open

A call to the OS Open function **open()** results in a call to a device-specific function having the following prototype:

```
int dev_open(file_block_t *fb, int oflag, va_list args);
```

fb points to the initialized file block.

oflag is passed unchanged from **open()**.

args passes additional parameters supplied to the **open()** variable argument list.

If successful, *dev_open()* returns 0. Otherwise, *dev_open()* returns `-1` and sets *errno* if appropriate.

There is no required processing for this function, although many drivers use this function to initialize the associated device.

OS Open device subsystem assigns only `O_ACCMODE` flags to the *file_mode* field in the *file_block_t* pointer during execution of the **open()** function. Other open mode flags such as `O_NONBLOCK`, `O_TRUNC` must be assigned to the *file_mode* field by the device driver if these flags are used by the device driver.

Device-Specific Close

A call to the OS Open function **close()** results in a call to a device-specific function, if no other descriptors are open against the device. The device-specific function has the following prototype:

```
int dev_close(file_block_t *fb);
```

fb points to the initialized file block.

If successful, *dev_close()* returns 0. Otherwise, *dev_close()* returns `-1`.

There is no required processing for this function, although many drivers use this function to disable the associated device.

Device-Specific I/O Control

A call to the OS Open function **ioctl()** results in a call to a device-specific function having the following prototype:

```
int dev_ioctl(file_block_t *fb, int cmd, va_list vargs);
```

fb points to the initialized file block.

cmd is passed unchanged from **ioctl()**.

vargs passes additional parameters supplied to the **ioctl()** variable argument list.

If successful, *dev_ioctl()* returns 0. Otherwise, *dev_ioctl()* returns `-1` and sets *errno* if appropriate.

ioctl() provides flexible device-specific I/O processing. *cmd* specifies a device-specific command, which can be passed optional parameters. For example, *cmd* may eject a CD-ROM or query disk drive capacity.

Device-Specific Strategy

A call to the OS Open function **strategy()** results in a call to a device-specific function having the following prototype:

```
int dev_strategy(file_block_t *fb, block_req_t *req_block);
```

fb points to the initialized file block.

req_block points to a request block, which contains the address of a buffer in user space, the block number of the request, and the type of request (read or write).

If successful, *dev_strategy()* returns 0. Otherwise, *dev_strategy()* returns `-1` and sets *errno* if appropriate.

The device driver performs a device-specific action to access the requested block number before performing the requested operation.

Performance Considerations

For optimal performance and flexibility, a device driver must ensure its internal state consistency. OS Open device support protects file block parameters, but a device driver must provide whatever additional protection is required.

For example, if a device supports read and write operations, two threads could simultaneously try to read and write the device. OS Open device support ensures that the file block is not invalidated by a **close()** while the two operations are pending. However, coordination of the file offset and driver-specific area manipulation is the responsibility of the device driver.

File or device sharing is the responsibility of the application. Individual device drivers may or may not support file or device sharing.

Chapter 9. Writing Logical File Systems

In the OS Open operating system, logical file systems are implemented as specialized character device drivers. Logical file system drivers do not directly manage a device. Instead, logical file system drivers provide an interface to an underlying device.

For example, a logical file system implementing a DOS file allocation table (FAT) file system can be used with disk and diskette block device drivers to provide named regular file support for OS Open. **read()** and **write()** receive data from and send data to the FAT file system, respectively. The FAT file system may access a physical device to obtain and store the data contained in the files.

Implementing OS Open Logical File Systems

To implement device drivers, developers provide the following device-specific functions to support device operations:

<code>init</code>	Initializes a device
<code>config</code>	Configures a device
<code>open</code>	Opens a file descriptor for a device
<code>close</code>	Disassociates a file descriptor and a device
<code>read</code>	Reads from the device
<code>write</code>	Writes data to the device
<code>ioctl</code>	Performs device-specific I/O commands
<code>select</code>	Queries a device for read, write, or exceptions

The OS Open function **strategy()** is not used in logical file systems, and should simply return `-1`.

Each device driver provides an entry point that implements the installation of the device driver. Entry points for device-specific functions are provided to the real-time executive when a device driver is initialized.

Each type of device driver implements device-specific functions slightly differently.

Installing Logical File Systems

See “Installing and Initializing Device Drivers” on page 8-2 for information about installing device drivers, including logical file systems.

The name supplied to **device_install()** forms the prefix of fully qualified file names managed by the logical file system.

Writing Logical File System Functions

User-supplied functions for logical file systems can support POSIX 1003.1b file systems to the limits imposed by underlying directory and file structures.

The following descriptions of logical file system functions, particularly **ioctl()**, provide information needed to write an OS Open logical file system.

To write a strictly compliant POSIX file system, you may need more information. For additional details about the POSIX file model, refer to the publications listed in “Standards” on page xxiv, particularly POSIX 1003.1b.

Descriptions of user-supplied functions for logical file systems follow.

Driver-Specific Open

A call to the OS Open function **open()** results in a call to a driver-specific function having the following prototype:

```
int drv_open(file_block_t *fb, int oflag, va_list args);
```

fb points to the initialized file block.

oflag is passed unchanged from **open()**.

args passes additional parameters supplied to the **open()** variable argument list.

The first additional parameter is a character pointer to the path name specified in **open()**, with the file system qualifier removed. For example, if a logical file system is named **/mylfs**, and the driver-specific open was against the regular file **/mylfs/sample.txt**, the pointer would point to the string “sample.txt”. This name locates the file on the underlying device.

If successful, *drv_open()* returns 0. Otherwise, *drv_open()* returns -1 and sets *errno* if appropriate.

Driver-Specific Close

A call to the OS Open function **close()** results in a call to a driver-specific function having the following prototype:

```
int drv_close(file_block_t *fb);
```

fb points to the initialized file block.

If successful, *drv_close()* returns 0. Otherwise, *drv_close()* returns -1.

There is no required processing for *drv_close()*, although many logical file systems deallocate internal resources and update file modification times.

Driver-Specific Read

A call to the OS Open function **read()** results in a call to a driver-specific function having the following prototype:

```
int drvr_read(file_block_t *fb, void *buffer, size_t length);
```

fb points to the initialized file block.

buffer points to the application buffer supplied to **read()**.

length is passed unchanged from **read()**.

drvr_read() moves *length* bytes of device data into the buffer pointed to by *buffer*. Logical file systems primarily interpret device data and do not generally block, so synchronization is often left to the underlying physical device drivers.

If successful, *drvr_read()* returns the number of bytes transferred. Otherwise, *drvr_read()* returns -1 and sets *errno* if appropriate.

drvr_read() must properly maintain the file offset pointer in the file block.

Driver-Specific Write

A call to the OS Open function **write()** results in a call to a driver-specific function having the following prototype:

```
int drvr_write(file_block_t *fb, void *buffer, size_t length);
```

fb points to the initialized file block.

buffer points to the application buffer supplied to **write()**.

length is passed unchanged from **write()**.

If successful, *drvr_write()* returns the number of bytes transferred. Otherwise, *drvr_write()* returns -1 and sets *errno* if appropriate.

drvr_write() moves *length* bytes of data in the buffer pointed to by *buffer* to the device. Logical file systems primarily interpret device data and do not generally block, so synchronization is often left to the underlying physical device drivers.

drvr_write() must properly maintain the file offset pointer in the file block.

Driver-Specific I/O Control

A call to the OS Open function **ioctl()** results in a call to a driver-specific function having the following prototype:

```
int drvr_ioctl(file_block_t *fb, int cmd, va_list vargs);
```

fb points to the initialized file block.

cmd is passed unchanged from **ioctl()**.

vargs passes additional parameters supplied to the **ioctl()** variable argument list.

ioctl() is not directly called by application programs to support logical file systems. Instead, logical file systems use **ioctl()** to implement commands, specified by *cmd*, to support the regular files and directories used by application programs. The commands are in turn used by other OS Open functions which application programs can then call.

The user-supplied commands needed to support logical file systems are described under the following headings.

The input parameters shown for each command represent parameters that are retrieved using the OS Open variable argument list macros.

If successful, *drv_ioctl()* returns 0. Otherwise, *drv_ioctl()* returns -1 and sets *errno* if appropriate.

OPENDIR

Input parameter: **char *dirname**

OPENDIR, used by **opendir()**, opens a directory.

OPENDIR should locate the directory named by *dirname* and return an identifier locating the directory stream for subsequent **REaddir**, **REaddir_R**, and **REWINDDIR** I/O control commands.

If directories are implemented as regular files containing directory entries in a special format, the application program implementing **OPENDIR** can simply call **open()**.

If unsuccessful, **OPENDIR** should return -1 and setting *errno* to the appropriate value:

[EACCES]	Search permission denied for a component of the path, or read permission denied for the directory.
[ENAMETOOLONG]	A path name component length exceeds { NAME_MAX } or the length of <i>dirname</i> exceeds { PATH_MAX }.
[ENOENT]	<i>dirname</i> does not exist.
[ENOTDIR]	A component of <i>dirname</i> is not a directory.

REaddir

Input parameter: **struct dirent **direntpp**

REaddir, used by **readdir()**, reads a directory entry.

REaddir locates, reads, and converts a directory entry into POSIX format. The structure **dirent**, defined in the file **<dirent.h>**, illustrates how the supported fields correspond to an actual directory entry.

READDIR must update locator information so that subsequent **READDIR** calls return sequential directory entries. If directories are regular files, the file offset pointer is sufficient.

If successful, **READDIR** returns a pointer to a **struct dirent** containing the directory information. This structure must be allocated internally, and its address supplied to the application in the updated pointer *direntpp*.

Otherwise, **READDIR** should return -1 and set *errno* to the following value:

[EBADF] *dirp* does not point to an open directory stream.

READDIR_R

Input parameter: **struct dirent *direntp**

READDIR_R, used by **readdir_r()**, reads a directory entry.

READDIR_R, a reentrant version of the **READDIR** I/O control command, returns a pointer to a user-supplied **struct dirent**.

Most applications implement **READDIR_R** and reuse its logic to implement **READDIR** by supplying a pointer to an internally allocated **struct dirent**.

If unsuccessful, **READDIR_R** should return -1 and set *errno* to the following value:

[EBADF] *dirp* does not point to an open directory stream.

REWINDDIR

Input parameters: None

REWINDDIR, used by **rewinddir()**, rewinds a directory entry.

REWINDDIR resets the directory offset to 0. If the directory is implemented as a regular file, this is equivalent to setting the *file_offset* to 0.

REWINDDIR does not report errors.

LINK

Input parameters: **char *existing_name, char *new_name**

LINK, used by **link()**, links directory entries.

If the file system directory structure supports the notion of links, **LINK** establishes *new_name* as a link to the file specified by *existing_name*.

If unsuccessful, **LINK** should return -1 and set *errno* to one of the following values:

[EACCES] Search permission denied for a component of the path, or the requested link requires writing while write permission is denied.

[EEXIST]	The link named by <i>new_name</i> already exists.
[EMLINK]	The number of links to the existing file would exceed {LINK_MAX} .
[ENAMETOOLONG]	A path name component length exceeds {NAME_MAX} or the length of the existing or new file name exceeds {PATH_MAX} .
[ENOENT]	<i>existing_name</i> does not exist, or <i>new_name</i> points to an empty string.
[ENOSPC]	The directory containing the link could not be extended.
[ENOTDIR]	A component of <i>existing_name</i> or <i>new_name</i> is not a directory.
[EPERM]	The file named by <i>existing_name</i> is a directory.
[EROFS]	Requested link requires writing in a directory on a read-only file system.
[EXDEV]	The requested link would be between file systems.

UNLINK

Input parameters: **char *file_name**

UNLINK, used by **unlink()**, unlinks directory entries.

If the file pointed to by *file_name* is not open, its directory entry should be removed and the file deleted. If the file is open, its directory entry is removed, but removal of file contents does not occur until all file references are closed.

If unsuccessful, **UNLINK** should return **-1** and set *errno* to one of the following values:

[EACCES]	Search permission denied for a component of the path, or write permission is denied for the directory containing the file.
[EBUSY]	The directory cannot be unlinked because it is in use. This error is implementation-dependent; some implementations may not have a busy state for a directory.
[ENAMETOOLONG]	A path name component length exceeds {NAME_MAX} or the length of the existing or new file name exceeds {PATH_MAX} .
[ENOENT]	<i>file_name</i> does not exist.
[ENOTDIR]	A component of the path prefix is not a directory.

[EPERM]	The file pointed to by <i>file_name</i> is a directory. This error is implementation-dependent; some implementations may not have a busy state for a directory. Implementations may optionally allow unlink() on directories.
[EROFS]	The requested UNLINK requires writing in a directory on a read-only file system.

MKDIR

Input parameters: **char *dirname, mode_t mode**

MKDIR, used by **mkdir()**, makes a new directory entry.

MKDIR must ensure that the new directory name is valid and does not refer to an existing file or directory.

If unsuccessful, **MKDIR** should return `-1` and set *errno* to one of the following values:

[EACCES]	Search permission denied for a component of the path, or the requested operation requires writing while write permission is denied.
[EEXIST]	The named file already exists.
[EMLINK]	The number of links to the parent directory would exceed {LINK_MAX} .
[ENAMETOOLONG]	A path name component length exceeds {NAME_MAX} or the length of the existing or new file names exceeds {PATH_MAX} .
[ENOENT]	A component of the path prefix does not exist.
[ENOSPC]	The file system has insufficient space to contain the new directory or to extend the parent directory.
[ENOTDIR]	A component of <i>dirname</i> is not a directory.
[EPERM]	The file named by <i>dirname</i> is a directory.
[EROFS]	The parent directory is on a read-only file system.

RMDIR

Input parameters: **char *dirname**

RMDIR, used by **rmdir()**, removes a directory entry.

RMDIR removes a directory only if it is the directory entry. If a directory is open, new entries should not be permitted, but removal of the directory contents should be postponed until the directory is no longer open.

If unsuccessful, **RMDIR** should return `-1` and set *errno* to one of the following values:

[EACCES]	Search permission denied for a component of the path, or the requested operation requires writing while write permission is denied.
[EEXIST]	The path argument names a directory that is not empty.
[ENAMETOOLONG]	A path name component length exceeds {NAME_MAX} or the length of the existing or new file name exceeds {PATH_MAX} .
[ENOENT]	<i>dirname</i> names a directory that does not exist or points to an empty string.
[ENOSPC]	The file system has insufficient space to contain the new directory or to extend the parent directory.
[ENOTDIR]	A component of <i>dirname</i> is not a directory.
[EROFS]	The directory is on a read-only file system.

RENAME

Input parameters: **char *old_name, char *new_name**

RENAME, used by **rename()**, renames a regular file or directory named by *old_name* to *new_name*.

Files can only be renamed to files, and directories to directories. The POSIX standards allow the new and old file names to be in different directories.

If the file or directory pointed to by *new_name* exists, the file or directory is processed as if **unlink()** had been called for it before **rename()**. If the file or directory pointed to by *new_name* file is open, the open descriptor continues to refer to the file, but subsequent **open()** calls return a descriptor to the renamed file.

If unsuccessful, **RENAME** should return `-1` and set *errno* to one of the following values:

[EACCES]	Search permission denied for a component of the path, the requested operation requires writing while write permission denied, or one of the directories containing <i>old_name</i> or <i>new_name</i> denies write permission.
[EBUSY]	For implementations that mark directories as busy, the directory is being used by another function.
[EEXIST]	<i>new_name</i> points to a directory that is not empty.

[EINVAL]	<i>new_name</i> points to a directory path name containing a prefix that names the directory <i>pointed to by old_name</i> .
[EISDIR]	<i>new_name</i> names a directory and <i>old_name</i> names a file.
[ENAMETOOLONG]	A path name component length exceeds { NAME_MAX } or the length of the existing or new file names exceeds { PATH_MAX }.
[EMLINK]	<i>old_name</i> names a directory, and the link count of the parent directory of <i>new_name</i> would exceed { LINK_MAX }.
[ENOENT]	<i>old_name</i> points to a file or directory that does not exist or points to an empty string.
[ENOSPC]	The directory that would contain <i>new_name</i> cannot be extended.
[ENOTDIR]	A component of either path prefix is not a directory, or <i>old_name</i> names a directory and <i>new_name</i> names an existing regular file.
[EROFS]	The directory is on a read-only file system.
[EXDEV]	<i>new_name</i> and <i>old_name</i> are on different file systems.

RF_STAT

Input parameters: **char *file_name**, **struct stat *file_stat**

RF_STAT, used by **stat()**, retrieves file statistics by name.

RF_STAT must locate the file, extract available local directory information, and map the local directory information into appropriate fields in the supplied **stat** structure, which is defined in the file **<stat.h>**. **struct stat** contains file type, file mode, file access times, and other information.

If unsuccessful, **RF_STAT** should return **-1** and set *errno* to one of the following values:

[EACCES]	Search permission is denied for a component of the path.
[ENAMETOOLONG]	A path name component length exceeds { NAME_MAX } or the length of <i>file_name</i> exceeds { PATH_MAX }.
[ENOENT]	<i>file_name</i> names a file that does not exist or points to an empty string.
[ENOTDIR]	A component of the path prefix is not a directory.

RF_FSTAT

Input parameters: **struct stat *file_stat**

RF_FSTAT, used by **fstat()**, retrieves file statistics by descriptor

RF_FSTAT should use file block information to locate the directory entry of a file, extract local directory information, and map the local directory information into appropriate fields in the supplied **stat** structure, which is defined in the file **<stat.h>**. **struct stat** contains file type, file mode, file access times, and other information.

RF_FSTAT does not report any errors.

RF_ACCESS

Input parameters: **char *file_name, int access_mode**

RF_ACCESS, used by **access()**, queries file permissions or existence.

RF_ACCESS must locate the named file, extract its mode from the local directory structure, and compare the reported mode against the supplied access mode.

The mode is the bitwise inclusive OR of one of the following constants, which are defined in **<unistd.h>**:

R_OK	Permission to read
W_OK	Permission to write
X_OK	Permission to execute
F_OK	Simple existence check

If unsuccessful, **RF_ACCESS** should return **-1** and set *errno* to one of the following values:

[EACCES]	Search permission denied for a component of the path.
[ENAMETOOLONG]	A path name component length exceeds {NAME_MAX} or the length of <i>file_name</i> exceeds {PATH_MAX} .
[ENOENT]	<i>file_name</i> names a file that does not exist or points to an empty string.
[ENOTDIR]	A component of the path prefix is not a directory.

UTIME

Input parameters: **char *file_name, struct utimbuf *utimes**

UTIME, used by **utime()**, sets file access and modification times.

UTIME must locate the directory entry and set the access and modification times of the file. If *utimes* is NULL, the times are set to the current time. Otherwise, the times are set to the values contained in the **utimbuf** structure, which is defined in the file **<utime.h>**.

If unsuccessful, **UTIME** should return **-1** and set *errno* to one of the following values:

[EACCES]	Search permission denied for a component of the path.
[ENAMETOOLONG]	A path name component length exceeds {NAME_MAX} or the length of <i>file_name</i> exceeds {PATH_MAX} .
[ENOENT]	<i>file_name</i> names a file that does not exist or points to an empty string.
[ENOTDIR]	A component of the path prefix is not a directory.
[EPERM]	The file permissions do not allow the times to be altered.
[EROFS]	<i>file_name</i> resides on a read-only file system.

PATHCONF

Input parameters: **char *file_name, int sysvar**

PATHCONF, used by **pathconf()**, retrieves POSIX path configuration limits by name.

Normally, the POSIX path configuration limits are the same for an entire file system, and a simple extended check for a file is all that needs to be done. Not all POSIX path configuration limits are appropriate for OS Open, but all are listed for completeness:

_PC_LINK_MAX	Number of links per directory entry.
_PC_MAX_CANON	Number of bytes in a terminal canonical input line. (terminal special files only).
_PC_MAX_INPUT	Minimum number of bytes available in a terminal input queue (terminal special files only).
_PC_NAME_MAX	Maximum number of bytes in a file name.
_PC_PATH_MAX	Maximum number of bytes in a path name.

`_PC_CHOWN_RESTRICTED`

chown() is restricted to threads having appropriate privileges.

`_PC_NO_TRUNC`

Path name components longer than **{NAME_MAX}** generate an error.

`_PC_VDISABLE`

Terminal special characters can be disabled using this character.

If unsuccessful, **PATHCONF** should return `-1` and set *errno* to one of the following values:

[EACCES]

Search permission denied for a component of the path.

[EINVAL]

The value of *sysvar* is invalid

[ENAMETOOLONG]

A path name component length exceeds **{NAME_MAX}** or the length of *file_name* exceeds **{PATH_MAX}**.

[ENOENT]

file_name names a file that does not exist or points to an empty string.

[ENOTDIR]

A component of the path prefix is not a directory.

FPATHCONF

Input parameters: **int file_descriptor**, **int sysvar**

FPATHCONF, used by **fpathconf()**, retrieves POSIX path configuration limits by descriptor.

Normally, these limits are the same for an entire file system, and the appropriate values are simply returned. Valid configurable limits are the same as those specified by the **PATHCONF** I/O control command.

If unsuccessful, **FPATHCONF** should return `-1` and set *errno* to the following value:

[EINVAL]

The value of *sysvar* is invalid

SEEK

Input parameters: **off_t seek_val**, **int whence**

SEEK, used by **lseek()**, repositions the file offset according to the value of *whence*.

whence is one of the following constants defined in the file **<unistd.h>**:

SEEK_SET

File offset is set to *seek_val*

SEEK_CUR

File offset is set to current value + *seek_val*

SEEK_END

File offset is set to the end of file (EOF) + *seek_val*

The file offset can be set past EOF. If data is later written at that point, reads from the gap return bytes of 0 until data is actually written.

If successful, **SEEK** should return the new offset. Otherwise, **SEEK** should return (**off_t**) -1 and set *errno* to the following value:

[EINVAL] *whence* is invalid or the resulting offset would be invalid.

CHMOD

Input parameters: **char *file_name, mode_t mode**

CHMOD, used by **chmod()**, changes the file permission mask.

CHMOD must locate the directory entry and change the file modes to the values specified in *mode*. The mode defines the access permission for the file owner, group members, and others.

CHMOD should return -1 and set *errno* to one of the following values:

[EACCES] Search permission denied for a component of the path.
[ENAMETOOLONG] A path name component length exceeds {**NAME_MAX**}
or the length of *file_name* exceeds {**PATH_MAX**}.
[ENOENT] *file_name* names a file that does not exist or points to an
empty string.
[ENOTDIR] A component of the path prefix is not a directory.
[EPERM] The file permissions do not allow the mode to be altered.
[EROFS] *file_name* resides on a read-only file system.

FSYNC

Input parameters: **int file_descriptor**

FSYNC, used by **fsync()**, causes all modified data in the open file specified by the *file_descriptor* parameter to be saved to permanent storage.

If successful, **fsync()** returns 0. Otherwise, **fsync()** return -1.

Driver-Specific Select

A call to the OS Open function **select()** results in a call to a driver-specific function having the following prototype:

```
int drvr_select(file_block_t *fb, int *flags);
```

fb points to the initialized file block.

flags are set by OS Open device support to any combination of the following flag constants: **SEL_ASYNC**, **SEL_READ**, **SEL_WRITE**, and **SEL_EXCP**. The flag constants are defined in the file **<sys/select.h>**.

The device-specific function returns 0 for success or -1 to indicate an error.

drv_select() must examine the flags and return the conditions that are true at the time of the call. For example:

- If **SEL_READ** is set, and characters are available to be read, **SEL_READ** remains set. Otherwise, the flag is cleared.
- If **SEL_WRITE** is set and the device can accept characters, **SEL_WRITE** remains set. Otherwise, the flag is cleared.
- If the **SEL_EXCP** flag is set and there are outstanding exceptions, the flag remains set.
- If no specified conditions are true at the time of the call, and **SEL_ASYNC** is set, the device driver must note that an asynchronous select request was made. If the specified condition becomes true, the device driver calls **select_notify()**, supplying the file block pointer of the initial **select()** and the updated flag value.

Very few logical file systems implement a driver-specific **select()**. Instead, they treat regular file access as a synchronous operation that is not subject to asynchronous completion or the exceptions provided for in **select()**. To redirect **select()** to an underlying device driver, **select_redrive()** can be used.

Chapter 10. Using the DOS Logical File System

The OS Open DOS logical file system (DOS file system) implements a DOS file allocation table (FAT) file system. The DOS file system provides an interface between high-level POSIX 1003.1b file servers and block device drivers. The block device drivers manage the block devices on which DOS file system data are stored.

The DOS file system manages data on disk and diskette media formatted for DOS 5.0 as a set of POSIX 1003.1b files. The underlying POSIX file system supports the DOS file system directory and file structure, and DOS file system operations.

The DOS file system supports multiple DOS formatted units, and uses a sector buffer cache to reduce media access.

For more information about logical file systems, see Chapter 9, "Writing Logical File Systems."

Installing the DOS File System Driver

A call to **driver_install()** installs the DOS file system driver. The following code fragment provides an example:

```

        :
int devhandle;
rc=driver_install(&devhandle, fatfs_init,
    num_units;          /* integer specifying the number of supported
                        DOS formatted units */
    num_buffs);         /* integer specifying the number of buffers
                        in the sector cache */
        :
```

num_units controls the internal allocation of control blocks and sets the upper limit of supported control blocks.

num_buffs specifies the number of 512-byte sector buffers in the sector cache. All formatted units share the sector cache. Sector buffering enhances the performance of slower devices.

If the DOS system driver installs successfully, **driver_install()** returns 0. Otherwise, **driver_install()** returns -1.

Installing Devices for DOS File Systems

After the DOS file system driver is installed, **device_install()** is used to install the underlying block device or devices.

The following code fragment provides an example:

```
•
•
rc=device_install("/fat", LFSTYPE, devhandle,
    unit_number, /* integer specifying unit number that names the device */
    block_fd,    /* open file descriptor of underlying block device */
    offset);     /* offset of start of DOS file system on block device */
•
•
```

unit_number ranges from 0 to *num_units* - 1. (*num_units* was specified when the driver was installed.)

block_fd provides an open file descriptor for a block device.

offset specifies the block offset of the start of the DOS file system. This parameter enables multiple DOS file systems to be on one physical block device.

If the devices install successfully, **device_install()** returns 0. Otherwise, **device_install()** returns -1.

After device installation, DOS files can be accessed using **/fat** as the file system name.

Uninstalling Devices for DOS File Systems

The **device_uninstall()** call can be used to uninstall a particular device that does not have any outstanding opens against it. Following is an example of the **device_uninstall** call.

```
/* "/fat" is whatever the system name the DOS File System */
/* used when it installed */
rc = device_uninstall("/fat");
```

If successful, **device_uninstall()** return 0; otherwise -1 is return.

Opening DOS Files

The name given to **device_install** names the DOS file system and serves as the “root” of the file system. The name determines which file system receives the request. The rest of the path name, passed to the logical file system, locates the file in the DOS directory tree.

Note: The POSIX file delimiter / (slash) is used to delimit DOS file system path name components.

For example, suppose that a DOS file system is installed using the name **/diskette** and its underlying device is a diskette drive. If a diskette containing the file “rootdir\myfile.bat” was inserted in the diskette drive, the following command would open the file:

```
fd = open("/diskette/rootdir/myfile.bat", O_RDWR);
```

Note: OS Open device I/O library supports the notion of a working directory and provides **chdir()** and **getcwd()** to manage the working directory. Its path is prepended to any file name not beginning with the file delimiter. The working directory applies to all file systems.

Reading and Writing DOS Files

An open DOS file can be read from or written to using **read()** and **write()** commands.

DOS File System I/O Control Commands

Logical file systems support several I/O control (ioctl) commands. These commands are rarely issued directly by application programs. One exception is the **FREE_SPACE** ioctl command which an application can issue to determine the number of bytes represented by the unallocated clusters at the time of the call.

More commonly, the commands are used by POSIX 1003.1b file functions. The ioctl commands are described in detail in Chapter 9, "Writing Logical File Systems." Table 10-1 lists the commands and the higher-level OS Open functions that use them, along with a summary description.

Table 10-1. I/O Control Commands for DOS File Systems

Command	OS Open Function	Description
OPENDIR	opendir()	Opens a DOS directory
READDIR	readdir()	Reads a DOS directory (implemented using READDIR_R)
READDIR_R	readdir_r()	Reformats a DOS directory entry into the POSIX directory structure
REWINDDIR	rewinddir()	Resets the DOS directory offset to 0
LINK	link()	Not supported in the DOS file system
UNLINK	unlink()	Deletes a file when no open descriptors are on the file.
MKDIR	mkdir()	Creates a file having special format in the parent DOS directory.
RMDIR	rmdir()	Deletes a DOS directory
RENAME	rename()	Changes the directory entry of a file
RF_STAT	stat()	Retrieves file statistics by name
RF_FSTAT	fstat()	Retrieves file statistics by descriptor
RF_ACCESS	access()	Queries file permissions or existence

Table 10-1. I/O Control Commands for DOS File Systems

Command	OS Open Function	Description
UTIME	utime()	Sets file access and modification times
PATHCONF	pathconf()	Retrieves POSIX path configuration limits by name
FPATHCONF	fpathconf()	Retrieves POSIX path configuration limits by descriptor
SEEK	lseek()	Repositions the file offset
CHMOD	chmod()	Changes the file permission mask Valid modes for the DOS file system are: S_IRUSR Read permission S_IWUSR Write permission S_IRUSR S_IWUSR Read and write permission Write-only permission is not supported for DOS files. S_IRUSR S_IWUSR and S_IWUSR give the same permission
FSYNC	fsync()	Assures all data associated with the file will be saved on the permanent storage.specified

Interfacing a DOS File System to a Block Device Driver

After each block device driver is installed successfully, the attributes of each block device driver are set and saved.

The DOS file system issues an **ioctl()** call with QDEVATTR command to query the device attributes at the initialization stage. The return of this query is REMOVEABLE_MEDIA, ERASE_ON_DELETE, or zero.

When the DOS file system needs to update the file system, it issues a **strategy()** call with BLKMEDIA_CHANGE request type, if the block device supports the REMOVABLE_MEDIA attribute. If the device has been removed, the DOS file system will invalidate all the data related to the block device.

Whenever a DOS file system releases data sectors and deletes a file, it issues a **strategy()** call with BLOCK_ERASE request type, if the block device supports the ERASE_ON_DELETE attribute. The DOS file system minimizes the number of **strategy()** calls by consolidating reads or writes to contiguous sectors where possible.

Using the OS Open RAM Disk

The OS Open RAM disk can be used with the DOS Logical File system library to implement a DOS file system at the specified or system-provided location in system RAM. The RAM disk provides either an initialized (containing files) or uninitialized (empty) file system in memory.

After the RAM disk is installed, the DOS (FAT) Logical File System is installed to create a DOS directory (eg. "/ram"). Users can then read/write or ftp (file transfer protocol) files under this new directory.

To initialize files in the RAM disk, a host utility, **rambuild**, is provided. **rambuild** copies files from a designated directory to produce an assembler source file which can be assembled and linked with the RAM disk to pre-loaded files in RAM.

Installing the RAM Disk Device Driver

The RAM Disk device driver can be initialized by two different ways, depending on usage. If the RAM disk device driver needs to be located in a predefined location, then it should be initialized using *ramdisk_init_ext*; otherwise *ramdisk_init* is used. A call to **driver_install()** installs the RAM disk driver. The following code fragment provides an example:

```
.
.
.
#include <ramdiskLib.h>
int devhandle;
#ifdef SPECIFIED_ADDR /* build disk image at the specified location */
    rc = driver_install(&devhandle, ramdisk_init_ext);
#else /* build disk image at system provided location */
    rc = driver_install(&devhandle, ramdisk_init);
#endif
.
.
.
```

If the RAM disk driver installs successfully, **driver_install()** returns 0. Otherwise, **driver_install()** returns -1.

Installing Devices for RAM Disk File Systems

After the RAM disk file system driver is installed, **device_install()** is used to install the underlying block devices. The different initialization functions will pass different parameters to **device_install()**.

When the *ramdisk_init_ext* parameter is used to initialize **device_install**, the call should look like the following fragment:

```
extern char _ram_image_data[];
rc = device_install("/ram", BLKTYPE, devhandle, ptr, size, loaded);
```

Named devices, such as "/ram", can be created using **device_install()**. Device type BLKTYPE is defined in <**sys/devDriver.h**>. *devhandle* is the value obtained from **driver_install()**.

The settings of the last three parameters in the call will work as described below:

ptr	size	loaded	explanation
NULL	y_size	NULL _ram_image_data	RAM Disk is built from heap memory for a specified size, with optional preload.
x_ptr	y_size	NULL _ram_image_data	A new disk image is built at the specified address for the specified size, with optional preload.
x_ptr	0	NULL _ram_image_data	A disk image is assumed to exist at the specified address, with optional preload.

When the *ramdisk_init* parameter is used to initialize **device_install**, the call should look like the following fragment:

```
extern char _ram_image_data[];
rc = device_install("/ram", BLKTYPE, devhandle, size, loaded);
```

The difference in the above calls is one fewer parameter: *ptr*. For this call the RAM Disk is built from heap memory of a specified size, and the value of the *size* parameter cannot be zero.

The following code fragment provides an example:

```
.
.
.
#ifdef INIT_RAM /* define INIT_RAM if pre-loading RAM disk */
extern char _ram_image_data[];
#endif
unsigned long ramsize;
ramsize = 0x2048; /* RAM Disk Data size = 2k bytes */
rc = device_install("/ram", BLKTYPE, devhandle, ramsize)
#ifdef INIT_RAM
    _ram_image_data); /* Data to pre-load from ramdisk.s */
#else
```

```

    NULL); /* Empty RAM Disk */
#endif

```

```

.
.
.

```

If the device installs successfully, **device_install()** returns 0. Otherwise, **device_install()** returns -1.

Opening RAM Disk Files

The DOS (FAT) file system uses the RAM disk file system as a block device driver. After RAM disk installation is complete, the installation procedure passes the RAM file descriptor to the DOS (FAT) file system. The following code fragment provides an example:

```

.
.
.
int fd_ram;
fd_ram = open("/ram", O_RDWR); /* Get a file descriptor to RAM Disk */
/* Install DOS File System */
rc=driver_install(&fsdevh, fatfs_init, 4, 16);
/* Pass RAM file descriptor to DOS File System */
rc=device_install("/fat", LFSTYPE, fsdevh, 0, fd_ram, 0)

```

Reading and Writing RAM Disk Files

An open RAM disk file can be read from or written to using OS Open **read()** and **write()** commands or "C" commands such as **fread()** and **fwrite()**.

Using rambuild to Preload the RAM Disk

On the host, type "rambuild" or "rambuild <directory name>" on the command line. For the ELF version of **rambuild**, type "rambuild -e <directory name>". If no directory name is entered, **rambuild** will prompt for the name of the directory that contains the files to preload. **rambuild** will then go to the directory and its subdirectories, reading each file and putting all of the information into the output file **ramdata.s** as a single directory. This output file must then be assembled and linked with OS Open to create an initialized RAM disk.

During the processing of **device_install()** for a RAM disk, the files that were put in **ramdata.s** by **rambuild** will be copied into the newly created RAM disk. The size of the created RAM disk (specified in the fourth parameter of **device_install()**)

must be large enough to hold all of these files. Otherwise, **device_install()** for the RAM disk will fail.

rambuild will not copy file names starting with a dot (‘.’) or containing more than two dots. File names which exceed eight characters are truncated to eight characters. File name extensions exceeding three characters are truncated to three characters. All alpha characters in the file names are set to uppercase, regardless of how they were entered.

Using OS Open SCSI Support

OS Open provides SCSI initiator support through the use of two stackable block device drivers. The highest level driver, called the “SCSI device driver”, accepts standard block device driver requests (**ioctl()** and **strategy()**) from an application or file system and converts them into the SCSI commands that are needed to satisfy the request. These SCSI commands are passed to the lower level “SCSI adapter device driver” via **ioctl()** calls. The SCSI adapter device driver will manage the SCSI adapter hardware in order to issue the SCSI commands to a SCSI device and manage the commands’ execution.

OS Open supplies the SCSI device driver and example SCSI adapter device driver skeleton source code. The skeleton source code is provided in the examples subdirectory and can be used as a starting point for creating an OS Open SCSI adapter device driver for any SCSI chip. The SCSI device driver currently supports only non-removable hard disks.

SCSI Device Driver

The OS Open SCSI device driver is a block device driver which accepts standard block device driver requests (**ioctl()** and **strategy()**) from an application or file system and converts them into the SCSI commands that are needed to satisfy the request. The SCSI commands are passed via **ioctl()** calls to an open SCSI adapter device driver for execution.

Device Driver Installation

The SCSI device driver is installed by calling **driver_install()**. Following is an example of the SCSI device driver installation.

```
#include <scsiLib.h>
int devhandle,rc;
rc=driver_install(&devhandle, SCSI_init);
```

SCSI_init() is declared in the file **<scsiLib.h>** as follows:

```
int SCSI_init(driver_t *dsw, va_list vargs)
```


Upon successful installation, **driver_install()** returns 0; otherwise -1 is returned. For more information on **driver_install()**, refer to the *OS Open Programmer's Reference*.

Device Installation

After the SCSI device driver is installed, named devices can be created using **device_install()**. Following is an example of device installation.

```
rc = device_install("/SCSI6", BLKTYPE, devhandle, fd_a, 6, 0, DD_SCDISK,  
DS_PV, 512);
```

For device installation, *devhandle* is the value obtained from the **driver_install()**. Device type BLKTYPE is defined in **<sys/devDriver.h>**.

Additional parameters passed in the **device_install()** call are as follows:

Parameter	Meaning
First Parameter	File descriptor from open() on a SCSI adapter device.
Second Parameter	SCSI ID of target device.
Third Parameter	LUN (logical unit number) used on target device (usually 0).
Fourth Parameter	Device type.
Fifth Parameter	Device sub-type.
Sixth Parameter	Block size (number of bytes per block) of target device.

These are positional parameters. The device type and sub-type constants can be found in **<sys/devinfo.h>**.

Upon successful installation, **device_install()** returns 0; otherwise -1 is returned. For more information on **device_install()**, refer to the *OS Open Programmer's Reference*.

Uninstalling a SCSI Device

The **device_uninstall()** call can be used to uninstall a particular device that does not have any outstanding opens against it. Following is an example of the **device_uninstall()** call.

```
rc=device_uninstall("/SCSI6");
```

If successful, **device_uninstall()** returns 0; otherwise -1 is returned. For more information on **device_uninstall()**, refer to the *OS Open Programmer's Reference*.

Opening a SCSI Device

After the device is installed, the **open()** system call can be used to open a particular device. Following is an example of the **open()** system call used against a SCSI device:

```
fd_s = open("/SCSI6", O_RDWR);
```

Note: The oflag parameter, `O_RDWR` in this example, which is passed in the `open` call, is ignored by the device driver.

Note: All opened devices must be closed before the device can be uninstalled.

When successful, **`open()`** returns a nonnegative integer file descriptor. Otherwise **`open()`** returns `-1`. For more information on **`open()`**, refer to the *OS Open Programmer's Reference*.

Closing a SCSI Device

After a particular device has been opened, the **`close()`** system call can be used to close the device. Following is an example of the **`close()`** system call used against a SCSI device:

```
rc = close(fd_s);
```

When successful, **`close()`** returns 0. Otherwise **`close()`** returns `-1`. For more information on **`close()`**, refer to the *OS Open Programmer's Reference*.

Reading and Writing

After successfully installing and opening a SCSI device, the **`strategy()`** function is used to read and write blocks. The SCSI device driver translates the **`strategy()`** function call into the SCSI commands that are needed to satisfy the request. These SCSI commands are passed via **`ioctl()`** calls to an open SCSI adapter device driver for execution.

A SCSI device, like other block devices, is treated as a linear collection of fixed size blocks (e.g. 512 bytes). Each block is assigned a number called its block ID with 0 being assigned to the first accessible block. The **`strategy()`** call to a SCSI device passes a pointer to an initialized structure that provides the type of the request (read or write), the memory address of the request, the first block ID of the request, and the number of sequential blocks requested.

If the **`strategy()`** call returns 0 the access was successful, otherwise it was not and *errno* will be set. An example of reading, clearing, and writing blocks is shown below. *fd_s* is the value obtained from the **`open()`** call.

```
unsigned char myblock[512 * 3]; /* buffer space for 3 blocks */
block_req_t bk_request;

bk_request.request_type = BLOCK_READ;
bk_request.block_buffer = myblock; /* destination of read data */
bk_request.block_id = 9; /* read starts with block 9 (the 10th block) */
bk_request.block_count = 3; /* reads blocks 9, 10, and 11 */
if (strategy(fd_s, bk_request) != 0) trouble();
memset(&myblock[512], 0x00, 512); /* clear data in middle block */
bk_request.request_type = BLOCK_WRITE;
if (strategy(fd_s, bk_request) != 0) trouble();
```

For more information on **strategy()**, refer to the *OS Open Programmer's Reference*.

I/O Control

An **ioctl()** call issued against the SCSI device driver accepts the following commands. If the **ioctl()** call returns 0 the command was successful, otherwise it was not and *errno* will be set. In each of these commands, *fd_s* is the value obtained from the **open()** call.

QDEVATTR

This command causes the SCSI device driver to return the current SCSI device's attributes in the structure pointed to by a passed parameter.

```
block_dev_attr_t blkattr;  
rc = ioctl(fd_s, QDEVATTR, &blkattr);
```

block_dev_attr_t is defined in **<sys/blockdev.h>**. The *block_dev_attr_t* variable pointed to by *blkattr* will be set to either *REMOVABLE_MEDIA* or 0 to indicate whether the block device supports removable media or not.

IOCINFO

This command causes the SCSI device driver to return information about the current SCSI device in the structure pointed to by a passed parameter.

```
devinfo_t devinfo;  
rc = ioctl(fd_s, IOCINFO, &devinfo);
```

devinfo_t is defined in **<sys/devinfo.h>**. The members of the *devinfo_t* structure have the following meaning:

<i>devtype</i>	Type of device (e.g. SCSI disk).
<i>devsubtype</i>	Sub-type of device (e.g. Physical Volume).
<i>un.scdk.blksize</i>	Number of bytes per block.
<i>un.scdk.numblks</i>	Total number of blocks.
<i>un.scdk.max_request</i>	Maximum number of blocks per request.

SCSI Adapter Device Drivers

In order to work with the OS Open SCSI device driver, a SCSI adapter device driver must be written to accept and execute the following IOCTL commands:

- RESET_ADAPTER
- RESET_SCSI_BUS
- EXECUTE_SCSI_COMMAND
- READ_ADAPTER_PARAMETERS

- WRITE_ADAPTER_PARAMETERS

These commands and the data structures that they use are declared in **<scsiLib.h>**. OS Open supplies example SCSI adapter device driver skeleton source code in the examples subdirectory and in the Sample Device Drivers appendix. The skeleton code can be used as a starting point for creating an OS Open SCSI adapter device driver that will work with the OS Open SCSI device driver. The example code consists of the two files **ex_sadpt.h** and **ex_sadpt.c**. A functional description of the scsiadpt SCSI adapter device driver is provided below that documents how the skeleton example should behave once completed. Implementation notes are provided in some sections to assist in the completion of the device driver.

The scsiadpt SCSI Adapter Device Driver

The scsiadpt SCSI adapter device driver is a SCSI-2 initiator implementation designed to work with the OS Open SCSI device driver. The scsiadpt device driver issues and oversees the execution of SCSI commands to SCSI targets in response to **ioctl()** calls.

Device Driver Installation

The scsiadpt device driver is installed by calling **driver_install()**. Following is an example of the scsiadpt device driver installation.

```
#include <ex_sadpt.h>
int devhandle,rc;
rc=driver_install(&devhandle, scsiadpt_init);
```

scsiadpt_init() is declared in the file **<ex_sadpt.h>** as follows:

```
int scsiadpt_init(driver_t *dsw, va_list vargs);
```

Upon successful installation, **driver_install()** returns 0; otherwise -1 is returned.

Implementation Notes

driver_install() calls a device driver initialization function (**scsiadpt_init()** in this case) which must provide the address of the driver's device switch table. The device switch table contains the function pointers for all of the device specific functions. For more information on **driver_install()**, refer to the *OS Open Programmer's Reference*.

Device Installation

After the scsiadpt device driver is installed, named devices can be created using **device_install()**. Following is an example of device installation.

```
rc=device_install("/scsiadpt", BLKTYPE, devhandle, 7, 0xf0000000,
EVENT_EXT1_NP, DMA_CHANNEL2, 1);
```

For device installation, *devhandle* is the value obtained from the **driver_install()**. Device type BLKTYPE is defined in **<sys/devDriver.h>**.

Additional parameters passed in the **device_install()** call are as follows:

Parameter	Meaning
First Parameter	SCSI Initiator ID to use.
Second Parameter	SCSI chip address.
Third Parameter	External interrupt used by SCSI chip.
Fourth Parameter	DMA channel used by SCSI chip.
Fifth Parameter	Queue depth (must be 1). This is the maximum number of ioctl commands that will be allowed to execute at one time. This also limits the number of SCSI commands that can be outstanding at any time.

These are positional parameters. The external interrupt parameter constants can be found in **<flih.h>**. The DMA channel parameter constants can be found in **<ioLib.h>**.

Upon successful installation, **device_install()** returns 0; otherwise -1 is returned.

Implementation Notes

device_install() calls a device driver configuration function (**scsiadpt_configure()** in this case) which allocates a device specific control structure (dds) that is filled in with information about the device that is being installed. The address of this dds must be stored in the location pointed to by the passed dds argument. The dds address will be provided to other device specific functions in the *driver_specific* member of a file block structure for the installed device. The address of this file block structure is passed as the first argument to the other device specific functions.

Note: **device_install()** may be called multiple times against an installed device driver to configure new devices.

Note: The device driver configuration function is also called by **device_uninstall()**. Prior to calling the configuration function, **device_uninstall()** sets the first variable argument to -1. Thus, -1 must not be a valid value for the first variable argument passed to the configuration function.

For more information on **device_install()**, refer to the *OS Open Programmer's Reference*.

Uninstalling a scsiadpt Device

The **device_uninstall()** call can be used to uninstall a particular device that does not have any outstanding opens against it. Following is an example of the **device_uninstall()** call.

```
rc=device_uninstall("/scsiadpt");
```

If successful, **device_uninstall()** returns 0; otherwise -1 is returned.

Implementation Notes

device_uninstall() calls the device driver configuration function (**scsiadpt_configure()** in this case) with the first variable argument set to -1 to distinguish it from the call due to a device install. The configuration function should destroy all objects associated with the device such as semaphores and mutexes, uninstall any first level interrupt handlers used by the device, free any allocated memory associated with the installed device including the dds, and then return. For more information on **device_uninstall()**, refer to the *OS Open Programmer's Reference*.

Opening a scsiadpt Device

After the device is installed, the **open()** system call can be used to open a particular device. Following is an example of the **open()** system call used against a scsiadpt device:

```
fd_a = open("/scsiadpt", O_RDWR);
```

Note: The oflag parameter, O_RDWR in this example, which is passed in the open call, is ignored by the device driver.

When successful, **open()** returns 0, otherwise -1 is returned.

Implementation Notes

open() calls the device driver open function (**scsiadpt_open()** in this case). This function counts the number of outstanding opens against the device in the dds. This count must be zero to uninstall the device.

For more information on **open()**, refer to the *OS Open Programmer's Reference*.

Closing a scsiadpt Device

After a particular device has been opened, the **close()** system call can be used to close the device. Following is an example of the **close()** system call used against a scsiadpt device:

```
rc = close(fd_a);
```

When successful, **close()** returns 0. Otherwise **close()** returns -1. For more information on **close()**, refer to the *OS Open Programmer's Reference*.

Implementation Notes

close() calls the device driver close function (**scsiadpt_close()** in this case). This function decrements the count of outstanding opens against the device in the dds. This count must be zero to uninstall the device.

For more information on **open()**, refer to the *OS Open Programmer's Reference*.

Required I/O Control Commands

An **ioctl()** call issued against the `scsiadpt` device driver accepts the following required commands in order to work with the OS Open SCSI device driver. If the **ioctl()** call returns 0 the command was successful, otherwise it was not and *errno* will be set. In each of these commands, *fd_a* is the value obtained from the **open()** call.

RESET_ADAPTER

This command causes the `scsiadpt` device driver to reset and initialize the SCSI adapter and to reset the SCSI bus. It also resets the current adapter transfer parameters for all targets.

```
rc = ioctl(fd_a, RESET_ADAPTER);
```

RESET_SCSI_BUS

This command causes the `scsiadpt` device driver to reset the SCSI bus and to reset the current adapter transfer parameters for all targets.

```
rc = ioctl(fd_a, RESET_SCSI_BUS);
```

EXECUTE_SCSI_COMMAND

This command causes the `scsiadpt` device driver to issue and oversee the execution of a SCSI command to a specified target.

```
unsigned char data_buffer[255];
scsi_command_t command;
command.linked_command = NULL; /* Must be NULL */
command.target_id = 0;
command.identify_message = SCSI_IDENTIFY;
command.queue_tag_message = 0;
command.queue_tag = 0;
command.message_bytes_to_send = 0;
for (i = 0; i < 5; command.message_out_byte[i++] = 0);
for (i = 0; i < 12; command.cdb[i++] = 0);
command.cdb_length = 6;
command.cdb[0] = SCSI_REQUEST_SENSE;
command.cdb[4] = 255; /* allocation length (largest transfer) */
command.data_length = 255;
command.data_address = (void *) data_buffer;
command.data_direction = SCSI_IN;
command.trunc_transfer_ok = true;
command.bus_free_expected = false;
command.check_expected = false;
command.command_timeout = 0;
rc = ioctl(fd_a, EXECUTE_SCSI_COMMAND, &command);
```

scsi_command_t is defined in **<scsiLib.h>**. The following members of the *scsi_command_t* structure are inputs to the EXECUTE_SCSI_COMMAND command. Comments in braces ({}) are used to describe the OS Open SCSI device driver's use of a member.

Member	Use
<i>linked_command</i>	Points to the next <i>scsi_command_t</i> in a series of linked commands {always NULL}.
<i>target_id</i>	SCSI id of target to send command to.
<i>identify_message</i>	Identify message to send to target. 0 means none.
<i>queue_tag_message</i>	Queue Tag message to send to target. 0 means none {always 0}.
<i>queue_tag</i>	Queue Tag to send to target. Only used if <i>queue_tag_message</i> is not 0 {unused}.
<i>message_bytes_to_send</i>	Number of additional message bytes to send to the target {always 0}.
<i>message_out_byte[n]</i>	Additional message bytes to send to target {unused}.
<i>cdb_length</i>	Length of SCSI CDB to send to target.
<i>cdb[n]</i>	CDB bytes to send to target.
<i>data_length</i>	Number of data bytes expected for this command.
<i>data_address</i>	Pointer to first data byte for this command. Only used if <i>data_length</i> is not 0.
<i>data_direction</i>	Direction of data transfer. Only used if <i>data_length</i> is not 0.
<i>trunc_transfer_ok</i>	<i>true</i> : Allows the command to be considered successful even if all of the data bytes expected for this command were not transferred. <i>false</i> : The command will only be considered successful if all of the data bytes expected for this command are transferred. This member is only used if <i>data_length</i> is not 0.
<i>bus_free_expected</i>	<i>true</i> : Allows the command to be considered successful even if a bus free condition is detected. <i>false</i> : The command will only be considered successful if status is received. {always <i>false</i> }
<i>check_expected</i>	<i>true</i> : Allows the command to be considered successful even if a check condition is received. <i>false</i> : The command will only be considered successful if good status is received.
<i>command_timeout</i>	Command timeout in seconds. 0 means none.

Note: These input members are not modified by the SCSI adapter device driver.

The following members of the *scsi_command_t* are working variables of the EXECUTE SCSI_COMMAND command and are used as follows:

Member	Use
<i>additional_error</i>	Error information in addition to the set <i>errno</i> value. Valid only if <i>rc</i> != 0; Possible values are described below.
<i>state</i>	Working variable for device driver. It should be ignored.
<i>message_bytes_received</i>	Count of additional message bytes received from the target.
<i>message_in_byte[n]</i>	Additional message bytes received from the target.
<i>residual_data_length</i>	Number of expected bytes not transferred for this command.
<i>residual_data_address</i>	Pointer to byte beyond the last byte transferred for this command.
<i>scsi_status</i>	SCSI status byte received for this command.
<i>scsi_command_complete_message</i>	SCSI message received after status byte.
<i>misc</i>	Pointer to additional working variables.

Additional Errors

The following additional errors are defined in **<scsiLib.h>**.

- [SCSI_SELECTION_TIMEOUT] Selection timeout.
- [SCSI_BAD_INTERRUPT] Unexpected interrupt/error from SCSI chip.
- [SCSI_DATA_SHORT] Data transfer did not complete.
- [SCSI_BAD_STATUS] SCSI Status byte not good.
- [SCSI_BAD_SEQUENCE] Invalid SCSI phase sequence.
- [SCSI_COMMAND_TIMEOUT] Command timeout.

READ_ADAPTER_PARAMETERS

This command causes the scsiadpt device driver to return the current adapter parameters in the structure pointed to by a passed parameter.

```
scsi_adapter_parm_t adapter_parm;  
rc = ioctl(fd_a, READ_ADAPTER_PARAMETERS, &adapter_parm);
```

scsi_adapter_parm_t is defined in **<scsiLib.h>**. The members of the *scsi_adapter_parm_t* are used as follows:

Member	Use
<i>scsi_initiator_id</i>	SCSI ID of initiator.
<i>adapter_base_address</i>	SCSI chip address.

<i>interrupt_level</i>	External interrupt used by SCSI chip.
<i>dma_channel</i>	DMA channel used by SCSI chip.
<i>queue_depth</i>	Queue depth.
<i>max_scsi_devices</i>	Number of SCSI IDs supported.
<i>best_xfer_parm</i>	Structure containing the best data transfer period, offset, and width supported by the device driver. These values are bytes in the format of the SCSI synchronous data transfer request message period and offset bytes and the SCSI wide data transfer request exponent byte.
<i>current_xfer_parm[n]</i>	Structure containing the current data transfer period, offset, and width for target ID n.
<i>requested_xfer_parm[n]</i>	Structure containing the requested data transfer period, offset, and width for target ID n.

A SCSI initiator and target must agree on the transfer parameters with which they will exchange data. The *current_xfer_parm[n]* structure contains the transfer parameters that the SCSI adapter device driver is currently using when it exchanges data with target n. The *best_xfer_parm* structure contains the best transfer parameters that the device driver is capable of using. By using the `WRITE_ADAPTER_PARAMETERS` ioctl command to update the *requested_xfer_parm[n]* structure, the device driver can be instructed to negotiate a new agreement with target n using the requested parameters. After the negotiation is complete, the *current_xfer_parm[n]* structure will contain the newly agreed upon transfer parameters.

WRITE_ADAPTER_PARAMETERS

This command causes the `scsiadpt` device driver to update the current adapter parameters with the values in the structure pointed to by a passed parameter. Only the requested data transfer parameters can be updated by this command. The other members in the adapter parameter structure are ignored.

```
scsi_adapter_parm_t adapter_parm;
rc = ioctl(fd_a, READ_ADAPTER_PARAMETERS, &adapter_parm);
adapter_parm.requested_xfer_parm[0] = adapter_parm.best_xfer_parm;
rc = ioctl(fd_a, WRITE_ADAPTER_PARAMETERS, &adapter_parm);
```

Using SCSI Devices with the DOS File System

The SCSI device driver is designed to work well with the DOS logical file system. This is accomplished by passing the file descriptor returned from a successful **open()** call against the SCSI device driver to the **device_install()** call against the DOS logical file system. The following code fragments provide an example of this using the `scsiadpt` example SCSI adapter device driver.

```

int do_scsi() {
    int adev_handle, sdev_handle, fsdevh, fd_scsi, fd_adapter;

    if (driver_install(&adev_handle, scsiadpt_init) != 0) trouble();
    if (device_install("/scsiadpt", BLKTYPE, adev_handle,
        7, 0xf0000000, EVENT_EXT1_NP, DMA_CHANNEL2, 1) != 0) trouble();
    fd_adapter = open("/scsiadpt", O_RDWR);
    if (fd_adapter == -1) trouble();
    if (driver_install(&sdev_handle, SCSI_init) != 0) trouble();
    if (device_install("/scsiLib", BLKTYPE, sdev_handle,
        fd_adapter, 6, 0, DD_SCDISK, DS_PV, 512) != 0) trouble();
    fd_scsi = open("/scsiLib", O_RDWR);
    if (fd_scsi == -1) trouble();
    if (driver_install(&fsdevh, fatfs_init, 4, 16) != 0) trouble();
    if (device_install("/scsi", LFSTYPE, fsdevh, 0, fd_scsi, 0x20) != 0)
        trouble();
    chdir("/scsi");
    return 0;
}

```

In this example, a SCSI disk drive formatted for the FAT file system with a partition starting at block 0x20 can now be accessed using the DOS file system calls (e.g. **fopen()**, **fread()**, **fclose()**).

Note: In order to work with the DOS file system, the installed SCSI target device's block size must be 512 bytes.

Chapter 11. OS Open PCMCIA Support

OS Open provides PCMCIA support by supplying a Card Services/Enabler software layer, an Exchangeable Card Architecture (ExCA) compliant Socket Services implementation, and a device driver for a PCMCIA hard disk (pataLib).

The device drivers for PCMCIA devices are standard OS Open device drivers that must satisfy a few additional requirements in order to work correctly with the Card Services/Enabler layer. These requirements are described in the Card Services/Enabler software layer section.

The Socket Services implementation (ssLib) manages PCMCIA socket controller chips, providing an industry standard, hardware independent interface to functions that control the socket power, memory and I/O mapping, and interrupt routing.

The Card Services/Enabler software layer (csLib) handles the dynamic configuration of cards as they are inserted into and removed from PCMCIA sockets.

The following section introduces some basic PCMCIA terms and concepts. An understanding of these terms and concepts will allow OS Open users to more easily take advantage of OS Open's PCMCIA support.

Introduction to PCMCIA

The Personal Computer Memory Card International Association (PCMCIA) has produced a set of standards to promote the interchangeability of Integrated Circuit Cards (IC Cards) between a variety of computer and electronic products. The 68 pin cards that are compliant with these standards are called PC Cards. The standards also specify a multi-layered software architecture that provide hardware independent interfaces to access the PC Cards. These layers are called Socket Services and Card Services. In order to promote the goal of interoperability of PC Cards, the Intel corporation has produced the Exchangeable Card Architecture (ExCA) Specification. ExCA specifies a minimum set of PCMCIA standard interfaces that a PC Card, Socket Services, and Card Services implementation must support in order to be considered ExCA compliant. The Socket Services implementation provided by OS Open is compliant with release 1.50 of the ExCA Specification.

Note: The Intel corporation has turned over the ExCA Specification to PCMCIA so that it can be used as a compliance guide in future PCMCIA standards. It is anticipated that PCMCIA will adopt a different name for the ExCA Specification.

Figure 12-1 illustrates the relationships between the hardware and software components in a PCMCIA system and may be a useful reference in the next sections.

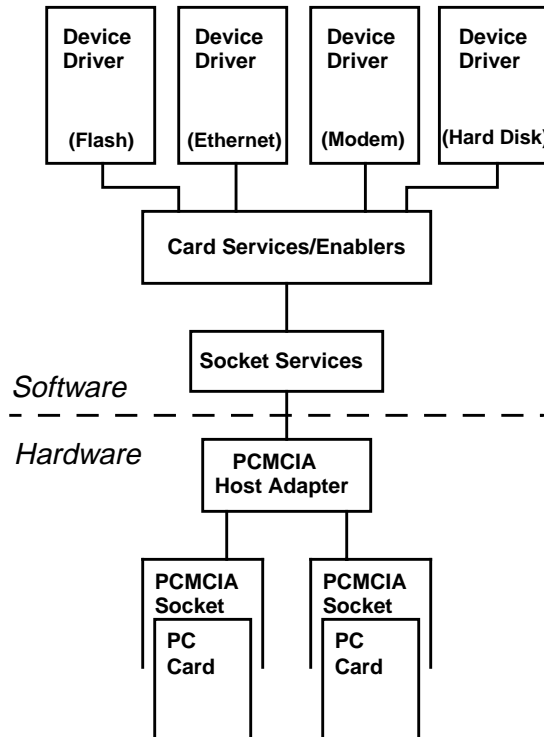


Figure 12-1. PCMCIA Hardware/Software Structure

PC Cards

A PC Card is a memory or I/O device with a 68 pin connector on one side and is roughly the size of a small stack of credit cards. The connector side of the PC Card can be inserted into and removed from a socket that has pins that mate with the PC Card's connector. The mechanical dimensions of both the PC Cards and the sockets are specified in the PCMCIA standards.

Each PC card may contain memory and/or registers that are accessed in its common memory, attribute memory, or I/O address space. Each of these address spaces are 64 MB in size. When a card is first powered on, only the common memory and attribute memory address space can be accessed. The common

memory space is most often used to address banks of memory on the PC Card (such as flash banks, SRAM memory, etc.).

The attribute memory space is used to access a memory on the PC Card that contains a data structure called the Metaformat or the Card Information Structure (CIS). The CIS begins at address 0 and contains information about the card, often including its identification, capabilities, access timings, and power requirements. The information in the CIS is organized in a series of multi-byte tuples. Each tuple begins with a tuple code that identifies the tuple and a link field that indicates the number of bytes following the link field in the tuple. The format of the tuple data is specified in the PC Card standard.

The attribute memory space is also used to address a PC Card's configuration registers. The configuration registers are used to set configuration options such as the address range that the PC card will respond to. A PC Card that has memory or registers that are accessed in its I/O address space must first be configured as an I/O device by writing the appropriate values to its configuration registers.

Once a PC Card is configured as an I/O device, the I/O address space may be used to access memory and/or registers that control the PC Card. These registers are typically accessed by device drivers in order to cause the PC Card to perform a desired function or to determine the status of an operation. In addition, a PC Card configured as an I/O device can generate an interrupt which is available at the PC card connector.

Socket Controllers

The pins on the PCMCIA socket are connected to a chip called a PCMCIA socket controller, socket adapter, or host adapter (these names are used interchangeably in this chapter). The socket controller is usually designed so that it can be easily attached to an ISA bus and one or more PCMCIA sockets with little or no glue logic. The socket controller is accessed by the system software at some well known ISA address. The socket controller can be set to interrupt the system when it detects a socket interface status change (such as a card insertion or removal). The socket controller also controls power to the sockets, provides a mapping between address spaces on the ISA bus and address spaces at the socket (called windowing), provides steering between the socket interrupt line and the available ISA interrupts, and controls the access timings at the socket interface.

The socket controller provides windows in order to map between ISA addresses and PC card addresses. A window is an ISA address range that when accessed, causes a concurrent access to a PCMCIA socket and is either an I/O or a memory window. An I/O window provides a mapping between ISA I/O space and the PC Card I/O space whereas a memory window provides a mapping between ISA memory space and the PC Card memory space.

Each window has several attributes including its base, size, speed and state. The base and size determine the starting ISA address and range of the window. The speed represents the access time at the socket for access in this window. The state determines whether the window is a memory or an I/O window, the access width (8 or 16 bits), whether it is currently enabled, and if the window is subdivided into 16 KByte pages. Most socket controllers allow each window to consist of only a single page (page 0), having the same size as the window.

Each window page also has attributes including its offset and state. The offset is the socket address that is accessed when the start of the page is accessed. The state determine whether the page maps to common or attribute memory, if the page is enabled, and if the page should be write protected by the controller.

In addition to providing windows, the socket controller can also steer a socket interrupt request and its own status change interrupt to one of several ISA interrupt lines. This steering (or routing) as well as the other features of the socket controller is usually managed by a system software layer called Socket Services.

Socket Services

Socket Services is the lowest layer of system software that provides an industry standard, hardware independent interface to functions that control the socket power, memory and I/O mapping, and interrupt routing. These functions are described completely in the PCMCIA Socket Services Specification. Although the interfaces to these functions are hardware independent, the functions themselves are not and thus differing socket controllers will need different Socket Services implementations.

Each socket controller (or adapter) in a system that is controlled by Socket Services is assigned a number starting sequentially with zero. Each of these adapters may be controlled by only one instance of Socket Services called a Socket Services handler. A Socket Services handler may control more than one adapter. The number of Adapters in a system that are being controlled by Socket Services can be obtained using the *GetAdapterCount* function. Information about the Socket Services handler that is controlling a particular adapter can be obtained using the *GetSSInfo* function.

Each adapter controls one or more sockets which are assigned a number starting sequentially with 0. Each adapter also provides one or more windows which are assigned a number starting sequentially with 0. Each window is made up of one or more pages which are assigned a number starting sequentially with 0. An adapter may have hardware restrictions that require certain windows to be used in specific states for specific sockets. In the Cirrus Logic PD672X adapter, for example, window 13 must be used as an I/O window for socket 1. The number of sockets and windows that are controlled by an adapter can be obtained by the *InquireAdapter* function. The restrictions and capabilities of a socket or window on an adapter can be obtained by calling the *InquireSocket* or *InquireWindow* function respectively.

The Socket Services function names reflect their functions in the following manner. The capabilities and limitations of adapters, sockets, and windows are obtained by calling functions whose names begin with “Inquire” (eg *InquireAdapter*). The set of features that are currently enabled for adapters, sockets, windows, and pages are obtained by calling functions whose names begin with “Get” (eg *GetAdapter*). The set of features that are currently enabled for adapters, sockets, windows, and pages can be changed by calling functions whose names begin with “Set” (eg *SetAdapter*).

The Socket Services section in this chapter specifies how to initialize Socket Services and OS Open's binding of the Socket Services functions, that is, how the functions are invoked. A complete description of the function and parameters for each of the Socket Services functions can be found in the PCMCIA Socket Services Specification.

Card Services and Enablers

Card Services coordinates access to PC Cards, sockets, and system resources among clients. These clients may be device drivers, system utilities, or application programs. Unlike Socket Services, only one implementation of Card Services may exist in a system. Card Services must use the Socket Services interfaces to access and control the socket adapters. Card Services must also be aware and keep track of system resources in order to correctly coordinate their use.

Card Services instructs Socket Services to route status change interrupts to an interrupt owned by Card Services so that it can be interrupted when card insertions or other changes occur. Clients can also request to be informed by Card Services when these changes occur through a call back interface. Typically, a client called an enabler will be called when a card is inserted. The enabler will then use Card Services to allocate the system resources needed by the inserted device and its device driver such as ISA I/O address space and interrupts.

The Card Services functions are completely specified in the PCMCIA Card Services Specification with a minimum set of required functions specified in the ExCA Specification. Since the size and complexity of Card Services may not be appropriate for most embedded systems, OS Open provides a custom Card Services/Enabler software layer that is neither PCMCIA or ExCA compliant. OS Open's Card Services/Enabler software layer is described in the next section.

Card Services/Enabler Software Layer

OS Open supplies a Card Services/Enabler (CS/Enabler) software layer (csLib.a) that handles the dynamic configuration of cards as they are inserted into and removed from PCMCIA sockets. It allows multiple cards to share the same interrupt by determining which card generated an interrupt and calling the appropriate FLIH. It also manages the allocation of Socket Services memory and

I/O windows. The design of the CS/Enabler layer allows only one of each kind of card to be plugged into the system at any time. For example, two hard disks should not be plugged into the system at the same time.

The CS/Enabler layer performs these functions in concert with functions and a data structure that are defined by card specific enablers which are supplied with the OS Open device drivers. The data structure, called a *cs_card_info_struct* structure, is an initialized data structure that contains pointers to enabler functions as well as some other control variables. It is passed to the CS/Enabler layer when it is initialized and is used by the CS/Enabler layer to invoke enabler functions and to keep track of the state of the specific card.

When a card is inserted, the CS/Enabler code applies power to the card and scans the Card Information Structure (CIS) to determine if it has a *cs_card_info_struct* structure for the inserted card. If it does, it allocates a memory window for the card's configuration registers, routes the card's interrupt to the appropriate ISA interrupt, and then calls the card's specific enabler function to handle insertions or "re-insertions". The CS/Enabler code also allocates Socket Services windows for the card specific enabler functions as needed.

When a card is removed, the CS/Enabler code calls the card's specific enabler function to handle removals, powers off the socket, and frees and disables the Socket Services windows that were allocated for that card.

The CS/Enabler layer allows multiple cards to share the same interrupt as long as each card shares at most one interrupt and each shared interrupt is only shared between PCMCIA cards. To facilitate the sharing of interrupts, PCMCIA device drivers must use **cs_int_install()** and **cs_int_query()** instead of **ext_int_install()** and **ext_int_query()** for the interrupt they will be sharing.

In order to work correctly with the Card Services/Enabler layer, the card specific enablers must have the ability to initialize a card to an operational state after it has been removed and re-inserted. This function can be provided by the PCMCIA device driver for each card through an **ioctl()** call or some other interface. The **CS_RE_INITIALIZE** constant, which is defined in **<sys/ioctl.h>**, is intended to be used by device drivers that implement this function using the **ioctl()** interface.

Initializing the Card Services/Enabler Software Layer

The Card Services/Enabler layer is initialized by calling **cs_init()**. Since the CS/Enabler layer uses Socket Services, the Socket Services library must be initialized before the CS/Enabler layer can be initialized. If successful, **cs_init()** will return 0 indicating that the CS/Enabler layer is now initialized and is using Socket Services functions to control the PCMCIA sockets.

```
#include <sys/csLib.h>
rc = cs_init(sc_event, sc_routing, ISA_start, first_addr, cards, card1_info, ...);
```

The parameters passed to **cs_init()** are as follows:

Parameter	Meaning
<code>event_t sc_event</code>	OS Open event to use for the status change interrupts generated when cards are inserted and removed.
<code>IRQ sc_routing</code>	ISA Interrupt routing to use for the status change event.
<code>void *ISA_start</code>	System address that corresponds to ISA address 0.
<code>void *first_address</code>	System address for the first (and highest) allocated memory window.
<code>int cards</code>	Number of pointers to initialized <code>cs_card_info_struct</code> structures that follow.
<code>struct cs_card_info_struct *card1_info</code>	First pointer to a <code>cs_card_info_struct</code> structure.

The CS/Enabler layer reserves 0x1000 byte blocks in the ISA memory space in order to access a PC Card's CIS and configuration registers. A separate block is needed for each socket in the system and possibly each `cs_card_info_struct` pointer passed to **cs_init()**. *first_address* specifies the starting system address of the first and highest reserved block. Each additional blocks that is needed is reserved from the system memory map just below the most recently reserved block.

`struct cs_card_info_struct` is defined in **<csLib.h>**. The members of the `cs_card_info_struct` structure have the following meaning:

Member	Meaning
<code>card_id</code>	Unique integer card identifier (from <csLib.h>).
<code>id_string</code>	Pointer to string containing product name that is found in the card's CISTPL_VERS_1 tuple.
<code>initial_access</code>	Indicates that the function pointed to by <i>initial_fcn</i> has been called. Must be Initialized to 0.
<code>initial_fcn</code>	Pointer to card's specific enabler function that handles the first insertion.
<code>re_init_fcn</code>	Pointer to card's specific enabler function that handles subsequent insertions.
<code>removed_fcn</code>	Pointer to card's specific enabler function that handles removals.
<code>interrupt_installed</code>	Indicates that cs_int_install() has installed a FLIH for a shared interrupt. Must be initialized to 0.
<code>flih</code>	flih installed by cs_int_install() . Must be initialized to {NULL, NULL, NULL}.
<code>event</code>	<code>event_t</code> of shared interrupt (-1 if interrupt not shared).
<code>ISA_int</code>	ISA interrupt number of card interrupt (-1 if no interrupt).

<i>Adapter</i>	Number of adapter controlling the socket that the card is plugged into. Must be initialized to -1.
<i>Socket</i>	Socket number card is plugged into. Must be initialized to -1.
<i>ccr_offset</i>	Offset of card's configuration registers in card's attribute memory.
<i>rsvd_ccr</i>	System memory address reserved to access card's configuration registers. Must be initialized to NULL.
<i>plugged_state_changed</i>	Indicates that the card has been inserted or removed since last tested. Must be initialized to 0.

An example of initializing the CS/Enabler layer is provide in the putting it all together section.

Installing and Querying Shared Interrupts

First level interrupt handler (FLIH) for interrupts that are shared between multiple cards must be installed with the **cs_int_install()** and queried with the **cs_int_query()** functions.

```
rc = cs_int_install(card_id, event, &new_flih, &old_flih);
rc = cs_int_query(card_id, event, &old_flih);
```

card_id is the unique card identifier (from **<csLib.h>**) for the card. *event* is the event that the flih should be called for. *new_flih* is a pointer to a *flih_t* structure for the flih that will be installed and *old_flih* is a pointer to a *flih_t* structure that will be filled in with information about the current flih.

Note: Device drivers that share interrupts must use **ext_int_enable()** and **ext_int_disable()** very carefully since they may be disabling or enabling interrupts for other devices. In general, the device drivers should call **ext_int_enable()** after calling **cs_int_install()** to ensure that the device interrupt is enabled. If the device interrupt needs to be subsequently disabled it should be masked off in the device.

Note: If a device driver will have exclusive use of an OS Open interrupt event, it should use the *ext_int_xxxx* functions and not the *cs_int_xxxx* functions.

Allocating Socket Services Windows

Device specific enablers that need either a memory window an I/O window can call **cs_get_mem_window()** or **cs_get_io_window()**.

```
window = cs_get_mem_window(Adapter, Socket);
window = cs_get_io_window(Adapter, Socket);
```

These functions reserve an available window for the *Socket* socket on the *Adapter* adapter and returns its window number to the caller. The window remains reserved until the PC Card that is plugged into the specified socket is removed.

Once removed, the window is disabled and freed so that it is available for a subsequent **cs_get_mem_window()** or **cs_get_mem_window()** call.

Determining the Insertion State of a Card

The insertion state of a card can be determined by calling **cs_card_status()** and **cs_stateless_card_status()**.

```
state = cs_card_status(card_id);  
state = cs_stateless_card_status(card_id);
```

cs_card_status() and **cs_stateless_card_status()** returns 0 if the card identified by *card_id* is currently plugged into a socket in the system and has not been inserted since the last call to **cs_card_status()**. If the card is plugged in but was not inserted since the last call to **cs_card_status()**, both functions will return 1. If the card is not currently plugged into a socket in the system, **cs_card_status()** and **cs_stateless_card_status()** will return -1.

Note: A call to **cs_card_status()** updates an internal flag that is used to differentiate between a card that has been inserted subsequent to the last call to **cs_card_status()** and a card that has been inserted prior to the last call to **cs_card_status()**. **cs_stateless_card_status()** returns the same information as **cs_card_status()** but does not update the internal flag.

Advanced Card Services/Enabler Card Matching

When a card is inserted, the CS/Enabler code reads the card's product name from the CISTPL_VERS_1 tuple to determine what kind of card it is and which enabler function to call. This scheme works well when a limited number of cards need to be supported. The CS/Enabler code can also match other tuples in the card's CIS in the event that the CISTPL_VERS_1 tuple alone is not sufficient to determine which enabler function to invoke. This is accomplished by initializing the *id_string* member of a card's *cs_card_info_struct* so that it points to a linked list of *cs_tuple_spec* structures instead of a character string.

Each *cs_tuple_spec* structure contains a tuple code and a pointer to a linked list of *cs_match_spec* structures that describe characteristics of that tuple. An inserted card is considered a match if a tuple represented by every *cs_tuple_spec* in the linked list of *cs_tuple_spec* structures is present in the CIS. A tuple represented by a *cs_tuple_spec* is considered present if every characteristic described by the linked list of *cs_match_spec* structures is satisfied. Since a CIS may have many tuples with the same tuple code, the CS/Enabler code tests each tuple with a matching tuple code against the linked list of *cs_match_spec* structures. If none of the tuples satisfies each characteristic, the tuple is not present and the card is not considered a match.

struct cs_tuple_spec is defined in **<csLib.h>**. The members of the *cs_tuple_spec* structure have the following meaning:

Member	Meaning
--------	---------

<i>vers_1_string</i>	character with value '0' indicating that this is a <i>cs_tuple_spec</i> and not a character string.
<i>tuple_id</i>	tuple code of tuple to match.
<i>offset</i>	integer scratch value used by CS/Enabler code.
<i>first_match</i>	pointer to first <i>cs_match_spec</i> for this tuple.
<i>next</i>	pointer to next <i>cs_tuple_spec</i> in this list.

struct cs_match_spec is defined in **<csLib.h>**. The members of the *cs_match_spec* structure have the following meaning:

Member	Meaning
<i>type</i>	type of match required (described below).
<i>spec</i>	pointer to structure detailing match requirements (described below).
<i>next</i>	pointer to next <i>cs_tuple_spec</i> in this list.

There are currently six match types that can be used to specify the contents of a tuple. The six match types are: *CS_MATCH_MASK*, *CS_MATCH_EXACT*, *CS_MATCH_EXCLUDE*, *CS_MATCH_GREATER*, *CS_MATCH_STRING*, and *CS_MATCH_POSITION*. Depending on the match type, the *spec* pointer in a *cs_match_spec* structure must point to one of four types of structures. The are: *cs_mask_match*, *cs_offset_match*, *cs_string_match*, and *cs_position_match*. These structures are described below.

CS_MATCH_MASK

If the *type* field in a *cs_match_spec* is set to *CS_MATCH_MASK*, the *spec* pointer must point to a *cs_mask_match* structure. *struct cs_mask_match* is defined in **<csLib.h>**. The members of the *cs_mask_match* structure have the following meaning.

Member	Meaning
<i>offset</i>	offset from start of tuple of byte to compare.
<i>mask</i>	byte value ANDED with tuple byte being compared.
<i>value</i>	required result of AND of compared byte and <i>mask</i> .

If the result of the AND of the byte to compare and *mask* equals *value*, this characteristic of the tuple is satisfied.

CS_MATCH_EXACT, CS_MATCH_EXCLUDE, and CS_MATCH_GREATER

If the *type* field in a *cs_match_spec* is set to *CS_MATCH_EXACT*, *CS_MATCH_EXCLUDE*, or *CS_MATCH_GREATER*, the *spec* pointer must point to a *cs_offset_match* structure. *struct cs_offset_match* is defined in **<csLib.h>**. The members of the *cs_offset_match* structure have the following meaning:

Member	Meaning
<i>offset</i>	offset from start of tuple of byte to compare.

value value compared with tuple byte being compared.

This characteristic of the tuple is satisfied if any of the following are true:

- *type* is *CS_MATCH_EXACT* and the value of the byte to be compared equals *value*.
- *type* is *CS_MATCH_EXCLUDE* and the value of the byte to be compared does not equal *value*.
- *type* is *CS_MATCH_GREATER* and the value of the byte to be compared is greater than *value*.

CS_MATCH_STRING

If the *type* field in a *cs_match_spec* is set to *CS_MATCH_STRING*, the *spec* pointer must point to a *cs_string_match* structure. *struct cs_string_match* is defined in **<csLib.h>**. The members of the *cs_string_match* structure have the following meaning:

Member	Meaning
<i>string</i>	pointer to character string to match.

If the string pointed to by *string* exists in the tuple being compared, this characteristic of the tuple is satisfied.

CS_MATCH_POSITION

If the *type* field in a *cs_match_spec* is set to *CS_MATCH_POSITION*, the *spec* pointer must point to a *cs_position_match* structure. *struct cs_position_match* is defined in **<csLib.h>**. The members of the *cs_position_match* structure have the following meaning.

Member	Meaning
<i>follows</i>	pointer to a <i>cs_tuple_spec</i> structure.

If a tuple represented by the *cs_tuple_spec* structure pointed to by *follows* has been determined to be present and precedes this tuple in the CIS, this characteristic of the tuple is satisfied.

Dynamic Configuration Register Offsets

Since this advanced card matching feature allows the enabler functions to be invoked based on a successful complex pattern match, it may not be possible to statically specify the offset of all of the cards' configuration registers in the *cs_card_info_struct* for the cards described by a particular pattern. In this case, the *ccr_offset* member of the *cs_card_info_struct* should be initialized to -1. This will prevent the CS/Enabler code from allocating a memory window for accessing the configuration registers. If the inserted card uses any interrupts that are shared with other cards, the enabler functions must call the **cs_set_ccr_offset()** function every time the card is inserted. **cs_set_ccr_offset()** allocates a memory window for the newly inserted card's configuration registers (if needed) and calculates and

returns the system address that can be used to access those registers. This will allow the CS/Enabler code to correctly dispatch the card's FLIH when the card asserts its interrupt request.

Socket Services

OS Open provides a Exchangeable Card Architecture (ExCA) version 1.50 compatible Socket Services implementation for the Cirrus Logic PD6710 and PD672X PCMCIA host adapters. This section describes the binding that OS Open uses to implement Socket Services as well as a brief description of each Socket Services function provided. A complete description of each Socket Services function can be found in the PCMCIA Socket Services Specification (release 2.1).

The Socket Services functions are invoked by passing a function number to the **socket_services()** function along with the parameters used by the specified Socket Services function. The Socket Services function numbers as well as the Socket Services type definitions and constants can be found in **<ssLib.h>**. If the Socket Services library has been initialized and there's no outstanding Socket Services call, the desired function will be invoked. Since Socket Services is not generally reentrant, **socket_services()** will return *BUSY* if it is called while a Socket Services function is executing. The exception to this is the *Acknowledge_Interrupt* function that should be invoked in response to a status change interrupt.

Initializing the Socket Services Library

The Socket Services library is initialized by calling **ss_init()** with at least four passed parameters. The first parameter is the adapter count which specifies the number of PCMCIA socket adapters that will be controlled by this Socket Services handler. For each of these adapters three additional positional parameters must be supplied that specify the base system address of the adapter, the device index of the adapter, and the number of sockets that the adapter is controlling. The device index of the adapter is a byte value of either 0x00 or 0x80 depending on the initial level of the SPKR_OUT/C_SEL pin of the PD6710 or PD672X chip. Since **ss_init()** must not be called more than once, all adapters that will be controlled by this Socket Services handler must be included in the adapter count and subsequent parameters. More than one call to **ss_init()** will result in a return value of -1 and errno set to *EINVAL*.

ss_init() initializes the Socket Services library and PCMCIA socket adapter hardware. **ss_init()** returns zero if the Socket Services library initialization was successful, indicating that **socket_services()** can now be called to invoke Socket Services functions.

```
rc = ss_init(2, 0x800003e0, 0x00, 1, 0x800003e0, 0x80, 2);
```


In this example, Socket Services is being initialized to control two adapters. The first adapter controls one socket and has a base address and device index of 0x800003e0 and 0x00 respectively. The second adapter controls two sockets and has a base address and device index of 0x800003e0 and 0x80 respectively. Notice that the device index allows more than one adapter to share the same base address.

Socket Services Functions

The implemented Socket Services functions can be divided into the following categories:

- Non-Specific function that applies to the Socket Services interface in general.
 - *GetAdapterCount*
- Adapter functions that deal with a specific adapter.
 - *GetSSInfo*
 - *InquireAdapter*
 - *GetAdapter*
 - *SetAdapter*
 - *AcknowledgeInterrupt*
- Socket functions that deal with a specific socket on an adapter.
 - *InquireSocket*
 - *GetSocket*
 - *SetSocket*
 - *GetStatus*
 - *ResetSocket*
- Window functions that deals with memory and I/O windows on an adapter.
 - *InquireWindow*
 - *GetWindow*
 - *SetWindow*
 - *GetPage*
 - *SetPage*

Non-Specific Function

GetAdapterCount

The *GetAdapterCount* function reports the number of adapters supported by all Socket Services handlers in the system. It is also used to determine if any Socket Services handlers are installed and initialized.

```
rc = socket_services(GET_ADP_CNT, &TotalAdapters, &Signature);
```

The variable arguments are (in order):

int *TotalAdapters* Count of adapters.
SIGNATURE *SignatureSS*_SIGNATURE signature.

If the Socket Services library is initialized, *GetAdapterCount* will return *SUCCESS*, report the number of adapters in the system, and will fill in the Socket Services signature *SS_SIGNATURE*. Otherwise, *GetAdapterCount* will return *BAD_FUNCTION*.

Adapter Functions

GetSSInfo

The *GetSSInfo* function reports the compliance level of the Socket Services handler controlling the specified adapter. It also reports the number of adapters being controlled and the adapter number of the first adapter being controlled by that handler.

```
rc = socket_services(GET_SS_INFO, Adapter, &Compliance, &NumAdapters,  
                      &FirstAdapter);
```

The variable arguments are (in order):

int <i>Adapter</i>	Physical adapter number (input).
BCD <i>Compliance</i>	BCD formatted output (set to <i>PCMCIA_LEVEL</i>).
int <i>NumAdapters</i>	Number of adapters controlled by this specific Socket Services handler.
int <i>FirstAdapter</i>	Adapter number of first adapter controlled by this specific Socket Services handler.

If the specified adapter is being controlled by the Socket Services library, *GetSSInfo* will return *SUCCESS* and update the output variables. Otherwise *GetSSInfo* will return *BAD_ADAPTER*.

InquireAdapter

The *InquireAdapter* function reports information about the capabilities of the specified adapter.

```
rc = socket_services(INQ_ADAPTER, Adapter, &pBuffer, &NumSockets,  
                      &NumWindows, &NumEDCs);
```

The variable arguments are (in order):

int <i>Adapter</i>	Physical adapter number (input).
AISTRUCT <i>pBuffer</i>	Adapter information structure to be filled with information about the adapter.
int <i>NumSockets</i>	Number of sockets provided by this adapter.
int <i>NumWindows</i>	Number of windows provided by this adapter.
int <i>NumEDCs</i>	Number of Error Detection Code generators provided by this adapter.

If the specified adapter is being controlled by the Socket Services library, *InquireAdapter* will return *SUCCESS* and update the output variables. Otherwise *InquireAdapter* will return *BAD_ADAPTER*.

GetAdapter

The *GetAdapter* function reports the current configuration of the specified adapter.

```
socket_services(GET_ADAPTER, Adapter, &State, &SCRouting);
```

The variable arguments are (in order):

int Adapter	Physical adapter number (input).
FLAGS8 State	Current state of the adapter H/W.
IRQ SCRouting	Status change interrupt routing status.

If the specified adapter is being controlled by the Socket Services library, *GetAdapter* will return *SUCCESS* and update the output variables. Otherwise *GetAdapter* will return *BAD_ADAPTER*.

SetAdapter

The *SetAdapter* function sets the configuration of the specified adapter.

```
socket_services(SET_ADAPTER, Adapter, &State, &SCRouting);
```

The variable arguments are all inputs and are (in order):

int Adapter	Physical adapter number.
FLAGS8 State	Requested state of the adapter H/W.
IRQ SCRouting	Requested status change interrupt routing.

If the specified adapter is being controlled by the Socket Services library and *SCRouting* is a supported IRQ value, *SetAdapter* will return *SUCCESS*. Otherwise, *SetAdapter* will return either *BAD_ADAPTER* or *BAD_IRQ*.

The Socket Services library does not support power down states. Therefore, the requested state of the adapter is ignored.

AcknowledgeInterrupt

The *AcknowledgeInterrupt* function reports which sockets on the specified adapter experienced a changed in status and enables subsequent status change interrupts.

```
socket_services(ACK_INTERRUPT, Adapter, &Sockets);
```

The variable arguments are (in order):

int Adapter	Physical adapter number (input).
SKTBITS Sockets	Bit significant representation of which socket or sockets has experienced a change in status.

If the specified adapter is being controlled by the Socket Services library, *AcknowledgeInterrupt* will return *SUCCESS* and update the output variable. Otherwise *AcknowledgeInterrupt* will return *BAD_ADAPTER*.

Socket Functions

InquireSocket

The *InquireSocket* function reports information about the capabilities of the specified socket.

```
rc = socket_services(INQ_SOCKET, Adapter, Socket, &pBuffer, &SCIntCaps,
&SCRptCaps, &CtlIndCaps);
```

The variable arguments are (in order):

int <i>Adapter</i>	Physical adapter number (input).
int <i>Socket</i>	Physical socket number on the adapter (input).
SISTRUCT <i>pBuffer</i>	Socket information structure to be filled in with information about the window.
FLAGS8 <i>SCIntCaps</i>	Events which can trigger a Status Change interrupt.
FLAGS8 <i>SCRptCaps</i>	Status Change events that the socket is capable of reporting.
FLAGS8 <i>CtlIndCaps</i>	Control and indicator capabilities of the socket.

If the specified adapter and socket is being controlled by the Socket Services library, *InquireSocket* will return *SUCCESS* and update the output variables. Otherwise *InquireSocket* will return either *BAD_ADAPTER* or *BAD_SOCKET*.

GetSocket

The *GetSocket* function reports the current configuration of the specified socket.

```
socket_services(GET_SOCKET, Adapter, Socket, &SCIntMask, &VccLevel,
&Vpp1Level, &Vpp2Level, &State, &CtlInd, &IREQRouting, &IFTType);
```

The variable arguments are (in order):

int <i>Adapter</i>	Physical adapter number (input).
int <i>Socket</i>	Physical socket number on the adapter (input).
FLAGS8 <i>SCIntMask</i>	Current setting of mask for events that generate a Status Change interrupt.
int <i>VccLevel</i>	Current power level of Vcc signal.
int <i>Vpp1Level</i>	Current power level of Vpp1 signal.
int <i>Vpp2Level</i>	Current power level of Vpp2 signal.
FLAGS8 <i>State</i>	Latched values representing state changes experienced by the socket hardware.
FLAGS8 <i>CtlInd</i>	Current settings of socket controls and indicators.

IRQ <i>IREQRouting</i>	PC Card IREQ routing status.
FLAGS8 <i>IFType</i>	Current interface type.

If the specified adapter and socket is being controlled by the Socket Services library, *GetSocket* will return *SUCCESS* and update the output variables. Otherwise *GetSocket* will return either *BAD_ADAPTER* or *BAD_SOCKET*.

SetSocket

The *SetSocket* function sets the configuration of the specified socket.

```
socket_services(SET_SOCKET, Adapter, Socket, &SCIntMask, &VccLevel,
                &Vpp1Level, &Vpp2Level, &State, &CtlInd, &IREQRouting, &IFType);
```

The variable arguments are all inputs and are (in order):

int <i>Adapter</i>	Physical adapter number.
int <i>Socket</i>	Physical socket number on the adapter.
FLAGS8 <i>SCIntMask</i>	Requested setting of mask for events that generate a Status Change interrupt.
int <i>VccLevel</i>	Requested power level of Vcc signal.
int <i>Vpp1Level</i>	Requested power level of Vpp1 signal.
int <i>Vpp2Level</i>	Requested power level of Vpp2 signal.
FLAGS8 <i>State</i>	Latched values representing state changes experienced by the socket hardware to reset.
FLAGS8 <i>CtlInd</i>	Requested settings of socket controls and indicators.
IRQ <i>IREQRouting</i>	Requested PC Card IREQ routing status.
FLAGS8 <i>IFType</i>	Requested interface type.

If the specified adapter and socket is being controlled by the Socket Services library and all of the other arguments are supported and valid, *SetSocket* will return *SUCCESS*. Otherwise *SetSocket* will return *BAD_ADAPTER*, *BAD_IRQ*, *BAD_SOCKET*, *BAD_TYPE*, *BAD_VCC*, or *BAD_VPP*.

The supported Cirrus chips do not support distinct VPP1 and VPP2 signals. Therefore *Vpp1Level* must be equal to *Vpp2Level*.

GetStatus

The *GetStatus* function returns the current status of the card, socket, controls and indicators for the identified socket.

```
socket_services(GET_STATUS, Adapter, Socket, &CardState, &SocketState,
                &CtlInd, &IREQRouting, &IFType);
```

The variable arguments are (in order):

int <i>Adapter</i>	Physical adapter number (input).
int <i>Socket</i>	Physical socket number on the adapter (input).

FLAGS8 <i>CardState</i>	Instantaneous state.
FLAGS8 <i>SocketState</i>	Latched values representing state changes experienced by the socket hardware. (Same value as <i>State</i> returned by <i>GetSocket</i>).
FLAGS8 <i>CtlInd</i>	Current settings of socket controls and indicators. (Same value as <i>CtlInd</i> returned by <i>GetSocket</i>).
IRQ <i>IREQRouting</i>	PC Card IREQ routing status. (Same value as <i>IREQRouting</i> returned by <i>GetSocket</i>).
FLAGS8 <i>IFType</i>	Current interface type. (Same value as <i>IFType</i> returned by <i>GetSocket</i>).

If the specified adapter and socket is being controlled by the Socket Services library, *GetStatus* will return *SUCCESS* and update the output variables. Otherwise *GetStatus* will return either *BAD_ADAPTER* or *BAD_SOCKET*.

ResetSocket

The *ResetSocket* function resets the PC Card in the specified socket and returns the socket hardware to its power-on default state.

```
socket_services(RESET_SOCKET, Adapter, Socket);
```

The variable arguments are all inputs and are (in order):

int <i>Adapter</i>	Physical adapter number.
int <i>Socket</i>	Physical socket number on the adapter.

If the specified adapter and socket is being controlled by the Socket Services library, *ResetSocket* will return *SUCCESS*. Otherwise *ResetSocket* will return either *BAD_ADAPTER* or *BAD_SOCKET*.

Window Functions

InquireWindow

The *InquireWindow* function reports information about the capabilities of the specified window.

```
socket_services(INQ_WINDOW, Adapter, Window, &pBuffer, &WndCaps, &Sockets);
```

The variable arguments are (in order):

int <i>Adapter</i>	Physical adapter number (input).
int <i>Window</i>	Physical window number on adapter (input).
WISTRUCT <i>pBuffer</i>	Window information structure to be filled in with information about the window.
FLAGS8 <i>WndCaps</i>	Capabilities of the window.

SKTBITS *Sockets* Bit significant representation of which socket can be assigned to the specified window.

If the specified adapter and window is being controlled by the Socket Services library, *InquireWindow* will return *SUCCESS* and update the output variables. Otherwise *InquireWindow* will return either *BAD_ADAPTER* or *BAD_WINDOW*.

GetWindow

The *GetWindow* function reports the current configuration of the specified window.

```
socket_services(GET_WINDOW, Adapter, Window, &Socket, &Size, &State,
&Speed, &Base);
```

The variable arguments are (in order):

int <i>Adapter</i>	Physical adapter number (input).
int <i>Window</i>	Physical window number on adapter (input).
int <i>Socket</i>	Physical socket the window is currently assigned.
int <i>Size</i>	Window's current size.
FLAGS8 <i>State</i>	Current state of window hardware.
SPEED <i>Speed</i>	Actual access speed being used by the window.
int <i>Base</i>	Current base address of the window.

If the specified adapter and window is being controlled by the Socket Services library, *GetWindow* will return *SUCCESS* and update the output variables. Otherwise *GetWindow* will return either *BAD_ADAPTER* or *BAD_WINDOW*.

SetWindow

The *SetWindow* function sets the current configuration of the specified window.

```
socket_services(SET_WINDOW, Adapter, Window, &Socket, &Size, &State,
&Speed, &Base);
```

The variable arguments are all inputs and are (in order):

int <i>Adapter</i>	Physical adapter number.
int <i>Window</i>	Physical window number (on adapter).
int <i>Socket</i>	Physical socket the window is to be assigned.
int <i>Size</i>	Window's requested size.
FLAGS8 <i>State</i>	Requested state of window hardware.
SPEED <i>Speed</i>	Requested access speed to be used by the window.
int <i>Base</i>	Requested base address of the window.

If the specified adapter and socket is being controlled by the Socket Services library and all of the other arguments are supported and valid, *SetWindow* will return *SUCCESS*. Otherwise *SetWindow* will return *BAD_ADAPTER*,

BAD_ATTRIBUTE, *BAD_BASE*, *BAD_SIZE*, *BAD_SOCKET*, *BAD_SPEED*, *BAD_TYPE*, or *BAD_WINDOW*.

GetPage

The *GetPage* function reports the current configuration of the specified page. It is only valid for memory windows (WS_IO is reset for the window).

```
socket_services(GET_PAGE, Adapter, Window, Page, &State, &Offset);
```

The variable arguments are (in order):

int <i>Adapter</i>	Physical adapter number (input).
int <i>Window</i>	Physical window number on adapter (input).
int <i>Page</i>	Page number within the window (input).
FLAGS8 <i>State</i>	Current state of the specified page.
int <i>Offset</i>	Offset of card's memory currently being mapped into system memory by this page.

If the specified adapter, window, and page is being controlled by the Socket Services library, *GetPage* will return *SUCCESS* and update the output variables. Otherwise *GetPage* will return *BAD_ADAPTER*, *BAD_PAGE*, or *BAD_WINDOW*.

SetPage

The *SetPage* function configures the specified page. It is only valid for memory windows (WS_IO is reset for the window).

```
socket_services(SET_PAGE, Adapter, Window, Page, &State, &Offset);
```

The variable arguments are all inputs and are (in order):

int <i>Adapter</i>	Physical adapter number.
int <i>Window</i>	Physical window number (on adapter).
int <i>Page</i>	Page number within the window.
FLAGS8 <i>State</i>	Requested state of the specified page.
int <i>Offset</i>	Offset of card's memory to be mapped into system memory by this page.

If the specified adapter, window, and page is being controlled by the Socket Services library and all of the other arguments are supported and valid, *SetPage* will return *SUCCESS*. Otherwise *SetPage* will return *BAD_ADAPTER*, *BAD_ATTRIBUTE*, *BAD_OFFSET*, *BAD_PAGE*, or *BAD_WINDOW*.

PCMCIA ATA/IDE Device Driver

OS Open provides PCMCIA ATA/IDE support for the IBM and Integral 105 MB and 260 MB hard disks through the use of the pataLib block device driver. The pataLib

device driver accepts standard block device driver requests (**ioctl()** and **strategy()**) from an application or file system and converts them into the ATA/IDE commands that are needed to satisfy the request. The pataLib device driver issues these ATA/IDE commands and oversees their execution on an ATA/IDE device.

Prior to installing any PCMCIA device driver, the PCMCIA socket controller must be initialized correctly so that the device driver can communicate with the PCMCIA device. This is usually accomplished by a higher level layer of software called “Card Services” or “enablers” that makes calls to low level code called “Socket Services”. An enabler that is designed to work with OS Open’s CS/Enabler software layer is included with the pataLib block device driver and is documented below. For more information about enablers and Socket Services refer to the sections above.

Device Driver Installation

The pataLib device driver is installed by calling **driver_install()**. Following is an example of the pataLib device driver installation.

```
#include <sys/pataLib.h>
int devhandle,rc;
rc=driver_install(&devhandle, pata_init);
```

pata_init() is declared in the file **<pataLib.h>** as follows:

```
int pata_init(driver_t *dsw, va_list vargs)
```

Upon successful installation, **driver_install()** returns 0; otherwise –1 is returned. For more information on **driver_install()**, refer to the *OS Open Programmer's Reference*.

Device Installation

After the pataLib device driver is installed, named devices can be created using **device_install()**. Following is an example of device installation.

```
rc = device_install("/pata0", BLKTYPE, devhandle, 0, 0x80000170,
0x80000376, 14);
```

For device installation, *devhandle* is the value obtained from the **driver_install()**. Device type BLKTYPE is defined in **<sys/devDriver.h>**.

Additional parameters passed in the **device_install()** call are as follows:

Parameter	Meaning
First Parameter	Drive number (Master:0, Slave:1).
Second Parameter	System address of ATA Task File.
Third Parameter	System address of ATA Control Block.
Fourth Parameter	External Interrupt used for ATA interrupt.

These are positional parameters. The system address of the ATA task file and control block are controlled by the PCMCIA configuration registers. The device specific enabler included with this device driver will cause the task file and control block to be located at ISA I/O address 0x170 and 0x376 respectively.

Upon successful installation, **device_install()** returns 0; otherwise -1 is returned. For more information on **device_install()**, refer to the *OS Open Programmer's Reference*.

Uninstalling a PCMCIA ATA/IDE Device

The **device_uninstall()** call can be used to uninstall a particular device that does not have any outstanding opens against it. Following is an example of the **device_uninstall()** call.

```
rc=device_uninstall("/pata0");
```

If successful, **device_uninstall()** returns 0; otherwise -1 is returned. For more information on **device_uninstall()**, refer to the *OS Open Programmer's Reference*.

Opening a PCMCIA ATA/IDE Device

After the device is installed, the **open()** system call can be used to open a particular device. Following is an example of the **open()** system call used against a device:

```
fd_p = open("/hd0", O_RDWR);
```

Note: The oflag parameter, O_RDWR in this example, which is passed in the open call, is ignored by the device driver.

Note: All opened devices must be closed before the device can be uninstalled.

When successful, **open()** returns a nonnegative integer file descriptor. Otherwise **open()** returns -1. For more information on **open()**, refer to the *OS Open Programmer's Reference*.

Closing a PCMCIA ATA/IDE Device

After a particular device has been opened, the **close()** system call can be used to close the device. Following is an example of the **close()** system call used against a device. *fd_p* is the value obtained from the open() call.

```
rc = close(fd_p);
```

When successful, **close()** returns 0. Otherwise **close()** returns -1. For more information on **close()**, refer to the *OS Open Programmer's Reference*.

Reading and Writing

After successfully installing and opening a PCMCIA ATA/IDE device, the **strategy()** function is used to read and write blocks. The pataLib device driver

translates the **strategy()** function call into the ATA/IDE commands that are needed to satisfy the request. The pataLib device driver issues these ATA/IDE commands and oversees their execution on an ATA/IDE device.

An ATA/IDE device, like other block devices, is treated as a linear collection of 512 byte blocks. Each block is assigned a number called its block ID with 0 being assigned to the first accessible block. The **strategy()** call to an ATA/IDE device passes a pointer to an initialized structure that provides the type of the request (read or write), the memory address of the request, the first block ID of the request, and the number of sequential blocks requested.

If the **strategy()** call returns 0 the access was successful, otherwise it was not and *errno* will be set. An example of reading, clearing, and writing blocks is shown below. *fd_p* is the value obtained from the *open()* call.

```
unsigned char myblock[512 * 3]; /* buffer space for 3 blocks */
block_req_t bk_request;

bk_request.request_type = BLOCK_READ;
bk_request.block_buffer = myblock; /* destination of read data */
bk_request.block_id = 9; /* read starts with block 9 (the 10th block) */
bk_request.block_count = 3; /* reads blocks 9, 10, and 11 */
if (strategy(fd_p, bk_request) != 0) trouble();
memset(&myblock[512], 0x00, 512); /* clear data in middle block */
bk_request.request_type = BLOCK_WRITE;
if (strategy(fd_p, bk_request) != 0) trouble();
```

For more information on **strategy()**, refer to the *OS Open Programmer's Reference*.

I/O Control

An **ioctl()** call issued against the pataLib device driver accepts the following commands. If the **ioctl()** call returns 0 the command was successful, otherwise it was not and *errno* will be set. In each of these commands, *fd_p* is the value obtained from the **open()** call.

QDEVATTR

This command causes the pataLib device driver to return the PCMCIA ATA/IDE device's attributes in the structure pointed to by a passed parameter.

```
block_dev_attr_t blkattr;
rc = ioctl(fd_p, QDEVATTR, &blkattr);
```

block_dev_attr_t is defined in **<sys/blockdev.h>**. The *block_dev_attr_t* variable pointed to by *blkattr* will be set to *REMOVABLE_MEDIA* to indicate that the block device supports removable media.

IOCINFO

This command causes the pataLib device driver to return information about the PCMCIA ATA/IDE device in the structure pointed to by a passed parameter.

```
devinfo_t devinfo;  
rc = ioctl(fd_p, IOCINFO, &devinfo);
```

devinfo_t is defined in **<sys/devinfo.h>**. The members of the *devinfo_t* structure have the following meaning:

Member	Meaning
<i>devtype</i>	Type of device (e.g. ATA disk).
<i>devsubtype</i>	Sub-type of device (e.g. Physical Volume).
<i>un.atadk.numblks</i>	Total number of blocks.
<i>un.atadk.max_request</i>	Maximum number of blocks per request.

BLKMEDIA_CHANGE

This command causes the pataLib device driver to check if the media has been removed and possibly replaced. The driver determines this by calling the *cs_card_status()* function which is not provided in the pataLib code. This function should be provided in the Card Services or Enabler software layer.

```
rc = ioctl(fd_p, BLKMEDIA_CHANGE);
```

The return code from the **ioctl()** call indicates one of the following conditions:

0	Media has not been changed.
1	Media has been changed. A different hard disk is in the PCMCIA slot.
-1	Media has been removed or an invalid hard disk is now in the PCMCIA slot.

CS_RE_INITIALIZE

This command causes the pataLib device driver to re-initialize the attached ATA/IDE device so that it can be accessed. This command is typically used by the Card Services or Enabler software layer after the ATA/IDE device has been removed and re-inserted in a PCMCIA socket.

```
rc = ioctl(fd_p, CS_RE_INITIALIZE);
```

Using the PCMCIA ATA/IDE Device Enabler

The PCMCIA ATA/IDE device enabler invokes Socket Services functions to initialize the socket controller, and installs and configures the pataLib device driver and optionally the FAT file system when a PCMCIA ATA/IDE device is plugged into a socket. A *cs_card_info_struct* structure named *pata_info* is available to be passed to the CS/Enabler software layer initialization routine. The parameters in

this structure and others used by this enabler should be initialized prior to initializing the CS/Enabler software layer.

Initializing the Enabler Parameters

The parameters used in the *pata_info* structure and by the pataLib enabler are initialized by calling **cs_pata_parm_init()**.

```
rc = cs_pata_parm_init(pata_name, pata_event, pata_ISA_int, install_fat,
fat_dh, fat_name, unit_no, offset);
```

The parameters passed to **cs_pata_parm_init()** are as follows:

Parameter	Meaning
char * <i>pata_name</i>	Name to bind the device to in the device_install() call for the PCMCIA ATA/IDE device.
event_t <i>pata_event</i>	OS Open event generated by the device interrupts.
int <i>pata_ISA_int</i>	The ISA interrupt number of the event generated by device interrupts.
int <i>install_fat</i>	If non-zero, the enabler will attempt to configure the device into the FAT file system. If zero, the enabler will not configure the device into the FAT file system and will ignore the remaining parameters.
int <i>fat_dh</i>	The device driver handle for the FAT file system that was obtained from driver_install() .
char * <i>fat_name</i>	File system name that will be used to access files on the device (eg. "/fat").
int <i>unit_no</i>	Unit number of device.
int <i>offset</i>	The block offset of the start of the FAT file system.

If successful, **cs_pata_parm_init()** will return 0, otherwise it will return a non-zero value.

Putting it All Together

The **pcmcia_install()** function listed below is an example of code that installs and initializes the various OS Open PCMCIA software components needed to support a PCMCIA ATA/IDE hard disk in a system.

```
#define PATA_EVENT      14
#define PATA_ISA_INT    14
#define SC_EVENT        11
#define SC_ROUTING      (IRQ_ENABLED | IRQ_HIGH | 11)
```

```

int pcmcia_install(void)
{
    int rc;
    /* initialize socket services */
    rc = ss_init(1, 0x800003e0, 0x00, 2);
    if (rc)
    {
        printf("ss_init(1, 0x800003e0, 0x00, 2); failed rc: %d\n", rc);
        return rc;
    }
    /* install the FAT file system */
    rc = driver_install(&cs_pata_ffsdh, fatfs_init, 4, 16);
    if (rc)
    {
        printf("FAT file system driver_install() failed rc: %d\n", rc);
        return rc;
    }
    /* initialize the PCMCIA ATA/IDE enabler parameters */
    cs_pata_parm_init("/pata", PATA_EVENT, PATA_ISA_INT, true,
                     cs_pata_ffsdh, "/hd0", 0, 17);
    /* initialize the PCMCIA Card Services/Enabler software layer */
    rc = cs_init(SC_EVENT, SC_ROUTING, (void *) 0xc0000000, (void *)
0xc0fff000, 1, &pata_info);
    if (rc)
    {
        printf("cs_init(); failed rc: %d\n", rc);
        return rc;
    }
    return rc;
}

```

In this example, Socket Services is initialized to control one adapter with two sockets. Next the FAT file system installed. Then the parameters for the PCMCIA ATA/IDE enabler are initialized. Finally, the Card Services/Enabler software layer is initialized. If **pcmcia_install()** completes successfully and a PCMCIA hard disk

(that contains a FAT partition at offset 17) is inserted into one of the two slots, the enabler will configure the hard disk into the FAT file system and it will be accessible in the /hd0 file system.

Chapter 12. OS Open Networking Features

The OS Open operating system provides a rich set of networking features based on draft POSIX and de facto networking standards.

OS Open networking, based primarily on TCP/IP, provides the following network interfaces:

- Ethernet
- Serial Line IP (SLIP)
- Token-ring

Network interfaces for other networks can be easily integrated to work with the TCP/IP library. For more information about adding network interfaces, see Chapter 13, “Writing OS Open Network Interface Drivers.”

Introducing the TCP/IP Library

The TCP/IP library provides general networking support compatible with the 4.3.2 UNIX Berkeley Software Distribution (BSD) implementation. Socket support and TCP, IP, ICMP, and UDP protocol stacks are provided in the TCP/IP library.

The TCP/IP library does not contain a network interface. Instead, the library uses separate network interface modules; three are provided, and others can be added.

The TCP/IP library must be initialized by **tcPIP_init()** before functions in the library can be used.

Parameters passed to **tcPIP_init()** contain:

- Local host name

This is the name of the machine being configured.

- Number of memory clusters

This is the amount of storage, in 4096-byte clusters, that can be used to buffer transmitted data.

- Number of memory buffers

This is the amount of storage, in 128-byte buffers, that can be used to buffer transmitted data.

Internally, the TCP/IP library uses memory buffers instead of memory clusters. By using memory buffers (at the expense of network performance for large packets), greater overall memory usage efficiency can be obtained. Memory clusters are included for use by user-supplied network interfaces.

At least one memory buffer and one cluster must be specified in a **tcpip_init()** call. The maximum number of memory clusters and memory buffers is limited by heap size.

Sockets and file descriptors can be used interchangeably in calls to **read()**, **write()**, **select()**, **ioctl()** and **close()**. Other functions that take a file descriptor as an argument can also be used with sockets. Table 12-1 lists **ioctl()** commands that can be issued for socket file descriptors.

Table 12-1. I/O Control Commands for Socket File Descriptors

Command	Data	Description
FIONREAD	int *	Returns number of bytes ready to read
FIONBIO	int *	Sets or clears non blocking flag
FIOASYNC	int *	Sets or clears asynchronous I/O flag (SIGUSER2 provides asynchronous notification)
SIOCATMARK	int *	Gets out-of-band data flag
SIOCSPGRP	pthread_t *	On receipt of out-of-band data signal, SIGUSER1 is sent to the thread specified in pthread_t (default is to send SIGUSER1 to the real-time executive)
SIOCGPGRP	pthread_t *	Gets thread ID of a thread that receives SIGUSER1 as a result of out-of-band data (thread ID 0 indicates that the signal will be sent to the real-time executive)
SIOCADDRT	struct ortentry *	Add route to routing table
SIOCDELRT	struct ortentry *	Delete route from routing table
SIOCSIFADDR	struct ifreq *	Sets ifnet address
SIOCGIFADDR	struct ifreq *	Gets ifnet address
SIOCSIFDSTADDR	struct ifreq *	Sets peer destination address
SIOCGIFDSTADDR	struct ifreq *	Gets peer destination address
SIOCSIFFLAGS	struct ifreq *	Sets ifnet flags
SIOCGIFFLAGS	struct ifreq *	Gets ifnet flags

Table 12-1. I/O Control Commands for Socket File Descriptors

Command	Data	Description
SIOCSIFBRDADDR	struct ifreq *	Sets broadcast address
SIOCGIFBRDADDR	struct ifreq *	Gets broadcast address
SIOCGIFCONF	struct ifconf *	Gets ifnet list
SIOCSIFNETMASK	struct ifreq *	Sets net address mask
SIOCGIFNETMASK	struct ifreq *	Gets net address mask
SIOCSIFMETRIC	struct ifreq *	Sets interface metric count
SIOCGIFMETRIC	struct ifreq *	Gets interface metric count
SIOCDIFADDR	struct ifreq *	Delete interface address
SIOCAIFADDR	struct ifaliasreq *	Add or change interface address
SIOCSARP	struct arpreq *	Sets arp entry
SIOCGARP	struct arpreq *	Gets arp entry
SIOCDARP	struct arpreq *	Deletes arp entry

For **FIONBIO** and **FIOASYNC** calls, a value of 1 sets conditions and a value of 0 clears conditions.

Note: Any program that uses the TCP/IP library should not be compiled using the **KERNEL** preprocessor directive. This directive defines TCP/IP global variables that should not be visible to application code.

Corequisite Libraries

The TCP/IP library requires the following libraries in the OS Open system:

- cLib.a
- devLib.a
- rtxLib.a

Introducing the Network Library

The network library provides general networking utilities for TCP/IP. The network library provides functions for displaying and configuring network interfaces and protocol stacks, and UNIX 4.3.2 BSD functions for manipulating Internet addresses.

The network library contains network interface functions for connecting Ethernet and SLIP to the TCP/IP protocol stacks. The network interface functions require Ethernet or serial port device drivers to be configured in the OS Open system. Refer to the OS Open documentation specific to individual hardware platforms for more information about device driver availability.

The network library must be initialized by calling **net_init()** before functions in the library can be used. The TCP/IP library must be initialized by calling **tcpip_init()** prior to initializing the network library.

Users must define the **_Tcpip_services**, **_Tcpip_resolv**, **_Tcpip_protocols**, **_Tcpip_hosts**, and **_Tcpip_networks** symbols as arrays of strings containing appropriate network database entries. The arrays of strings must terminate with a NULL string.

Even if no entries are required for a particular array, each array should be defined and should contain a terminating NULL string (see “Defining the **_Tcpip_networks** Array” on page 12-6). A pound sign (#) starts a comment in an array; any subsequent characters are ignored.

The network library supports name server services. To configure a name server for a given host, the **_Tcip_resolv** array must contain a valid name server configuration. A name server configuration consists of a domain name and a name server address.

If the name server is configured for the host system and the name server machine is temporarily unavailable, the system retries the request for 75 seconds. If all attempts to contact the name server during this time are unsuccessful, name resolution is attempted using the **_Tcpip_hosts** array.

Note: These arrays are equivalent to `/etc/services`, `/etc/resolv.conf`, `/etc/protocols`, `/etc/hosts`, and `/etc/networks` files on a typical UNIX BSD system.

Defining the **_Tcpip_services** Array

The **_Tcpip_services** array defines port numbers for well-known services such as telnet, ftp, mailserver, and tftp.

The following is an example of a **_Tcpip_services** array:

```
char *_Tcip_services[] = {  
    "echo    7/tcp",  
    "echo    7/udp",  
    "discard 9/tcp",  
    "tftp     69/udp",  
    ""};
```

Defining the **_Tcpi****p_resolv** Array

The **_Tcpi****p_resolv** array defines the domain name and name server addresses.

The following is an example of a **_Tcpi****p_resolv** array:

```
char *_Tcpi_p_resolv[] = {  
    "domain raleigh.ibm.com",  
    "nameserver 9.67.160.1",  
    ""};
```

In the previous examples, the domain name is set to “raleigh.ibm.com”, and the nameserver address is set to “9.67.160.1 (domain and nameserver strings are keywords identifying the following field).

Defining the **_Tcpi****p_protocols** Array

The **_Tcpi****p_protocols** array defines the protocols supported on a given system.

The following is an example of a **_Tcpi****p_protocols** array (this example shows protocols supported in the TCP/IP library):

```
char *_Tcpi_p_protocols[] = {  
    "ip 0",  
    "icmp 1",  
    "tcp 16",  
    "udp 17",  
    ""};
```

Defining the **_Tcpi****p_hosts** Array

The **_Tcpi****p_hosts** array provides host name to host IP address resolution. This array must contain at least the name and address of the local host and the loopback address.

If the name server is in use, host name resolution is performed first using data base query to the name server. If a connection is made to the name server but it fails to resolve the host name, host name resolution is performed using the **_Tcpi****p_hosts** array.

The following is an example of a **_Tcpi****p_hosts** array:

```
char *_Tcpi_p_hosts[] = {  
    "#Address      Hostname      Alias/Comments"  
    "127.0.0.1    loopback      localhost",  
    "9.67.160.5   tanis.raleigh.ibm.com tanis #local host",  
    "9.67.164.200 kaboom",  
    ""};
```

Defining the `_Tcpip_networks` Array

The `_Tcpip_networks` array is supplied for compatibility.

The following is an example of a `_Tcpip_networks` array:

```
char *_Tcpip_networks[] = {  
    ""};
```

Corequisite Libraries

The network library requires the following libraries in the OS Open system:

- cLib.a
- devLib.a
- rtxLib.a
- tcpipLib.a

Introducing the NFS Client Library

OS Open provides a client implementation of NFS. A full description of NFS is outside the scope of this document: you should consult the appropriate documentation for your NFS server system (for example, AIX) for more details. For AIX, see "Communications Concepts and Procedures", GC23-2487.

In order to use NFS to mount a directory on another host, the host needs to have the NFS server configured and running, and it also needs to have exported the directory so that OS Open may mount it. OS Open must have TCP/IP running before its NFS client is started. Also, NFS requires the services provided by RPC, so **rpcLib.a** must also be present when the NFS client is used.

Here is an example of mounting an NFS directory.

```
#include <sys/nfsLib.h>  
...  
int rc;  
int myuserid=230;  
int mygroupid=1;  
    rc=nfs_dinit();  
if (0!=rc) {  
    printf("nfs_dinit() returned %d\n",rc);  
    return(rc);  
}  
  
rc=nfs_authset(myuserid, mygroupid, 0, NULL);  
if (0!=rc) {  
    printf("nfs_authset() returned %d\n",rc);
```

```

        return(rc);
    }
    rc=nfs_mount("gallifrey", "/u/neil", "/skaro", NULL);
    if (0!=rc) {
        printf("nfs_mount() returned %d\n",rc);
        return(rc);
    }

```

In this example, first **nfs_dinit()** is called to initialize the NFS subsystem. This only needs to be done once, no matter how many file systems are to be mounted. Next, **nfs_authset()** is called. The purpose of this call is to set the userid and groupid that the OS Open NFS client will pass to the NFS system on the server. Typically you will need to pass the userid and groupid of the owner of the directory that you are mounting, in order for the server system to allow appropriate access to the directory. Finally the file system is mounted. In this case we are mounting the directory "/u/neil" on the host "gallifrey". On OS Open this will be seen as the "/skaro" directory.

There are also a couple of utility functions provided with NFS.

nfs_authget() is the counterpart to the **nfs_authset()** function. It will return the values that were set in a previous call to **nfs_authset()**.

nfs_showmount() displays information about exported file systems on a remote host. For more details of its parameters, see the *OS Open Programmer's Reference*.

Defining the _Rpc_rpc Array

The _Rpc_rpc array provides mapping of RPC services to port numbers that are used by these services.

The following is an example of a Rpc_rpc array:

```

char *_Rpc_rpc[] = {
    "#Name      Port  Program",
    "portmapper  100000 portmap sunrpc",
    "rstatd      100001 rstat rup perfmeter",
    "rusersd     100002 rusers",
    ""},

```

Chapter 13. Writing OS Open Network Interface Drivers

A network interface driver converts IP packets into data formatted for a particular network medium. This activity may comprise segmenting an IP packet, attaching media-dependent addressing, data compression and expansion, or some other processing.

Network interface drivers are often involved in the initial stages of address resolution protocol (ARP) processing. ARP, given an IP address, uses a broadcast packet to determine the network address of a target node. LANs such as Ethernet and Token-Ring Networks use ARP for media-dependent hardware addressing to support packet routing and broadcast packets.

A network interface driver must provide logic for input and output paths. The input path takes data from the medium, reassembles the data into an IP packet, and then queues the packet onto the TCP/IP protocol stack. The output path must accept an IP packet, build the media-dependant data stream, perform IP address to media address translation if necessary, and then transmit the data on the network.

Introducing Network Interface Concepts

Because the OS Open TCP/IP networking software is derived from UNIX BSD 4.3, network interfaces written for OS Open resemble network interfaces written for UNIX BSD systems.

Working with Memory Buffers

A memory buffer (mbuf) is a specialized form of a linked list fixed buffer mechanism for processing TCP/IP packets. Each 128-byte mbuf contains control information and data. OS Open uses buffer pools to implement mbufs and provides many functions and macros that work with mbufs.

Serializing Network Interface Control Structures

net_slimp() and **net_splx()** acquire and release a serialization lock on the network interface control structures. This lock should be acquired before the state of any network interface is changed, or the IP packet input queue is manipulated.

Scheduling Network Software Interrupts

schednetisr() schedules a network software interrupt handler. In a network interface driver, this is typically done to notify the protocol stack that new packets are waiting in the input queue.

Reading an Annotated Network Interface Driver

It is difficult to describe network interface drivers using only natural language. The following annotated section of code outlines the major elements of a sample network interface driver.

Attaching the Network Interface Driver

A network interface driver generally defines an entry point that is called by an application. This function must allocate and initialize the structure that defines the interface, and call **if_attach()** to attach the new interface to the network. The structure, **struct ifnet**, is defined in the file **<net/if.h>**.

```
static struct ifnet sc_if;          /* interface structure */
/*
 * Slip_attach. Attach slip interface to asynchronous port file descriptor.
 */

int slip_attach(int fd)
{
    /* initialize interface structure */
    sc_if.if_name="sl";              /* interface name*/
    sc_if.if_unit=0;                  /* sub unit number, if needed*/
    sc_if.if_mtu=SLMTU;               /* maximum transmission unit size; protocol
                                     stack will not create a larger IP packet */
    sc_if.if_flags=IFF_POINTOPOINT; /* flags, including connection type */
    sc_if.if_type=IFT_SLIP;           /* See if_types.h for possible values */
    sc_if.if_ioctl=slip_ioctl;        /* function pointer for local ioctl */
    sc_if.if_output=slip_output;      /* function pointer for local output function */
    /* setup the output queue in the interface maximum length */

    sc_if.if_snd.ifq_maxlen=SL_MAX_QLEN;
    /* attach interface to network */
    (void)if_attach(&sc_if);
```

Sending a Packet

Assume *ifp* is a pointer to **struct ifnet**, which describes the network interface. When a packet is ready for output, the network in effect issues the call:

```
(*ifp->if_output)(ifp, m, dst);
```

ifp points to **struct ifnet**, which describes the network interface. *m* is the mbuf chain to be sent and *dst* is the destination address, expressed as a socket address structure:

```

/*
 * Slip_output. Queue a packet, then call send routine.
 */

static int slip_output(struct ifnet *ifp,
struct mbuf *m, struct sockaddr *dst)
pthread_mutex_t *s
{
/*
 * If interface is not equal to AF_INET, drop packet.
 */

if (dst->sa_family!=AF_INET) {
(void)m_freem(m);
return(EAFNOSUPPORT); }

/*
 * If interface is down, return error.
 */

if ((ifp->if_flags&(IFF_UP| IFF_RUNNING))!=(IFF_UP| IFF_RUNNING)) {
(void)m_freem(m);
return(ENETDOWN);
}

/*
 * Enqueue packet to the interface and call slip_send to send it.
 */

s=net_splimp();          /* for serialization */
if (IF_QFULL(&sc_if.if_snd)) {
IF_DROP(&sc_if.if_snd);
(void)m_freem(m);        /* drop packet */
sc_if.if_oerrors++;      /* indicate output error */
(void)net_splx(s);       /* drop serialization */
return(ENOBUFS);
}
IF_ENQUEUE(&sc_if.if_snd, m);
(void)net_splx(s);

```

An independent thread could dequeue the packet from **sc_if.if_snd** and send it. If the network's data transmission rate is sufficiently high, queue operations could be eliminated entirely and the packet could be sent synchronously.

Receiving a Packet

When a packet is received, the network interface must accept the packet and insert it into the common input queue.

OS Open network interfaces typically start a thread at attach time that synchronously waits for input from the network. The following code fragment would execute in the context of such a thread.

```
static unsigned char inbuffer[MAX_BLEN];/* input buffer */
static int b_length;          /* actual data length */

/*
 * Copy packet to mbuf chain and queue on common input queue
 */

int slip_input(){
    struct mbuf *m;
    struct mbuf *temp_mptr;
    struct mbuf *m_anchor;
    char *from_ptr= inbuffer;
    char *to_ptr;
    int to_len;

    /*
     * Allocate packet header and initialize some information
     */

    MGETHDR(m, M_DONTWAIT, MT_DATA);
    if (m==NULL) {
        sc_if.if_ierrors++;
        return (-1);
    }
    m_anchor=m;
    m->m_pkthdr.len=b_length;
    m->m_pkthdr.rcvif=sc_if;
    m->m_len=0;
    /* set pointer to point to beginning of target data */
    to_ptr=mtod(m, unsigned char *);
    to_len=MHLEN;

    /*
     * While more data is in the receive buffer
     * Effectively, this step copies the input buffer to a linked
     * chain of mbufs.
     */
}
```

```

while (from_len!=0) {
    if (from_len>to_len) {
        (void)memcpy(to_ptr, from_ptr, to_len);
        m->m_len+=to_len;
        from_len-=to_len;
        from_ptr+=to_len;
        temp_mptr=m;
    /* get another buffer */
    MGET(m, M_DONTWAIT, MT_DATA);
    if (m==NULL) {
        sc_if.if_ierrors++;
        (void)m_freem(m_anchor);
        return(-1);
    }
    to_len=MLEN;
    to_ptr=mtod(m, unsigned char *);
    m->m_len=0;
    temp_mptr->m_next=m;
} else {
    (void)memcpy(to_ptr, from_ptr, from_len);
    m->m_len+=from_len;
    to_len-=from_len;
    to_ptr+=from_len;
    from_len=0;
}
}
/* end while loop */
/* bump packet count */
sc_if.if_ipackets++;
s=net_splimp();
/* queue up packet on common queue */
if (IF_QFULL(&ipintrq)) {
    /* queue full, drop packet */
    IF_DROP(&ipintrq);
    sc->sc_if.if_ierrors++;
    sc->sc_if.if_iqdrops++;
    (void)m_freem(m);
} else {
    IF_ENQUEUE(&ipintrq, m);
    /* signal network subsystem that work is waiting */
    schednetisr(NETISR_IP);
}
(void)net_splx(s);

```

I/O Control Processing

Network interfaces respond to **ioctl()** calls primarily to mark the interface as running or bring the interface up and down. The **ioctl()** calls generally originate from **ifconfig()**.

The **ioctl()** function is called as if the following sequence were executed.

```
(*ifp->if_ioctl)(ifp, cmd, data);
```

ifp points to **struct ifnet**, which describes the network interface. *cmd* is an integer specifying the action to be taken and *data* is specific to the command. For example:

```
/*
 * Slip_ioctl.
 */

static int slip_ioctl(struct ifnet *ifp,
int cmd, caddr_t data)
{
    struct ifaddr *ifa=(struct ifaddr *)data;
    pthread_mutex_t *s=net_splimp();
    int error=0;

    switch (cmd) {
    case SIOCSIFADDR:
        /*
         * Mark interface running. Only Internet protocols are supported.
         */
        if (ifa->ifa_addr->sa_family==AF_INET) {
            if ((ifp->if_flags&IFF_UP)!=0) {
                ifp->if_flags|=IFF_RUNNING;
            }
        } else {
            error=EAFNOSUPPORT;
        }
        break;
    case SIOCSIFFLAGS:
        /*
         * If the interface is being brought down, IFF_UP is not set. If, on
         * the other hand, the interface is being brought up IFF_UP is set.
         */
```

```

if ((ifp->if_flags&IFF_UP)!=0) {
    ifp->if_flags|=IFF_RUNNING;
} else {
    ifp->if_flags&=(~IFF_RUNNING);
}
break;
default:
error=EINVAL;
}
(void)net_splx(s);
return(error);
}

```

ARP Processing

ARP processing consists of a subsystem of queues, tables, and protocol logic. Processing involves holding a packet for which the output media address is unknown, querying the network for the address, and then transmitting the held packet with the discovered address.

ARP protocols are standardized for the various types of networks. The OS Open implementation of the UNIX BSD 4.3 protocol stack supports standard Ethernet and Token Ring ARP protocols. The following code illustrates the output routine for an OS Open Ethernet device driver using the provided routines.

```

/*
ent_output
*/

static int ent_output(struct ifnet *ifp,
    struct mbuf *m, struct sockaddr *dst)
{
    struct arpcom *acp=&enet_arp_if;
    unsigned char enet_dest[6];
    unsigned short ether_type;
    struct in_addr idst;
    struct ether_header *eh;
    struct mbuf *m0;
    pthread_mutex_t *s;
    int error;

    idst.s_addr=((struct sockaddr_in *)dst)->sin_addr.s_addr;
    if ((ifp->if_flags&(IFF_UP| IFF_RUNNING))!=(IFF_UP| IFF_RUNNING)) {
        m_freem(m);
        return(ENETDOWN);
    }
}

```

```

switch(dst->sa_family) {
case AF_INET:
    if (!enet_arpresolve(acp,m,&idst,enet_dest) ) {
/*
 * Address not resolved.
 */
        return(0);
    }
    ether_type = ETHERTYPE_IP;
    break;
case AF_UNSPEC:
    eh = (struct ether_header *)dst->sa_data;
    memcpy(enet_dest, eh->ether_dhost, sizeof(enet_dest));
    ether_type = eh->ether_type;
    break;
default:
    m_freem(m);
    return(EAFNOSUPPORT);
}
M_PREPEND(m, sizeof (struct ether_header), M_DONTWAIT);
if (m == NULL) {
    error = ENOBUFS;
    return(error);
}
/* build up packet header */
eh = mtod(m, struct ether_header *);
eh->ether_type = ether_type;
memcpy(eh->ether_dhost, enet_dest, sizeof(enet_dest));
/*
 * enet_transmit() copies a chain of mbufs into a contiguous buffer
 * and transmit the resulting frame over Ethernet
 */
enet_transmit(m); /* transmit frame */
return(0);
}

```

Incoming Ethernet ARP mbuf chains (recognized by the type field) are directed to **enet_arpinput()** using a call of the form:

```
enet_arpinput(ifp, mchain);
```

ifp is a pointer to the **arpcom** structure describing the protocol and *mchain* is an mbuf chain containing the incoming ARP packet.

Chapter 14. Using File Transfer Protocols (FTP and TFTP)

The OS Open **ftp()** command implements the UNIX Berkeley Standard Distribution (BSD) v. 4.3 ftp program. **ftp()** supports file transfers between an OS Open system and a TCP/IP host. Detailed descriptions of the ftp commands are included in the *OS Open Programmer's Reference*.

The ftp program comprises two parts, a client and a server:

- The ftp client

The client is invoked using **ftp()**, either as a command from an OpenShell command line or as a function call in a program.

ftp() reads subcommands from the standard input and writes responses to the standard output. The subcommands perform or control file transfers between the two communicating systems.

The client can run only one copy of **ftp()** at a time.

- The ftpd server

ftpd_start() starts the ftpd server, also called a “daemon.”

ftpd_start() calls **pthread_create()**, which starts a thread running the **ftpd** daemon. This thread waits for connections from **ftp** clients on TCP/IP hosts and exchanges files with the clients.

Although only one instance of ftpd runs at a time, the daemon handles multiple file transfers from multiple clients simultaneously.

ftp() and **ftpd_start()** are located in the library **ftpLib.a**.

The OS Open **tftp()** command implements the IP Trivial File Transfer Protocol, which supports a simple mechanism for transferring files between a local host and a remote host.

The function can be used interactively from OpenShell or specified as a single command line. The command line form of **tftp()** can also be used directly in application code. Detailed descriptions of the **tftp()** commands and syntax are included in the *OS Open Programmer's Reference*.

Chapter 15. Using Telnet

The OS Open **tn()** command implements the UNIX Berkeley Standard Distribution (BSD) v. 4.3 telnet client program. telnet implements the TELNET Protocol, which supports logins on remote TCP/IP hosts. Detailed descriptions of the telnet commands are included in the *OS Open Programmer's Reference*.

The telnet program comprises two parts, a client and a server.

- The telnet client

The client is invoked using **tn()** or **telnet()**, either as a command from an OpenShell command line or as a function call within a program.

tn() operates in input and command modes. Initially, **tn()** is in input mode. To enter command mode, press the Ctrl and J keys simultaneously. Command mode is indicated by a **tn>** or **telnet>** prompt. Subcommands to manage the remote system are entered at the prompt. Some subcommands return **tn()** to input mode upon completion. To return to input mode at any time, press the Enter key.

Only one instance of **tn()** can run at a time.

tn() is located in the library **telnet.o**.

- The telnetd server

telnetd_start() starts the telnetd server, also called a “daemon”.

Only one instance of OpenShell can run at a time. If OpenShell is running on the console and a user access the associated OS Open system using telnet, the shell running on the console terminates.

telnetd_start() is located in the library **tnetdLib.a**. The **tcpipLib.a** and **netLib.a** libraries must be initialized before **telnetd_start()** is called.

Chapter 16. OS Open with Virtual Memory

OS Open with Virtual Memory is a distinct version of the OS Open Real-time Operating System. It includes enhanced functions over the current OS Open version that take advantage of MMU features available in some PowerPC processors.

OS Open with Virtual Memory is designed such that application code written for non-MMU version of OS Open will run on OS Open with Virtual Memory with minimal source code changes. These changes are primarily some additional library initializations. Although further changes to source code will be required to take full advantage of the OS Open with Virtual Memory features, some benefits are gained with no additional changes at all.

OS Open with Virtual Memory introduces the concept of thread groups. Thread groups are roughly analogous to processes in a multiprocess operating system, in that data allocated by threads in one thread group is inaccessible to threads in another thread group, unless special provisions are made. The maximum number of thread groups may vary among implementations. (The value for a given implementation can be found in the manifest constant **PTHREAD_MAXTG_NP** in **pthread.h**). Visibility (addressability) of text and data in the system is controlled with thread group granularity.

Also, OS Open with Virtual Memory makes meaningful distinctions between the supervisor and problem status of the processor. Supervisor state is defined by MSR Problem State bit equal to 0. User state is defined by MSR problem state bit equal to one. The processor architecture restricts some instructions to only be executed when in supervisor state. Examples include instructions that access the MSR and manipulate the MMU facilities. Also, write access to text and data in the kernel thread group is restricted to software executing in supervisor state.

OS Open with Virtual Memory will provide both instruction and data address translation. Data address translation allows the operating system flexibility in allocating memory. Using instruction address translation will mean more TLB entries must be used but provides some of the following advantages:

- Precise exceptions - It is easier to determine the cause of wild branch errors.
- Dynamic loading - Object files can be loaded into specified virtual addresses.
- Remapping - Discontinuous memory can be remapped into contiguous memory.

OS Open with Virtual Memory Features

OS Open with Virtual Memory provides its users with enhanced error detection and better RAM utilization than OS Open. It has the following features:

- Read-only code
- Stack protection
- Improved memory manager
- Protected I/O areas
- Private allocated memory
- Protection options for loaded objects

Read-Only Code

OS Open treats all segments of memory as read/write storage. If an errant program writes over program instructions, this will not be detected until possibly much later when the overwritten instructions are attempted to be executed. Problem determination can be difficult for these types of errors.

OS Open with Virtual Memory will define all code segments as read-only and executable. Kernel threads, which typically run in supervisor state, can also write code segments. An erroneous write to an instruction area by an application thread will immediately be detected and an error will be generated. Loaded functions and threads have the same behavior as the entity that loads them; a thread loaded by an application thread group will have read/execute access to code in the Kernel thread group and read/write/execute access to code in the application thread group. A thread loaded in the kernel thread group has read/write/execute access to the kernel thread group code. Thread groups are explained in detail later on.

Stack Protection

Currently, when a thread's stack is not large enough, it will write into other memory areas. This will not be immediately detected but will probably cause errors some time later. It is difficult to predict the exact stack size needed as well as to determine when stack overrun occurs. Therefore, most developers will specify stack size much larger than what is actually required, potentially wasting memory.

OS Open with Virtual Memory will allocate stacks so that writes below the start of the stack will be immediately detected. Another option is to have the operating system increase the stack in increments of a page as the stack overflows. Not only will this decrease errors due to insufficient stack sizes, but it will also allow better RAM utilization because developers can be more conservative in stack size estimates. Stack size will be increased by page size increments until it reaches a

user defined maximum, preventing a “runaway” thread from consuming all of free system memory. If insufficient free memory is available to extend the stack, the thread will wait until memory is available.

Improved Memory Manager

Current OS Open applications must be careful in their designs to avoid the problem of memory fragmentation. This occurs when portions of free memory are allocated and freed randomly, resulting in portions of free memory residing in between portions of allocated memory. Customers may have to add additional RAM to their system to ensure that there will always be enough free blocks of appropriate sizes.

Each thread group will have its own heap. The size of each thread group's heap will grow as needed and be destroyed when the thread group terminates. The pages that make up the individual heaps are not required to come from contiguous physical storage, as is the case with OS Open.

Additionally, the kernel thread group's heap will be compacted to reclaim free memory when the system runs out of free pages.

Protected I/O Areas

Memory mapped I/O areas can be protected from normal reads and writes by requiring access functions such as **inbyte()** and **outbyte()** to perform I/O.

This protection can be accomplished in two ways. One way is for the thread group to call **vm_allocate()** to map the I/O address space to a virtual address that is protected from other thread groups. Other thread groups can perform I/O operations via a message based interface, if needed.

Another way to protect I/O space is to map the area as inaccessible to user-state and provide a SVC-call interface.

Private Allocated Memory

OS Open with Virtual Memory memory manager will allocate memory that is private to the thread group. Threads will not be able to read or write private memory of another thread group unless system calls are used to explicitly allow shared memory access.

Protection Options for Loaded Objects

Objects are loaded into one of two contexts, either the kernel thread group or into one of the application thread groups.

Objects loaded into the kernel thread group are treated as extensions to the base load, that is, the text and data are readable/executable by all thread groups, but only writable from threads running in supervisor state, typically those belonging to the kernel thread group.

Objects loaded by an application thread groups are completely private to that thread group, the text and data are unaddressable from any other group.

Thread Groups

The granularity of protection for OS Open with Virtual Memory is the thread group. A thread belongs to exactly one thread group. Thread groups are roughly analogous to processes in a multiprocess operating system, in that data allocated by threads in one thread group is inaccessible to threads in another thread group unless special provisions are made. Also, mutexes, condition variables, and semaphores can be specified as being global or local to the thread group. Local objects are only visible to the owning thread group and are automatically recovered when the thread group is destroyed. Thread groups also have keys, which can be used to anchor data specific to thread groups.

The Initial Thread Group

To facilitate the portability of application from OS Open to OS Open with Virtual Memory, the initial (or kernel) thread group has some special properties. Threads belonging to the kernel thread group typically execute in supervisor state and have access to all processor instructions. All of the text and data in the initial OS Open image belongs to the initial thread group and is accessible read/write/execute to the kernel thread group, read/execute only to all application threads. To take advantage of the isolation afforded by thread groups, additional thread groups can be created by the initial thread with the desired protection and access attributes and new threads created in those new groups.

Creating and Destroying Thread Groups

Thread groups are created by the application when it is desired to partition the data that is managed by one set of threads from the rest of the system environment. The **pthread_tgcreate_np()** function creates a new thread group, and returns a new thread group ID, assuming the maximum number of simultaneous thread groups has not been exceeded.

To destroy a thread group, the **pthread_tgdestroy_np()** function is used. All private memory, threads, mutexes, condition variables, timers, unnamed semaphores, and message queue descriptors belonging to the thread group are destroyed. Note that the initial thread group, identified as **PTHREAD_KERNEL_NP**, cannot be destroyed.

Thread Group Keys

Analogous to thread keys, OS Open with Virtual Memory defines thread group keys that allow group specific data to be anchored. An example is the anchor for the thread group heap accessed via **malloc()** and **free()** calls. Accessing the anchor in this way allows the heap management code to be written in a very

general way. Once a key is created, it exists for all thread groups, with individual data values for each group. When a key is created, it can optionally be associated with a destructor function that will be called when the thread group is destroyed, if the corresponding data value is non-NULL.

Creating and Destroying Thread Group Keys

A thread group key is created with **pthread_tgkeycreate_np()**. When a thread group key is created, an optional destructor function can be associated with the key. If the corresponding key value is non-NULL at thread group destruction, the associated destructor function is called with the current key value as an argument.

To destroy a key, **pthread_tgkeydelete()** is used. If thread group-specific data is non-NULL when the key is destroyed, the results are undefined.

Accessing Thread Group Key Data

Once a key has been created, threads from within the thread group can access the data. **pthread_tgsetspecific_np()** binds a “void **” value to a defined thread group key. Different thread groups have distinct values for a given key. The key value is accessed using **pthread_tggetspecific_np()**, which returns the value of the key data to the calling thread.

Thread Group ID services

Two functions exist to interrogate thread group IDs. **pthread_tggetkernel_np()** returns the thread group ID of the initial, or kernel thread group. To return the value of the thread group of the calling thread, use **pthread_tgself_np()**.

Associating threads with Thread Groups

All threads are associated with exactly one thread group. The **tg** attribute, in the thread attributes object, controls which thread group a newly created thread belongs to. **pthread_attr_settg_np()** sets the **tg** attribute, which can be retrieved from an initialized thread attribute object using **pthread_attr_gettg_np()**. Possible values for the **tg** attribute are:

- **PTHREAD_CURRENT_TG_NP**, which creates the new thread in the same thread group as the creating thread. This option creates a peer thread with the same address scope.
- **PTHREAD_NEW_TG_NP**, which creates the new thread in a new thread group. This option is used to start the initial thread of an application that requires isolation from the creating thread group.
- **PTHREAD_SET_TG_NP**, which forces the new thread to be in a specified existing thread group. This places the new thread in the same addressability as an existing group. The thread group ID is specified in the **tgid** attribute of the thread's attributes object. The **pthread_attr_settgid_np()** function sets this value; **pthread_attr_gettgid_np()** retrieves the **tgid** attribute.

Additions to mutexes and condition variables

POSIX semantics for mutexes and condition variables assume the thread addressing model, that is, that threads have global addressing and visibility to all objects. OS Open with Virtual Memory extends the semantics to allow creation of mutexes and condition variables that are local to a thread group, or global across all thread groups. Local objects are tracked as part of the thread group, and automatically destroyed and their resources recovered when the thread group is destroyed. The “pshared” attribute controls whether the object is local or global, and similar to other POSIX thread attributes, there are “set” and “get” functions to access this value in the corresponding attribute object.

Note: Local mutexes and condition variables may also be used by threads in other thread groups

For mutexes, **pthread_mutexattr_setpshared()** can set the shared attribute of a mutex in the mutex attribute object to either **PTHREAD_PROCESS_SHARED** or **PTHREAD_PROCESS_PRIVATE**. **pthread_mutexattr_getpshared()** can retrieve the value from an initialized mutex attribute object.

Similarly, for condition variables, **pthread_condattr_setpshared()** sets the shared attribute, and **pthread_condattr_getpshared()** retrieves the value from an initialized condition variable attribute object.

Note: The scope of semaphores is determined by the type. Named semaphores are global, regardless of any shared parameter. Unnamed semaphores are local with respect to the thread group.

Thread Stack control

One of the significant advantages of OS Open with Virtual Memory is its ability to detect stack overrun, a cause of hard-to-diagnose errors. OS Open with Virtual Memory can also reduce overall stack memory usage, by permitting stacks to be dynamically extendable. Typically, to avoid any potential problems with stack overflow, application designers make stacks much larger than necessary.

The thread stack in OS Open with Virtual Memory can be created fully allocated, which means that the entire stack size specified in the thread attributes object has real memory assigned to it, or as extendable, meaning that one real page is initially assigned to the stack. The stack is allowed to grow, one real page at time, to the stack size value in the thread attributes object.

pthread_attr_setallocstack_np() sets the attribute to either **PTHREAD_ALLOCSTACK_NP** (preallocate entire stack) or **PTHREAD_EXTENDSTACK_NP** (extend stack as needed, up to stack size in attribute object). To retrieve the attribute value from a thread attribute object, use **pthread_attr_getallocstack_np()**.

Virtual Memory Manager

The Virtual Memory Manager provides a programming interface allowing the application to manage virtual address spaces. A virtual address space is bound to a particular thread group. Application calls exist to:

- Create and destroy ranges of virtual address space tied to a thread group
- Add or remove real memory pages backing a virtual address
- Move data from one virtual address space to another
- Query attributes and statistics of the Virtual Memory Manager

Create and Destroy virtual address spaces

When a thread group is created, it is not associated with any virtual storage. **vm_allocate()** establishes a virtual address range associated with a particular thread group. The initial thread group reserves virtual addresses from 0 through the constant **VM_KERNEL_END**. This reservation allows threads in other thread groups to call kernel services located in the initial image. The virtual address region can be created with several attributes that define the characteristics of the virtual address space and control the default attributes of real memory subsequently added to the region. (Default page attributes are modified by subsequent calls to **vm_protect()**, if needed.) The region attributes include:

- **VM_EXTENDABLE**, which causes references to addresses in the region to automatically assign real storage to the referenced address, if needed
- **VM_COPYON_WRITE**, which causes write references to pages referenced in the VM address range to create new real pages which are a copy of the aliased data.

The default page attributes include:

- **VM_PROTECT_READ**, which marks pages as Read Only
- **VM_PROTECT_WRITE**, which marks pages as Read/Write
- **VM_PROTECT_EXECUTE**, which allows a page to contain executable instructions
- **VM_PROTECT_ALL**, which includes **VM_PROTECT_READ**, **VM_PROTECT_WRITE**, **VM_PROTECT_EXECUTE**
- **VM_ATTR_CWRITETH**, which marks a page for write-through caching (WIMG bit W = 1)
- **VM_ATTR_CACHEOFF**, which marks a page as non-cacheable (WIMG bit I = 1)

- **VM_ATTR_NOCOH**, which marks a page as not requiring memory coherence (WIMG bit M = 0)
- **VM_ATTR_GUARDED**, which marks a page as guarded (WIMG bit G = 1)

vm_deallocate() deallocates virtual memory spaces. It is not necessary to use **vm_allocate()** and **vm_deallocate()** in pairs; any part of an allocated region can be deallocated without affecting other parts of the region. Any real memory pages obtained from the operating system free pool and not in use by other memory regions are returned to the free pool when the associated region is deallocated.

Changing attributes of virtual memory regions

vm_protect() allows the redefinition of protection attributes for a virtual memory region. These attributes govern the characteristics of any real pages or mappings that exist for the region. The region attributes **VM_EXTENDABLE** and **VM_COPYON_WRITE** govern the allocated address range as a whole and cannot be changed using **vm_protect()**.

Querying a virtual memory region

vm_region() is used to query virtual memory regions. A virtual memory region is defined as a virtual memory address range with identical attributes. Adding real pages to a defined virtual memory region, or using **vm_protect()** to change attributes, has the effect of changing a uniform address space created by **vm_allocate()** into a series of distinct virtual memory regions. **vm_region()** searches for the next memory region beginning at a supplied virtual address. Returned information includes the beginning address, size, and attributes of the detected region.

Adding and removing real mappings to a virtual address space

Unless a virtual region is marked **VM_EXTENDABLE** at creation time, any reference to the newly-created region causes a page fault at the Translation Manager level, indicating there is no real storage at the referenced address. To prevent this page fault, **vm_addrealpages()** can add real address mappings to a virtual region. Three basic types of mappings can be added.

The most common type is where real pages, supplied by the operating system, are used to establish mappings for a specified virtual address range in a defined virtual address region.

Sometimes, the real addresses that is desirable to have virtual mappings for are not being managed by the operating system. This typically includes memory mapped I/O buffers or devices.

The **VM_ATTR_IOSPACE** attribute is used to indicate that **vm_addrealpages()** is to establish mappings for specified real addresses. The real storage does not come from the free page pool, nor is there be special processing against these

real addresses when the region is deallocated. **vm_addrealpages()** can also establish aliases for real pages to implement memory sharing across thread groups. The **VM_ATTR_ALIASED** attribute indicates that the real address already has a valid virtual memory mapping in the system, and that an additional mapping is requested for this virtual region for the purpose of memory sharing.

Removing real pages from a virtual mapping

vm_removeallpages() removes mappings for a virtual address region. Subsequent access to these address will cause page faults. If the real pages were acquired from the operating system, and active aliases do not exist for them, the pages will be returned to the operating system free pool.

Virtual memory statistics

The **vm_query()** function will return various statistics of interest. Applications can use this information to monitor the real storage pool utilization, number of free translation ids, and an indication of TLB miss frequencies, page faults, and protection violations.

Moving information between virtual address regions

Certain operating system functions, such as device drivers, may have to move data between virtual address spaces. OS Open with Virtual Memory provides several functions to facilitate such transfers. **vm_read()** transfers a specified number of bytes from an address in a specified thread group to a target address in the caller's virtual address region. New pages are acquired from the free pool, filled with the desired information, and placed at a specified address in the caller's region.

vm_write() transfers data from the current thread group to a target thread group. The address at the target must already exist and be writable.

vm_copy() is similar to **vm_read()** in that data is transferred from a target thread group to the current thread group. However, **vm_copy()** assumes that the target address in the caller's thread group exists and is writable.

Obtaining the real address of a specified virtual address

vm_translate(), given a thread group and a valid virtual address within that group, returns the real address, if it exists, of the mapping.

Virtual Memory Manager Fault handling

PowerPC hardware generates several types of exceptions related to virtual memory management. For processors that use hardware-supported TLB maintenance, two basic kinds of faults are generated.

A page fault indicates that no translation exists for the specified virtual address. A protection fault indicates that the translation exists, but the application does not have sufficient authority to complete the access. An example of the latter case is a write operation to an address marked Read Only by the operating system. Processors using software to maintain TLB entries do not generate page faults, but instead are notified via exceptions that the desired translation does not exist in the TLB buffer. Exception handling software then either locates the correct translation and loads it into the TLB, or reports to higher level software that the translation is unavailable and something else needs to be done.

OS Open with Virtual Memory minimizes these differences by partitioning the virtual memory support into a processor-specific Translation Manager and a common Virtual Memory Manager. The Translation Manager is aware of all of the currently valid translations and handles TLB reload for those processors requiring software support for this function. Thus, the Translation manager reflects up to the Virtual Memory Manager page faults, indicating that the Translation Manager is unaware of a translation for a given virtual address, and protection exceptions, when there is insufficient permission for the requested access.

The Virtual Memory Manager does not necessarily pass these faults back to the application. A page fault in an extendable region or on an extendable thread stack is handled in the Virtual Memory Manager by allocating real pages and resuming the application. Similarly, a protection exception to a copy-on-write region will trigger the allocation of new pages and copying of data.

For cases other than those described above, the Virtual Memory Manager will reflect the faults back to the application using the defined events

- **EVENT_PAGE_FAULT_NP** for page faults
- **EVENT_GP_FAULT_NP** for protection exceptions

These faults are intercepted using the **int_install()** function, just as any other fault would be handled. The exception handling function is passed the install argument and the register image. If the exception handler returns 0, the next thread is dispatched. This facility is useful if the exception handler decides to suspend the faulting thread via **pthread_suspend_np()**. If the exception handler returns a non 0 value, the faulting thread will be restarted at the point of interruption. The exception handler for these faults is called with a stack managed by the kernel, 4K in size.

Miscellaneous System Issues

This chapter will cover additional changes and usage information introduced by OS Open with Virtual Memory

Symbol Table Management

Symbols that are a part of the initial image (applprog) are visible to all thread groups. Symbols added by the kernel thread group are visible to all thread groups. When symbols are added to the OS Open namespace by an application thread group, the symbols are only visible to that thread group. Symbols can only be updated by the thread group which added them.

Statsym_remove()

In OS Open, **statsym_remove()** with **NULL** information will remove the entire symbol table. For OS Open with Virtual Memory, calling **statsym_remove()** from an application thread group will return -1 and no symbols will be removed.

Supervisor function privilege

OS Open with Virtual Memory has the concept of privilege protection for system functions that may affect system integrity. This notion is independent of the processor supervisor/problem state as controlled by the MSR[PR] bit. This privilege is maintained as part of the thread state. Threads that are part of the kernel thread group have a privilege status of **PTHREAD_PRIVILEGED_NP**, while threads in an application thread group have a status of **PTHREAD_APPLICATION_NP**. While in a supervisor call, the thread's privilege is temporarily set to **PTHREAD_PRIVILEGED_NP**. The original value is restored on exit from the system call. When user supervisor functions are installed, one of the options is to specify whether the function can be used by threads in the **PTHREAD_APPLICATION_NP** privilege state, restricting user functions that may effect overall system integrity from use by application threads.

The function `pthread_getpr_np()` will return the current privilege status of the current thread.

Loader Support

Loader functions have been enhanced so that executables loaded in an application thread group are accessible only to the loading thread group. Files loaded by the kernel thread group are accessible to all application thread groups with read/execute only access, including any static or global data. Threads in the kernel thread group and/or executing in supervisor state have read/write/execute access to all kernel thread group objects, including those loaded into the initial image.

First level Interrupt Handlers

When designing first level interrupt handlers (FLIHs), keep in mind that:

- All FLIHs receive control with virtual memory translation active
- For translation purposes, FLIHs run in the context of the kernel thread group

Device Drivers

Device drivers effectively execute in the application's thread group translation context, meaning that all of the private data in the application thread group is accessible to the device driver. Also, the driver member functions execute in supervisor state, which means they have read/write access to kernel thread group static data. Hence, all persistent working buffers allocated by a device driver must be in the kernel thread group, so they can be accessible to every application thread group that uses the device driver.

Critical Exception Processing

There are several important rules to follow when writing Critical Exception handlers for OS Open with Virtual Memory:

- Critical interrupt **EVENT_CRITICAL_NP** is required to execute with Virtual Memory turned off. This means that **EVENT_CRITICAL_NP** handler can not call any OS Open functions and must return to the interruption point by returning 1 from interrupt handling function. This restriction is necessitated by 403GC hardware implementation, which prevents reliable processing of a TLB miss from within a critical exception.
- The watchdog timer has the same restriction as **EVENT_CRITICAL_NP** handler for the same reasons.
- Debug exceptions are only allowed outside the OS Open kernel. Debug exceptions will execute with translation turned on.
- Machine check exceptions will execute with translation turned on. Due to imprecise nature of machine check, interrupted thread can not be restarted.

Application Thread Group Access to Kernel Thread Group

Application thread groups have read-only access to text and data contained in the kernel thread group, including everything in the initial image (applprog). One of the implications of this is that anything in the kernel thread group that is intended to be called by application threads must not have any global read/write data.

Chapter 17. Developing OS Open Applications

OS Open applications generally run in embedded systems, which are typically developed in a cross-development environment. A cross-development environment comprises host computers and target systems. The host computers provide software and project management tools for application developers, who are not restricted to the limited computing resources typically available on the target embedded system.

Developers write, compile, and debug embedded system applications on host computers. When appropriate, application programs are loaded on a target embedded system, where they are tested in the target operating environment.

Embedded system development is an iterative process; the application programs are refined on host computers and tested on the target system until the programs meet their requirements. Eventually, the application programs are shipped as part of an embedded system.

Chapter 1, “About the OS Open Operating System,” describes the general requirements for host computers and target systems. This chapter provides detailed information about the tools and techniques used to develop OS Open applications.

For most of the examples in this chapter, the host computer is assumed to be a RISC System/6000 (RS/6000) workstation, running the AIX operating system, version 3.2.5 or higher.

The C compiler for XCOFF format (eXtended Common Object File Format) is the IBM XL C Compiler/6000. The assembler, linker, and other software tools are part of the AIX operating system. For detailed information about XCOFF, refer to *AIX Files Reference*.

The C compiler for ELF format is the IBM High C/C++ compiler. For detailed information about the compiler and associated tools, consult the user documentation listed in “Related IBM Publications” on page xxii.

Note: In this chapter, objects known to the AIX operating system are highlighted using bold type, except in examples and some lists. AIX variables appear in italics.

Introducing Host Development Tools

The AIX operating systems supports many development tools from IBM and independent software vendors. Although these tools are used to develop AIX applications, almost all can be used to develop OS Open applications.

Many such application development tools are part of the AIX operating system, and are invoked as AIX commands. These commands are described in *IBM AIX Version 3.2 for RISC System/6000: Commands Reference*. Commands and

packages that supplement or replace AIX capabilities are described in publications supplied by the softwares' developers.

Creating OS Open XCOFF Object Files

OS Open source programs, written in C and assembler language, are compiled or assembled into object files on the host computers.

Compiling C Source Programs (XCOFF Object)

The AIX command **xlc** compiles C source programs for OS Open applications.

Several **xlc** options must be used to create OS Open object files:

-c	Compiles without linking.				
-Idirectory	(uppercase <i>i</i>) Searches OS Open directories, each specified by <i>directory</i> , for #include directives.				
-qarch=architecture	Specifies the architecture supported in object code generation; <i>architecture</i> can be one of: <table><tr><td>com</td><td>Generates object code containing instructions for PowerPC 601 and POWER processors.</td></tr><tr><td>ppc</td><td>Generates object code containing instructions for the PowerPC Architecture.</td></tr></table>	com	Generates object code containing instructions for PowerPC 601 and POWER processors.	ppc	Generates object code containing instructions for the PowerPC Architecture.
com	Generates object code containing instructions for PowerPC 601 and POWER processors.				
ppc	Generates object code containing instructions for the PowerPC Architecture.				

An example of an **xlc** command follows:

```
xlc -c -I/usr/osopen/403ga_evb/include -I/usr/osopen/include -qarch=ppc hello.c
```

Assembling Assembler Language Source Programs (XCOFF Object)

The AIX **m4** preprocessor, invoked using the AIX command **m4**, and the AIX hosted assembler, invoked using the AIX command **as-emb**, are used to assemble assembler language source files.

The following **m4** option must be provided:

-Dm4path=path	Specifies the OS Open <i>m4</i> directory for the AIX m4 command. Only one directory may be specified. The specified directory is platform-specific. Refer to platform-specific OS Open publications for details.
----------------------	---

An example of an **m4** command follows:

```
m4 -Dm4path=usr/osopen/250/m4 fred.s > fred.asm
```

The output of the **m4** preprocessor serves as input for the **as-emb** command.

The following **as-emb** option must be provided:

`-o object_file` Specifies the object file containing the output of the **as-emb** command.

An example of an **as-emb** command follows:

```
as-emb -o fred.o fred.asm
```

as-emb is a superset of the AIX **as** command. The command is in the **/usr/osopen/bin** directory. It recognizes mnemonics for PowerPC 600Series as well as 400Series families. **as-emb** accepts the following machine types:

COM	PowerPC, Common
PWR	Power
PWR2	Power, Power2
PPC	PowerPC
403GA	PowerPC + PPC403GA
403GB	PowerPC + PPC403GB
PPC-EMB	PowerPC + PPC403GA + PPC403GB
601	PowerPC + PPC601
603	PowerPC + PPC603
604	PowerPC + PPC604
ANY	PowerPC, ANY

For example, a valid assembler file could contain the following:

```
.machine "403GA"    # Use 403GA mnemonics
mfsrr2 7            # Move SRR2 contents to r7
```

Working with XCOFF Object Files

After object files are created using the C compiler or the assembler, the object files may be:

- Grouped with other object files in an archive
- Linked into a bootable image
- Loaded and linked dynamically by the OS Open operating system

Working with Archive Files

The AIX command **ar** creates an archive file containing a group of object files. Archive files are created for many purposes; the OS Open libraries are archive files.

The names of archive files typically end with **.a**, such as **mathLib.a**, the OS Open library of ANSI C math functions.

The **ar** command can be used to create libraries or to append object files to existing libraries.

The following command creates an archive, *library.a*, containing the object files *file1.o* and *file2.o*:

```
ar library.a file1.o file2.o
```

The **ar** option **-q** causes **ar** to append object files to an existing archive. For example, the following command appends the object files *file1.o* and *file2.o* to the OS Open library *library.a*:

```
ar -q library.a filename1.o filename2.o
```

No special **ar** options are required when creating archive files for OS Open.

Refer to *IBM AIX Files Reference* or *IBM AIX Version 3.2 for RISC System/6000: Commands Reference* for more information about archive files.

Linking XCOFF Object Files

The AIX command **ld** command links object and archive files into one executable file to be run on an OS Open operating system. The **ld** command invokes the binder and linkage editor, which creates the executable file.

The **ld** command offers many options. The options most appropriate for creating OS Open object files are discussed under this heading. *IBM AIX Version 3.2 for RISC System/6000: Commands Reference* describes the **ld** options in detail.

The OS Open operating system runs two types of executable files, created using differing **ld** options:

- Boot image object

This type of executable file, created using the **ld** option **-D**, is also called a base load. A boot image object runs when the OS Open operating system starts up.

- Dynamically loadable object

This type of executable file is loaded and linked by the OS Open operating system.

Descriptions of **ld** options follow:

- | | |
|----------------------|--|
| -D data_start | Defines the starting address of the data section. If <i>data_start</i> is -1 , the data section immediately follows the text (code) section. This option is used to create a boot image, and is not used to link dynamically loadable objects. |
| -e label | Specifies the label used as the entry point. When linking a boot image object, <i>label</i> should be __entry (two underscores precede "entry"). When linking a dynamically loaded object, <i>label</i> should specify the program entry point. |

<code>-o filename</code>	Assigns <i>filename</i> to the name of the output object file.
<code>-boptions</code>	Specifies binder options; <i>options</i> used for OS Open files follow: loadmap: <i>filename</i> Writes the results of the bind to <i>filename</i> ; useful when debugging linker errors. map: <i>filename</i> Writes a symbol map to <i>filename</i> . This may be useful for debugging the program.
<code>nogc</code>	Turns off garbage collection (the process of discarding unreferenced control sections (csects) from the linked object file). This option is used to refer to an otherwise unreferenced function or variable from OpenShell or a dynamically loaded object. Note, however, that files created using this option may grow too large.
<code>I</code>	(uppercase <i>i</i>) Specifies an import file containing external symbols whose references are not to be resolved by the binder. Used for creating dynamically loaded objects, this option should not be specified for boot image objects.
<code>-bE</code>	Specifies an export file containing external symbols that can be referenced by other objects. Such symbols are not affected by garbage collection, even when unreferenced.

The following command creates a boot image object file named **applprog**:

```
ld -bmap:applprog.map -bloadmap:applprog.loadmap -D-1 \
-e __entry -o applprog benchmk.o /usr/osopen/250/lib/*.a,o]
```

The following command creates a dynamically loaded object named **cat.ld**:

```
ld -e cat -o cat.ld -bl:applprog.import /usr/osopen/250/lib/xlcLib.a cat.o
```

When using C Set ++ to produce C++ applications, the **xlc_link** shell script accepts the same arguments as **ld** and performs the additional steps required to support static constructors/destructors and template functions.

The following command creates a boot image object file named **applprog**:

```
xlc_link -bmap:applprog.map -bloadmap:applprog.loadmap -D-1 \
-e __entry -o applprog benchmk.o /usr/osopen/250/lib/*.a,o]
```

The following command creates a dynamically loaded object named **cat.ld**:

```
xlc_link -e cat -o cat.ld -bl:applprog.import /usr/osopen/250/lib/xlcLib.a cat.o
```

Creating OS Open ELF Object Files

OS Open source programs, written in C/C++ and assembler language, are compiled or assembled into object files on the host computers.

Compiling C Source Programs (ELF Object)

The command **hcppc** compiles C/C++ source programs for OS Open applications.

Several compile options must be used to create OS Open object files:

- | | |
|-------------|--|
| -c | Compiles without linking. |
| -Idirectory | (uppercase <i>i</i>) Searches OS Open directories, each specified by <i>directory</i> , for #include directives. |
| -HB | Produces big-endian code. |

The use of several optional flags is recommended:

- | | |
|----------------------------|---|
| -Hoff=Command_line_ident | Prevents the compiler from placing extraneous information such as compiler version, date and time compiled, and so forth, in the output. This results in a smaller object file. |
| -Hon=Char_default_unsigned | Makes 'char' default type unsigned. |
| -Hansi | For C programs only. Accepts only ANSI-C conforming programs. |
| -Hcprext=cpp | For C++ programs only. |
| -Hcplus | For C++ programs only. |

For programs targeted for processors that have no hardware floating-point support, such as the IBM PPC403GA, the following options should be used:

- | | |
|---------------------|---|
| -fsoft | Uses floating-point emulation routines for floating-point operations. |
| -Hon=non_fp_varargs | Does not save floating-point argument registers. |

For dynamically loadable objects, the following flag must also be used:

- | | |
|------|-------------------------------------|
| -pic | Produces position-independent code. |
|------|-------------------------------------|

An example of an **hcppc** command follows:

```
hcppc -c -I/usr/osopen/m403_evb/include -HB -fsoft -Hon=non_fp_varargs hello.c
```

Assembling Assembler Language Source Programs (ELF Object)

The C preprocessor and the **asppc** assembler are used to assemble assembler language source files.

The **hcppc** command is used to preprocess assembler files (expand macros, include other files, and so on). Use the following options with the **hcppc** command:

-P	Preprocess only
-DMW	Define the “MW” variable to the preprocessor. Needed for dual-path code in some include files.
-Hasmcpp	Indicates that the C preprocessor is processing an assembler file
-Idirectory	(uppercase i) Searches OS Open directories, each specified by <i>directory</i> , for #include directives.

The output of the C preprocessor, file with extension **.i**, serves as input for the **asppc** command. The following **asppc** options must be provided:

-be	Produces big-endian code.
-o object_file	Specifies the object file containing the output from the asppc command.

An example of compiling an assembler file follows:

```
hcppc -Hasmcpp -P -I/usr/osopen/m403_evb/m4 fred.s
asppc -be -ofred.o fred.i
```

Working with ELF Object Files

After object files are created using the C compiler or the assembler, the object files may be:

- Grouped with other object files in an archive
- Linked into a bootable image
- Loaded and linked dynamically by the OS Open operating system

Working with Archive Files

The command **arppc** creates an archive file containing a group of object files. Archive files are created for many purposes; the OS Open libraries are archive files.

The names of archive files typically end with **.a**, such as **mathLib.a**, the OS Open library of ANSI C math functions.

The **arppc** command can be used to create libraries or to append object files to existing libraries.

The following command creates an archive, *library.a*, containing the object files *file1.o* and *file2.o*:

```
arppc library.a file1.o file2.o
```

No special **arppc** options are required when creating archive files for OS Open.

Linking ELF Object Files

The **ldppc** command links object and archive files into one executable file to be run on an OS Open operating system. The **ldppc** command invokes the binder and linkage editor, which creates the executable file.

The **ldppc** command offers many options. The options most appropriate for creating OS Open object files are discussed under this heading. *IBM High C/C++ ELF Linker User's Guide* describes the **ldppc** options in detail.

The OS Open operating system runs two types of executable files, created using differing **ldppc** options:

- Boot image object

This type of executable file is also called a base load. A boot image object runs when the OS Open operating system starts up.

- Dynamically loadable object

This type of executable file is loaded and linked by the OS Open operating system. Descriptions of **ldppc** options follow:

-Bnoalloc_rel_dyn	Suppresses the generation of unneeded output sections for dynamically loaded objects
-Bpage_size=size	Defines the outfile memory page size. A value of 0x4 is recommended. (This defines space that will separate text and data sections.)
-Bstart_addr=address	Specifies the address of the text section. A value of 0x20000 is typical.
-e entry_point_name	Defines the entry point of the executable. The OS Open entry point is called <code>__entry</code> .
-G	Produces a dynamically loadable object.
-znodefs	Allows unresolved references. Use for dynamically loadable objects only.
-o filename	Specifies the name of the output file.

To build an OS Open boot image, the entry point in **bootLib.o** must first be extracted from **bootLib.a**. The following commands create a boot image in the file *applprog*:

```
arppc -x /usr/osopen/m403_evb/lib/bootLib.a
ldppc -Bstart_addr=0x20000 -Bpage_size=0x4 -e __entry -o applprog \
thread0.o bootLib.o $(LIBS)
$(DEL) bootLib.o
```

In the above example the variable `$(LIBS)` contains the list of OS Open libraries. The variable `$(DEL)` contains the command to delete a file ('rm' on AIX, 'DEL' on PC-DOS, etc.).

When linking an object that contains C++ code, **crti.o** must be the first item passed to the linker, and **crtn.o** must be the last item. See the sample makefile for an example.

For a boot image object that includes C++ code, also link in **mwddctor.o**.

For a dynamically loadable object that includes C++ code, also link in **mwddctorl.o**.

To initialize static constructors within the object, you must make a call to **cpp_init()** from within the object. If you want to unload the object, you must invoke static destructors by calling **cpp_exit()** from within the object, immediately prior to unloading it.

The following command creates a dynamically loadable object:

```
ldppc -G -znodefs -Bpage_size=0x4 -Bnoalloc_rel_dyn cat.o -ocat.ld -e cat
```

Working with Makefiles

The command **make** allows developers to store build information in a file, called a makefile, to streamline the build process. At each build, the makefile is run; the build commands do not need to be entered at each build. If a build procedure changes, a makefile can be easily modified.

Two sample makefiles are provided to build the sample OS Open application programs. The sample makefile is used to create the **benchmk**, **mailsamp**, and **applprog** sample application programs. On AIX the makefile is named **makefile**. On PC-DOS it is called **MAKEFILE.MAK**.

The following command builds all of the sample applications when run from the sample directory:

```
make all
```

The following command builds the dynamically loadable sample program **cat.ld** when run from the sample directory:

```
make -f cat.mak
```

Working with the OS Open Sample Programs

To help developers get started writing and testing applications, the OS Open operating system is supplied with several sample programs, makefiles, build scripts, and a diskette build utility.

Introducing the Sample Programs

The **samples** directory contains several sample programs, including:

- applprog

This program uses OpenShell as a sample application. You can link in functions and use OpenShell to interactively run and debug them. See Chapter 18, “Using OpenShell,” for detailed information about using OpenShell.

- benchmk

This program runs RhealStone benchmarks derived from a series of real-time benchmarks proposed by R. P. Kar.[†]

The benchmark programs may be run from the OpenShell command line. Call the functions by entering their name, e.g., **start_1()**. The functions are named **start_1()**, **start_2()**, **start_3()**, **start_4m()**, **start_4s()**, **start_5()**, **start_6()**. You may produce a comprehensive run of all benchmarks five times each by using the function **start_bench()**.

- cppsamp

This program is the same as the applprog program mentioned earlier but it runs the C++ sample program ctest.cpp. For Elf Eval kits thread0.c can be modified for applprog to run the ctest.cpp.

- dhry

This program runs Dhrystone, a commonly available integer benchmark.

- mailsamp

This sample SMTP daemon runs under OpenShell.

[†] See “Implementing the RhealStone Benchmark”, *Dr. Dobbs's Journal*, Vol. 15 No. 4, April 1990, pp. 46–52

- `usr_samp`

The **`usr_samp.c`** program is a simple program that shows how to properly call the `get_board_cfg()` ROM Monitor user function to determine the ROM Monitor version and the amount of DRAM installed on the board. It also shows how to search the ROM's Vital Product Data (VPD) area to obtain the six byte network address assigned to the EVB's ethernet controller. Developers interested in using any of the ROM Monitor user functions should use this program as a guide.

Introducing the Source Programs

The following source files are also included with the sample programs:

- `asmsamp.s`

A simple "hello world" program written in assembler. Used by **`applprog`**.

- `basic_os.c`

Initializes the OS Open operating system before calling `main()`

- `benchmk.c`

Provides the benchmark application. Used by **`benchmk`**.

- `config.c`

Configures the OS Open kernel. Used by **`applprog`** and **`mailsamp`**.

- `dhry_1` and `dhry_2`

These are dhrystone benchmark routines.

- `io_init.c`

Initializes OS Open's I/O subsystem. Used by **`applprog`** and **`mailsamp`**.

- `mailguy.c`

Provides the mailsamp application. Used by **`mailsamp`**.

- `network.c`

Used by TCP/IP. Configure the host names and addresses for your environment. Used by **`applprog`** and **`mailsamp`**.

- `panic.c`

See the section "Introducing the Sample **`panic()`** Module". Used by **`applprog`** and **`mailsamp`**.

- `thread0.c`

Sets up OpenShell as a sample application. Can be edited to configure various features of OS Open (networking, remote debugger, etc.). Used by **applprog** and **mailsamp**.

- **cctest.cpp**
C++ sample program.
- **donfs.c**
Provides an example of an NFS startup routine.
- **utils.c**
Provides some useful utilities such as `dir()` to produce a directory listing, and `ptime()` to print out the current date and time. Used by **applprog**.
- **usr_samp.c**
An example showing how OS Open can call the ROM Monitor function `get_board_cfg()`.

Introducing the Sample Kernel Configuration Block

The file **config.c**, in the **samples** directory, provides a sample OS Open kernel configuration block. For more information about OS Open kernel configuration blocks, see Chapter 4, “Configuring the OS Open Operating System.”

Introducing the Sample `panic()` Module

A sample panic function is provided in the file **panic.c** in the **samples** directory.

When the panic function is invoked, the first one or two displayed lines indicate the cause of the failure. Common errors, such as a machine check, are decoded and displayed. Relatively uncommon errors are not decoded; their displayed text includes an error number *nn*. The file **<panic.h>** lists error numbers and their meanings. Other displayed information describes the internal state of the thread at the time of failure.

Note: A machine check is an imprecise interrupt. The actual failure usually occurs several instructions before the reported failure. If a machine check occurs in an OS Open application, check storage references using addresses that incorporate uninitialized bank registers.

Building and Running the OS Open Sample Programs

Building and running the sample programs illustrates basic OS Open programming and operation.

Compiling the Sample Programs

The makefile in the **samples** directory contains the directives needed to build the sample applications.

To build all of the sample programs, enter the following command while working in the **samples** directory:

```
make all
```

Notes:

1. A warning in the link step that “Entry point ‘__entry’ is not a descriptor” does not indicate a problem. However, any other errors or warnings may indicate a problem and should be investigated.
2. The sample makefiles assume that OS Open has been installed in the default directory, either **/usr/osopen** or **C:\OSOPEN**. If you have installed it in a different place, update the appropriate variables at the top of the makefiles.

Loading the OS Open Image

There are several possible methods of loading the images that are produced by the makefile. Which one you use depends on the hardware that is supported by your target platform and the system software on your host system. For those platforms that support diskette drives, follow the steps in the section “Building the Diskettes”. For those platforms that do not have diskette drives, the images are booted over a TCP/IP connection. For those hosts where TCP/IP does not support the **bootp** protocol, a remote debugger such as RISCWatch may load the bootable image.

Using RISCWatch to load an image on a target board is described in the RISCWatch user documentation for the individual PowerPC processors and controllers, as listed in “Related IBM Publications” on page xxii.

Booting the Sample Images over a TCP/IP Connection.

The TCP/IP connection between the target platform and the host platform where you have built the sample programs may be ethernet, token ring, or SLIP running over a serial cable.

You need to configure the ROM monitor on your target platform to boot over the network connection. Information on how to configure the ROM monitor is provided in the manual for your target platform (for example, the *PowerPC 403GA Evaluation Board Kit User's Manual*). This also provides information on how to configure the host environment, such as editing the `/etc/bootptab` file.

The **samples** makefile produces bootable files with names in the format `<name>.img`, such as **applprog.img**. You should ensure that the boot file name

is specified in the `/etc/bootptab` file. For example, if the `bootptab` file specifies a file name of `"/usr/osopen/403ga_evb/boot.img"`, then copy your **applprog.img** file to `"/usr/osopen/403ga_evb/boot.img"`.

Building the Diskettes

This section applies only to those target platforms that have an attached diskette drive.

1. Place a formatted diskette into the drive on your RISC System/6000, PC or Sun workstation. The precise type of formatting is not important; the requirement is that the diskette be formatted into 512-byte sectors and have no bad sectors.

Note: The build process will make this diskette IPL'able and destroy any information previously contained on the diskette.

2. Execute `blddskt.sh` on the RS/6000 and Sun, or `blddskt.bat` on the PC, specifying the application you wish to use. For example, to build a Rheelstone benchmark diskette, execute:

- `blddskt.sh benchmk` [on RS/6000 or Sun], or
- `blddskt.bat benchmk` [on PC]

Output from the **dd** command in the script indicates how many 4096-byte records were placed on the diskette.

3. Once the diskette is complete, place it in the target platform's diskette drive and reboot.

Building a Loadable Module

The OS Open loader can dynamically load object modules from an application or from OpenShell. The source file **cat.c** and its makefile, **cat.mak**, provide an example of such a module.

Note that the sample **cat.c** program is not compatible with the **run()** command. This is because the sample expects a *char ** argument pointing to filename, whereas **run()** loads programs that take *argc, argv* style arguments. A **cat.ld** that is compatible with **run()** is supplied in the **ld** directory.

The sample **cat** program, which is similar to the AIX `cat` command, displays the contents of a file.

To generate the loadable module **cat.ld**, enter the following command:

```
make -f cat.mak
```

To access the `cat.ld` file from OS Open, you may use any of the following methods:

- NFS

Use NFS to mount the host directory on the OS Open system. See “Introducing the NFS Client Library” on page 12-6.

- RAMdisk

For those platforms that support a RAMdisk:

1. Build a RAMdisk containing the `cat.ld` file. See “Using the OS Open RAM Disk” on page 10-5.

or:

2. You may use ftp to transfer the **cat.ld** file from the host to an OS Open system that has a RAMdisk already installed.

- Diskette

For those targets that have a diskette drive attached, write the file to a DOS-formatted diskette on the host:

```
doswrite cat.ld cat.ld
```

Transfer the diskette to the target system.

Chapter 18. Using OpenShell

OpenShell provides an interactive development tool for OS Open applications, featuring:

- An interactive C interpreter
- Interactive invocation of compiled C functions
- A symbolic disassembler
- A local symbolic debugger
- Instruction stepping
- Instruction tracing
- Stack trace back
- Formatted display of kernel data

Developers can use OpenShell to:

- Invoke compiled C functions from a command line
- Set and delete breakpoints
- Display and alter memory and registers
- Access external symbols

Developers typically use OpenShell to run, test, and debug application functions during development. Developers can code an application incrementally, without having to wait until all code is written to begin to verify correct operation.

This chapter describes how to start and run OpenShell, and provides a quick reference for the OpenShell interactive C interpreter. For detailed information about debugging OS Open applications with the RISCWatch Debugger, consult the RISCWatch user documentation for the individual PowerPC processors and controllers, as listed in “Related IBM Publications” on page xxii.

Preparing OS Open for OpenShell

Before OpenShell can be used, it must be linked with the application code to be examined. The OpenShell thread should run at a priority higher than the highest-priority threads that OpenShell will work with. This helps to ensure that OpenShell will not be locked out by other threads.

The file system must be initialized before OpenShell starts. Call **fs_init()** and open file descriptors 0, 1, and 2 (corresponding to **stdin**, **stdout**, and **stderr**, respectively).

Starting OpenShell

OpenShell is started in a thread. The following code fragment starts OpenShell as a thread, running at the default priority:

```
#include <shell.h>
#include <pthread.h>
pthread_t thread;
pthread_attr_t attr;
struct sched_param sparam;

:

pthread_attr_init(&attr);
pthread_attr_getschedparam(&attr, &sparam);
sparam.sched_priority=SHELL_PRIORITY;
pthread_attr_setstacksize(&attr, 40*1024);
pthread_attr_setschedparam(&attr, &sparam);
pthread_create(&thread, &attr, shell, NULL);
```

Entering and Working with OpenShell Commands

OpenShell interprets user commands, which can be entered at an OpenShell prompt or read from a file.

In the first case, a user types a command at a prompt and presses Enter. If the command is a valid OpenShell command, OpenShell runs the command and, when the command finishes, displays another prompt.

OpenShell supports input and output redirection, allowing commands to be read from a file and command output to be written to a file

OpenShell commands can be edited before they are entered; previously entered commands can be searched for, recalled, edited, and run again.

Editing Commands

Commands can be edited before the Enter key is pressed to run them. The command editor provides a subset of Korn shell vi-editing mode commands.

The command editor itself has two modes, input and control. Input mode is used to enter command text; each typed character is inserted into the command at the cursor. Control mode is used to position the cursor or delete command text.

The command editor remains in input mode unless the editor is put into control mode. Pressing the Esc key puts the editor into command mode. To exit the command editor, type **i** (for “insert”) or **a** (for “append”). Pressing Enter passes the command to OpenShell for execution.

Table 18-1 lists the control mode commands. Commands are single keystrokes. In the first column of the table, key names, such as Home, are used as they appear on an IBM workstation keyboard. Your keyboard may have differently labeled keys, and your emulator may not support all keys. The letter commands, however, are available in almost all emulations.

Table 18-1. Command Editor Control Mode Commands

Command	Description
a	Returns to input mode; the next characters typed is put into the command after the cursor position
h or ←	Moves the cursor left
i	Returns to input mode; the next characters typed is put into the command at the cursor position
j or ↓	Scrolls the command list forward; after displaying the most recent command, wraps to display the oldest command
k or ↑	Scrolls the command list backward; after displaying the oldest command, wraps to display the most recent command
l or →	Moves the cursor right
0 or Home	Moves the cursor to beginning of line
\$ or End	Moves the cursor to end of line
x or Delete	Deletes character at the cursor
Ctrl-h or Backspace	Destructive backspace

Using Command History

Commands entered from a terminal are saved in a circular list. Command lines that exactly match their predecessors command lines are not saved. After the command list grows to a certain size, newly saved commands overwrite the oldest commands in the list.

Commands in the list can be recalled, edited, and rerun. To recall commands, do one of the following:

- Press the up-arrow or the down-arrow key.

Pressing the up-arrow key scrolls the command list backward; pressing the down-arrow key scrolls the command list forward.

- Press Esc, followed by typing a **k** or a **j**.

Pressing Esc puts the command editor into control mode. Typing a **k** scrolls the command list backward; typing a **j** scrolls the command list forward. To exit the command editor, type **i** or **a**. Pressing Enter passes the current command to OpenShell for execution.

Because the command list is circular, command history wraps recalled commands. For example, when scrolling backward, the most recently entered command follows the oldest command.

Redirecting Input and Output

OpenShell can read input from files and write output to files, and file opens may be nested. A file remains open until end-of-file (EOF) is reached, an error occurs, or the file is closed on command. When a file is closed, input or output reverts back to the previously opened file, if one exists. If no file is open, no redirection occurs.

Input to OpenShell is redirected using a `<` operator followed by the name of an existing file, which must be enclosed in double quotes (`"`). OpenShell reads commands from the file until EOF is reached. Input file nesting is allowed; for example, *file1* can redirect input from *file2*.

For example, to read commands from the file **/fat/setup**, enter:

```
OS OPEN> < "/fat/setup"
```

Note: A semicolon (`;`) must terminate C statements read from a file.

Output from OpenShell is redirected using a `>` operator followed by a file name, which must be enclosed in double quotes (`"`). OpenShell writes its output to the named file until another `>` operator specifies a different file or `~>`, which closes the file, is entered.

For example, to write the output of an OpenShell **display** command to the file **/fat/memlist.out**, enter:

```
OS OPEN> > "/fat/memlist.out"
OS OPEN> display 0x412348, 4096
OS OPEN> ~>
```

Note: Only output from shell commands can be redirected.

Using the Special File **/more**

Output redirected to the special file **/more** is displayed in 24-line segments to conveniently display OpenShell command output that does not fit on one screen.

To redirect output to **/more**, enter:

```
OS OPEN> > "/more"
```

If output remains after a screen of output is displayed, OpenShell prompts for a key to continue. Output continues, screen by screen.

To momentarily stop the display of data, press the Esc key. Typing:

```
~>
```

at the command prompt resumes the display of data.

To close **/more**, type:

```
~>
```

at the command prompt, closing **/more** in the same way as any other redirected output file.

Output is redirected to **/more** is until it is closed.

Changing the Command Line Prompt

When OpenShell has control, it prompts the user at the console for input.

getprompt() returns the address of the current prompt string. **setprompt()** modifies the prompt string, which cannot be longer than 15 characters.

The following example shows how **setprompt()** can change the prompt from the default prompt (OS OPEN>) to a user prompt (MyPrompt->):

```
OS OPEN>setprompt("MyPrompt->")
MyPrompt->
```

In the following example, the user prompt is changed to T#; before the prompt change, the initial user prompt (MyPrompt->) is saved in a variable:

```
MyPrompt->char prompt_save[16];
MyPrompt->strcpy(prompt_save, getprompt());
MyPrompt->strcpy(getprompt(), "T# ");
T#
```

Using the OpenShell C Interpreter

OpenShell provides a C interpreter that allows developers to use C syntax while working interactively with OS Open and application functions. The interpreter supports a subset of ANSI C, and provides several elements that are not part of the standard C language. Some elements of ANSI C are not implemented because they are not appropriate in an interpreter. For example, storage class specifiers are not supported.

The interpreter reads and interprets user input as C statements. User-defined and library functions can be entered at the command prompt, or called from other functions.

Because statements are interpreted as they are read, functions and variables must be defined before they are used. In some cases, execution of the input line is deferred until the interpreter has enough information to know what to do. For example, after an **if** statement, the interpreter looks for an **else** statement before continuing.

Understanding C Interpreter Restrictions and Differences

Because the C interpreter provides a subset of the C language defined in the ANSI C standards, the following restrictions must be observed:

- Arrays can have no more than one dimension.
- Functions are not reentrant.
- Pointers to pointers to pointers, such as **char ***triple**, are not allowed.
- Variable argument lists are not supported in user-defined functions.
- Non-user-defined functions with **double** parameters cannot be called.
- Functions are assumed to return an integer.

The interpreter does not provide as much error checking as most C compilers. For example, there is very little type checking. Parameters are not checked when calling compiled functions. In user-defined functions, the number of provided parameters must match the defined number.

When a function is defined using the interpreter, the interpreter checks the syntax as statements are interpreted. Other errors, such as using an undefined name, are not detected until the function tries to run.

Note: The interpreter inserts a semicolon (;) at the end of each input line entered from a terminal. To enter a long statement without termination at the end of a line, escape the return using a backslash (\).

Constants

The interpreter supports integer and character constants in expressions:

Integer constants may be expressed in the following radices:

- Decimal, for example, 97 (0–9)
- Octal, for example, 014 (leading 0, 0–7)
- Hexadecimal, for example, 0x6 (leading 0x, 0–9, a–f, A–F)

Character constants, each comprising a single characters enclosed in single quotes, are supported in expressions.

String Literals

The interpreter supports string literals in expressions. A string literal is a sequence of characters enclosed in double quotes.

The following special characters are understood in string literals:

<code>\n</code>	New-line
<code>\"</code>	Double quote
<code>\\</code>	Backslash

Identifiers

Identifiers (also called names) must begin with an alphabetic character, which can be followed by one or more alphanumeric characters. Identifiers must be declared before they can be used in expressions.

Expression Operators

The primary expression operators supported by the C interpreter are as follows:

(expression)	Expression
primary-expression [expression]	Array reference
primary-expression (argument-expression-list _{opt})	Function call
primary-expression . identifier	Structure reference
primary-expression -> identifier	Structure reference

Unary Operators

The following C language unary operators are understood by the interpreter when used in expressions. An lvalue is an expression referring to an object. Some operators yield lvalues. For example, if *e* is an expression of pointer type, **e* is an lvalue expression referring to the object to which *e* points.

* expression	Indirection or pointer
& lvalue	Address of lvalue
- expression	Negation
! expression	Not
~ expression	Ones complement
++ lvalue	Prefix increment
--lvalue	Prefix decrement
lvalue ++	Postfix increment
lvalue --	Postfix decrement
(type-name) expression	Cast operator
sizeof expression	
sizeof (type-name)	

Binary Operators

Except for the , (comma) binary operator, the interpreter supports the ANSI C binary operators in expressions:

Multiplicative	*
	/
	%
Additive	+
	-
Shift	>>

	<<
Relational	<
	>
	<=
	>=
Equality	==
	!=
Bitwise AND	&
Bitwise exclusive OR	^
Bitwise inclusive OR	
Logical AND	&&
Logical OR	
Assignment	=
	+=
	-=
	*=
	/=
	%=
	>>=
	<<=
	&=
	^=
	=

Storage Class Specifiers

The C interpreter does not support storage class specifiers (**auto**, **static**, **extern**, **register**, and **typedef**).

Type Specifiers

The following C language type specifiers are understood by the interpreter:

char
double
float
int
long
short
unsigned
void

struct-specifier

The type-specifier **union** is not supported by the interpreter, nor are the type-qualifiers **const** and **volatile**.

Note: **unsigned** may precede **char**, **short**, **int** and **long** type specifiers.

Structure Specifier

The format of *struct-specifier* is one of the following, where *identifier* is a unique valid symbol name:

```
struct { struct-decl-list }  
struct identifier { struct-decl-list }  
struct identifier
```

Control Specifiers

The following C language control specifiers are understood by the interpreter:

```
if ( expression ) statement  
if ( expression ) statement else statement  
while ( expression ) statement  
do statement while ( expression ) ;  
for ( expression-1opt ; expression-2opt ; expression-3opt ) statement  
break ;  
continue ;  
return ;  
return expression ;  
; (null statement)  
{ statement-list } (compound statement)
```

The C interpreter does not support **switch** and **goto** statements. Labeled statements are not supported.

The compound statement or “block” is supported; several statements can be used where one is expected. The interpreter differs from ANSI C; declarations inside a block are treated the same as declarations outside the block.

User-Defined Functions

Functions can be defined to the interpreter and later called by the user. Interpreter functions must observe the following restrictions:

- The interpreter accepts only ANSI C function definitions comprising a type and a list of identifiers, separated by commas.
- The return type must be specified.
- Parameters must be specified in a single function definition line (a backslash (\) can be used to escape Return in long lines).
- Functions are not reentrant.
- Function definitions should be put in files and read using the < (input file re-direction) command.
- Function names must be valid symbol names that are not previously defined.
- The interpreter ignores C comments. Characters between “// ” and the end of the line are also ignored by the interpreter.

User-defined function keys are supported. The following OS Open functions work with function keys:

<code>tcsetattr()</code>	Associates a character string with a function key
<code>tcgetattr()</code>	Copies, to a buffer, the string assigned to a function key

For example, the following function saves the command associated with function key "F1" and reassigns the key to the string "myHelp()".

```
#include <ttyLib.h>
#include <unistd.h>

char F1_save[PFKSTR]

/*
 * Set function key F1 to myHelp()
 */
int setF1() {
    tcgetattr(STDIN_FILENO, 1, F1_save);
    tcsetattr(STDIN_FILENO, 1, "myHelp()");
}
```

Shell Commands

Shell commands provide capabilities that are not supported in C and would otherwise have to be provided in user-defined C functions:

<code>exit ;</code>	Exits OpenShell
<code>help ;</code>	Displays a help menu
<code>help keyword ;</code>	Displays a help menu for <i>keyword</i>
<code>print expression ;</code>	Evaluates and prints <i>expression</i>
<code>remove identifier ;</code>	Removes a symbol from dynamic symbol table
<code>set print-opt ;</code>	Sets print options
<code>unset print-opt ;</code>	Unsets print options
<code>whatis identifier ;</code>	Prints information about a symbol

Note: Semicolons automatically terminate commands read from a terminal.

The print options allowable on the **set** and **unset** commands are as follows:

<code>\$hexints</code>	Prints integer expressions in hexadecimal
<code>\$hexchars</code>	Prints character expressions in hexadecimal
<code>\$hexstrings</code>	Prints the address of a character string, not the string itself
<code>\$octints</code>	Prints integer expressions in octal

C Interpreter Examples

The following examples illustrate some typical programming tasks performed using the C interpreter. In the examples, “OS OPEN>” is the OpenShell prompt.

Defining and Printing an Integer Variable

The following example defines and prints an integer variable in decimal and hexadecimal formats:

```
OS OPEN>int foo
OS OPEN>foo = 0x12345
OS OPEN>print foo
74565
OS OPEN>set $hexints
OS OPEN>print foo
0x12345
OS OPEN>unset $hexints
OS OPEN>print foo
74565
```

Declaring a Character Pointer

The next example shows declaring a character pointer allocating storage for the pointer, copying a string to the storage and displaying the string and the address:

```
OS OPEN>char *string
OS OPEN>string=malloc(100)
OS OPEN>strcpy(string,"Hello, World!\n")
OS OPEN>print string
Hello, World!
OS OPEN>set $hexstrings
OS OPEN>print string
0x46ab88
OS OPEN>unset $hexstrings
OS OPEN>print string
Hello, World!
```

Defining a Character Variable

This example shows a character variable being defined, assigned and printed in two formats:

```
OS OPEN>char c
OS OPEN>c = 'a'
OS OPEN>print c
'a'
OS OPEN>set $hexchars
OS OPEN>print c
0x61
OS OPEN>unset $hexchars
OS OPEN>print c
'a'
```

Defining a Function

The following example comprises a user-defined function:

```
OS OPEN>/* Define min() function */
OS OPEN>int min(int a, int b) {
OS OPEN>if (a < b) return a
OS OPEN>else return b
OS OPEN>}
OS OPEN>print min(2,5)
2
OS OPEN>print min(99,-99)
-99
```

Note: Because OpenShell automatically inserts semicolons (;) at the ends of lines entered at the prompt, take extra care when defining functions at the prompt.

Working with Shell Symbols

OpenShell uses and can access the following symbol tables:

- Static

This symbol table is copied from the application object file at shell initialization. OpenShell reads this table, but does not write it. This table describes the application's external symbols.

- Dynamic

This symbol table is built by OpenShell, which subsequently reads and writes the dynamic symbol table.

Simply declaring a new symbol defines it. The following example:

```
OS OPEN>unsigned char c
OS OPEN>int x, y, z
OS OPEN>char *string
OS OPEN>unsigned long temp = 0xdeadbeef
OS OPEN>whatis temp
unsigned long temp
```

The C interpreter does not allow the redefinition of symbols in either symbol table. Symbols defined by OpenShell can be removed with the **remove** command and then defined again. Symbols defined in the static table cannot be removed by OpenShell.

The following example redefines the symbol *y* from **int** to **unsigned**:

```
OS OPEN>unsigned y
[ERROR] : "y" already declared.
OS OPEN>whatis y
int y;
OS OPEN>remove y
OS OPEN>unsigned y
OS OPEN>whatis y
unsigned y;
```

Examples

The following examples illustrate some tasks you can perform using OpenShell. In the examples, "OS OPEN>" is the OpenShell prompt.

Displaying Currently Defined Kernel Objects

The kernel abstract data type functions give you access to kernel data. For example, to display a list of currently defined kernel objects enter:

```
***** IBM Microelectronics *****
*           Welcome to OpenShell      20-Jul-94   *
*           Copyright IBM Corp. 1993      *
*****
*  SYSNAME: OS Open      NODE:          *
*  RELEASE: 1.00      VERSION: 1.29      *
*           SHELL VERSION: 1.29          *
*****
*                                           *
* Type "help" for information on OpenShell. *
*****

F1=help      F2=stepi      F3=nexti      F4=cont
OS OPEN>kda_dump()
KERNEL DATA AREA                               Wed Jul 20 17:20:28 1994
-----
Installed packages  Anchor/ Package name
0xbd104  dbLib.a      1.4  4/15/94
0x0      shell.o      1.51 6/1/94
0xbce48  fatLib.a     1.18 4/11/94
0xbcd70  dsktLib.a    1.5  4/11/94
0x4e9720 fsLib.a      1.21 4/13/94
0x0      ttyLib.a     1.15 4/12/94
0xbc934  asyncLib.a   1.6  7/6/94
0xbc5b8  devLib.a     1.25 5/17/94
-----
Thread    Starting function      State
0x10c868  0x95190  motor_control  BLOCKED IN WAIT
0x4fb8d0  0x94f90  select_thread  BLOCKED IN WAIT
0x4ffc68  0x94eb8  main          READY
----- Total Number of Threads: 3 -----
Condition
0x4f3040
```

Displaying Thread Information

The following example illustrates the display of information about threadID 0x3a941c:

```
OS OPEN>thread_dump(0x3a941c)
-----
Thread ID: 0x3a941c
Starting function: 0x12b0d0 (shell)
Thread state: READY  errno: (0) Operation successful(ESUCCESS)
Stacksize: 65536  Detach state: JOINABLE  Contention scope: PROCESS
Inherit sched: DEFAULT  Scheduler: FIFO  Priority: 21  Base priority: 21
Registers:
00 00000001 003a8b78 00130930 00000000 00000002 00000000 00000001 00000001
08 ffffffff 00200000 00000017 00122948 00122f70 00134084 00000000 00134074
16 00133e10 00134070 00133e10 00133bb8 00134078 0011bf28 0013407c 0011f228
24 ffff7fff 003ad5dc 0000004c 003ad5d4 003aa6d3 003ad5cc 003aa6d4 0000004d
xer: 00000006  lr: 0002cd4c  ctr: 00000012
cr: 22008022  srr0: 0002f638  srr1: 00029000
Cancel state: NONE  Interrupt state: ENABLE  Interrupt type: CONTROLLED
Floating-Point: ENABLED
Signals pending: 0x00000000  Signal mask: 0x00000422
-----
```

Disassembling Instructions

The following example illustrates use of the OpenShell debugger to disassemble 10 instructions starting at the address 0x2000:

```
OS OPEN>listi 0x2000,10
0x00002000 (__entry+0x1fdc)  4084000c  bc      0x4,0x4,0x200c
(__entry+0x1fe8)
0x00002004 (__entry+0x1fe0)  7d442814  a      r10,r4,r5
0x00002008 (__entry+0x1fe4)  7c835040  cmpl   cr1,r3,r10
0x0000200c (__entry+0x1fe8)  7c0903a6  mtspr  ctr,r0
0x00002010 (__entry+0x1fec)  38000020  lil    r0,0x20
0x00002014 (__entry+0x1ff0)  7c0103a6  mtspr  xer,r0
0x00002018 (__entry+0x1ff4)  54a006ff  rlinm. r0,r5,0x0,0x1f
0x0000201c (__entry+0x1ff8)  90610018  st     r3,0x18(r1)
0x00002020 (__entry+0x1ffc)  7c641810  sf     r3,r4,r3
0x00002024 (__entry+0x2000)  41840028  bc     0xc,0x4,0x204c
(__entry+0x2028)
```

Displaying Memory

The following example illustrates the display of 128 bytes of memory, starting at address 0:

```
OS OPEN>display 0,128
00000000: 00000000 badfca11 0070a050 004ff81c | .... .p.P .O.. |
00000010: 00089034 28002084 00000000 00000000 | ...4 (. . .... |
00000020: 00000000 48000085 84c20004 94c10004 | .... H... .... |
00000030: 7c0008ec 7c01d000 4180fff0 33390003 | |... |... A... 39.. |
00000040: 5739003a 38c00000 7cfac814 7c07d040 | W9.: 8... |... |..@ |
00000050: 41820014 335afffc 94da0004 7c1a3800 | A... 3Z.. .... |.8. |
00000060: 4180fff8 7c0004ac 4c00012c 3bc00003 | A... |... L., ;... |
00000070: 4800000d 7c631a78 480000fe 7fdea170 | H... |c.x H... ...p |
```

Testing Functions

The C interpreter enables users to define variables, perform arithmetic operations, define functions, call OS Open or application functions, and much more. Any functions available in the load map can be called from OpenShell.

The following example tests **printf()** from OpenShell:

```
OS OPEN>printf("Hello, World!\n")
Hello, World!
```

Printing Variable Information

The following example prints the value of the external integer variable **xyz**:

```
OS OPEN>print xyz
5
```

Using OpenShell Debugging Features

The **dbLib.a** library provides local debugger functions, which are usually called by OpenShell. The local debugger functions are also available to other debuggers.

OpenShell users can enter commands using the names of external functions and variables known to the application. These names are available unless symbols were omitted using the **-s** option of the AIX **ld** command or removed using the AIX **strip** command.

These names are not the debugging symbols created using the **-g** option of the AIX **cc** command. This option places information in the object file for source level debugging, which is not provided by the local debugger.

Note: The **-bnogc** (no garbage collection) linker option can be used to capture all symbol names available to OpenShell. Without garbage collection, unresolved external symbols are not removed and object files may be large. Alternatively, import/export lists may be used to retain external symbols to be resolved at run time, so that garbage collection can be done to reduce file size.

Creating a Debugging Environment

In a typical OpenShell debugging environment, the user sits in front of an RISC System/6000 (RS/6000) computer running AIX or a PC running DOS, OS/2, or Windows.

In either case, a terminal emulator is connected to the OS Open target system. The terminal emulator should be able to capture screen images in a file. The type of connection varies, depending on the communications capabilities of the target system.

OpenShell Variables

OpenShell variables, which are supplied as part of OpenShell, are set or interpreted in special ways. By convention, OpenShell variables begin with a dollar sign (\$). User defined variables should not use a dollar sign as the first character.

The following OpenShell special variables are supplied for all PowerPC implementations:

<code>\$dfthread</code>	Default thread ID. This special variable will be used by the following OpenShell commands: at , clear , nexti , stepi , and where . The default thread can be set from the shell command line, as shown in the following example:
-------------------------	--

OS Open>`$dfthread = pthread_self()`

<code>\$r0 ... \$r31</code>	General purpose registers (GPRs) of default thread.
<code>\$iar</code>	Instruction address register (IAR) of default thread
<code>\$scr</code>	Condition register (CR) of default thread
<code>\$msr</code>	Machine state register (MSR) of default thread
<code>\$link</code>	Link register (LR) of default thread
<code>\$ctr</code>	Count register (CTR) of default thread
<code>\$xer</code>	Fixed-point exception register (XER) of default thread
<code>\$srr0</code>	Save/restore register 0 (SRR0) of default thread
<code>\$srr1</code>	Save/restore register 1 (SRR1) of default thread

Some OS Open implementations may provide addition variables.

Note: Unpredictable results may occur if reading registers from a thread that is executing.

Using Local Debugger Commands in OpenShell

Local debugger commands can be used to:

- Disassemble object code
- Display memory contents
- Set breakpoints
- Clear breakpoints
- Continue execution after a breakpoint
- Perform stack trace-backs
- Step and trace instructions
- Start a thread for debugging
- Take kernel trace snapshots

Address and count arguments for the local debugger commands may be expressions.

Disassembling Object Code

The **listi** command disassembles memory containing object code and displays the result. Addresses are shown symbolically, displaying values from the static symbol table.

listi *address* ;

listi *address* , *count* ;

address, which specifies the starting address, must be an even full word address. *count*, specifies the number of instructions to disassemble; if given, *count* must be greater than 0. If not given, *count* defaults to 1.

Note: The disassembler lists an “S” (for stop) immediately following an address to indicate a breakpoint.

The following example shows the command to disassemble the program “shell” for four instructions. A breakpoint is set at address X'7B68'.

```
OS Open>listi shell, 4
0x00007b68S(.shell)      7c0802a6  mfspr  r0,lr
0x00007b6c (.shell+0x4)   9421ffc0  stu    r1,0xfffffc0(r1)
0x00007b70 (.shell+0x8)   90010048  st     r0,0x48(r1)
0x00007b74 (.shell+0xc)   8082021c  l      r4,0x21c(r2)
```

Displaying Memory Contents

The **display** command displays the contents of memory in hexadecimal and ASCII, suppressing duplicate display lines:

display *address* ;

display *address* , *count* ;

address specifies the starting address. *count*, if given, specifies the number of bytes to display. If *count* is not given, **display** displays 16 bytes of memory.

The following example shows the command to display 64 bytes of memory starting at the address in register 2 of the default thread (**\$dfththread**):

```
OS Open>display $r2, 64
000441ac: 00035c08 00035c48 000460c0 00035c58 | ..\..H..'..X |
000441bc: 00042588 00035c80 00004a10 000425e8 | ..%.. \..J..%.. |
000441cc: 00035ca0 00035cd0 00035ce8 000460c8 | ..\.. \..'.. |
000441dc: 00042648 00035d30 00007034 000025a8 | ..&H..[0..p4..%.. |
```

Setting Breakpoints

The **at** command sets a breakpoint at *address*:

at *address* ;

at all *address* ;

at *address* "*statement-list*" ;

at all *address* "*statement-list*" ;

The breakpoint is for the default thread (**\$dfththread**), unless **at all** is given. If **at all** is given, or if **\$dfththread** is 0, all threads may stop on the breakpoint.

statement-list, if given, lists statements to be run by OpenShell when a breakpoint is reached. The statement list should be enclosed in double quotation marks.

Note: Interrupt handlers and the OpenShell thread do not stop at breakpoints.

Some examples of setting breakpoints follow:

- Set a breakpoint at the first instruction of function "func1" for the default thread:

```
OS Open>at func1
```

- Set a breakpoint for any thread 12 bytes past the start of "func1." The symbol "func1" must be cast to an **int** to allow the arithmetic operation.

```
OS Open>at all (int)func1+12
```

- Set a breakpoint at function "func2" to print a message.

```
OS Open>at func2 "printf(\"func2(%d)\\n\", $r3);"
```

- Set breakpoint at function “err_print” to print a message and print a trace back for any thread.

```
OS Open>at all err_print "print \"Trace Back\"; where"
OS Open>status
[ BRK-ID ] (THREAD ID ) ADDRESS ...
[0049A074] (0x00000000) 0x2250 .err_print+0x0 "print"Trace Back";
where"
[0049A240] (0x004a4558) 0x22c8 .func2+0x0 "printf("func2(%d)\n", $r3
[0049A410] (0x00000000) 0x234c .func1+0xc ""
[0049A730] (0x004a4558) 0x2340 .func1+0x0 ""
... end of breakpoint list
```

Clearing Breakpoints

The **clear** command clears one or more breakpoints:

```
clear address ;
clear all address ;
clear all ;
```

If *address* is given, the corresponding breakpoint is cleared for the default thread (**\$dfthread**).

If **clear all** *address* is given, the corresponding breakpoint is cleared for all threads. If *address* is not given, all breakpoints are cleared.

Some examples of clearing breakpoints follow. In each example, the **status** command following the **clear** command demonstrates the effect of the **clear** command.

- Clear a breakpoint:


```
OS Open>clear func1
OS Open>status
[ BRK-ID ] (THREAD ID ) ADDRESS ...
[0049A074] (0x00000000) 0x2250 .err_print+0x0 "print"Trace Back";where"
[0049A240] (0x004a4558) 0x22c8 .func2+0x0 "printf("func2(%d)\n", $r3
[0049A410] (0x00000000) 0x234c .func1+0xc ""
... end of breakpoint list
```
- Clear a breakpoint for all threads


```
OS Open>clear all err_print
OS Open>status
[ BRK-ID ] (THREAD ID ) ADDRESS ...
[0049A240] (0x004a4558) 0x22c8 .func2+0x0 "printf("func2(%d)\n", $r3
[0049A410] (0x00000000) 0x234c .func1+0xc ""
... end of breakpoint list
```

- Clear all breakpoints
OS Open>clear all
OS Open>status
[BRK-ID] (THREAD ID) ADDRESS ...
... end of breakpoint list

Listing Breakpoint Status

The **status** command displays a list of active breakpoints:

status ;

Continuing after a Breakpoint

The **cont** command resumes the default thread and blocks OpenShell until the next breakpoint is reached:

cont ;

If the thread exits before reaching any breakpoints, a message states that the thread stopped on exit.

If no breakpoints are reached and the thread does not terminate, OpenShell does not accept any commands. To return to OpenShell, you must send a Break character to the OS Open system. **cont** then terminates and OpenShell prompts for input.

Instruction Step/Trace

The **stepi** and **nexti** commands step the default thread (**\$dfthread**) one or more instructions:

stepi ;

stepi count ;

nexti ;

nexti count ;

stepi steps through a subroutine call; **nexti** steps over a call.

count, if given, specifies the number of instructions to step. If *count* is 0, the thread is stepped indefinitely, similar to instruction trace. If *count* is not given, one instruction is stepped.

If the thread reaches a breakpoint before *count* is reached, the thread stops at the breakpoint.

Some examples of instruction step and trace follow:

- Step one instruction:
OS Open>stepi
0x00002418 (.testThread+0x4) bfa1fff4 stm r29,0xffffffff4(r1)
- Step one instruction, skipping subroutine calls:
0x00002430 (.testThread+0x1c) 48002605 bl 0x4a34 (.pthread_mutex_init)
OS Open>nexti
0x00002434 (.testThread+0x20) 4ffffb82 cror 0x1f,0x1f,0x1f
- Step 20 instructions, skipping subroutine calls:
OS Open>nexti 20
0x00002438 (.testThread+0x24) 2c030000 cmpi cr0,r3,0x0
0x0000243c (.testThread+0x28) 8002003c l r0,0x3c(r2)
•
•
0x0000249c (.testThread+0x88) 38800002 lil r4,0x2
0x000024a0 (.testThread+0x8c) 38a00004 lil r5,0x4
- Trace instructions until the function "func1" is reached:
0x000024a0 (.testThread+0x8c) 38a00004 lil r5,0x4
OS Open>at func1
OS Open>stepi 0
0x000024a4 (.testThread+0x90) 4bffe9d bl 0x2340 (.func1)
0x00002340S(.func1) 7c0802a6 mfspr r0,lr
!!! Thread 0x4a72d4 (testThread) stopped at 0x2340
0x00002340S(.func1) 7c0802a6 mfspr r0,lr

Performing Stack Trace-Backs

The **where** command formats and displays a trace of the function call sequence in reverse order:

where ;

where uses the stack pointer and link register of the default thread (**\$dfthread**) to access the stack frame and trace-back table.

where displays function names, parameters, and addresses if the information is available.

Function parameter values may not be available in optimized code generated by the XL C compiler.

Debugging a Thread

The **thread_debug()** function creates a thread having a breakpoint:

```
thread_debug( thread-id-addr, thread-attr-addr, start-addr, thread-param ) ;
```

thread_debug() sets the variable **\$dfthread** to the thread ID of the new thread and sets a breakpoint at the start of the thread.

The new thread uses the threads attributes object pointed to by *thread-attr-addr*, if *thread-attr-addr* is given. If *thread-attr-addr* is not given, default attributes are used. The thread starts at the address specified by *start-addr* and is passed *thread-param* as a parameter.

Although **thread_debug()** is similar to **pthread_create()**, the location of the starting function is passed differently. **pthread_create()** expects a pointer to a function descriptor, while **thread_debug()** expects the address of the first instruction of a thread. Because OpenShell translates compiled function names to starting addresses, **pthread_create()** cannot be called directly from OpenShell.

Note: No type checking is done on **thread_debug()** parameters.

Some examples using **thread_debug()** follow:

- Start function “testThread” as a thread having default attributes enter:

```
OS Open>int th_id, rc;  
OS Open>rc = thread_debug(&th_id, NULL, testThread, NULL);  
!!! Thread 0x4a72d4 (testThread) stopped at 0x2414  
0x00002414S(.testThread)    7c0802a6 mfspr    r0,lr  
OS Open>  
... Debug Pthread Start
```

- Change the priority of “testThread” to 3:

```
OS-Open>char attr[32];  
OS-Open>print pthread_attr_init(&attr);  
0  
OS-Open>struct sched_param { int sched_priority; } param;  
OS-Open>param.sched_priority = 3;  
OS-Open>print pthread_attr_setschedparam(&attr, &param);  
0
```

Taking Kernel Trace Snapshots

trace_snapshot() provides a copy of the kernel trace buffer defined in the kernel configuration block:

```
trace_snapshot();
```

trace_snapshot(), which can be called from a breakpoint or from application code, displays the memory where the copy of the buffer is stored. This display can be captured using a terminal emulator program.

The **ntrcrpt** command, running on an AIX system, formats the trace into useful information. The **ntrcrpt** command file is found in the directory **/usr/osopen/bin**.

Kernel trace can help determine problems caused by the interaction of multiple threads.

An example of taking and formatting a kernel trace follows. This example shows taking a trace snapshot.

1. Using a terminal emulator such as **FTTERM**, start capturing output into a file named **trace.buf**.
2. Enter the following at the OpenShell prompt:

```
OS Open>at func1() "trace_snapshot();"
OS Open>cont
!!! Thread 0x4a72d4 (testThread) stopped at 0x2340
0x00002340S(.func1)          7c0802a6 mfspr    r0,r1r
OS Open>>>>>>>>>> KERNEL TRACE <<<<<<<<<
current_location =
0x4a57f4
startaddr =
0x4a556c
004a556c: a00a004e 00000000 2d56fa00 a00a004b | ...N .... -V.. ...K |
004a557c: 004ffbf0 2d576f00 b00a0000 20000000 | .O.. -Wo. .... ...|
004a558c: 2d66ae00 a00a004e 00000000 2d66d800 | -.f.. ..N .... -f.. |
004a559c: a00a004b 004ffbf0 2d674d00 b00a0000 | ...K .O.. -gM. ....
```

3. Use the AIX command **ntcrpt** to format the kernel trace; the formatted output is stored in the file **trace.fmt**.

A portion of a **trace.fmt** file follows:

ID	ELAPSED_SEC	DELTA_MSEC	APPL	SYSCALL	KERNEL
----	-------------	------------	------	---------	--------

```
A00  0.0000000000    0.000000  Kernel Call:
                                memheap_alloc
                                current thread id: 004FFBFC
```


time stamp: 2DED0200
A00 0.982342656 982.342656 Kernel Call:
pthread_mutex_lock
current thread id: 004FFBFC
time stamp: 2CDF9400

Appendix A. OS Open Functions Listed by Library

This appendix lists the OS Open user-callable functions by library.

The listing presents the libraries described in Chapter 3, “Understanding the OS Open Architecture.”

OS Open *Programmer's Reference* contains detailed descriptions of the listed functions.

Functions in rtxLib.a

System Management

setsysconf()
setuname()
sysconf()
sysconfig()
uname()

Thread Management

pthread_attr_destroy()
pthread_attr_getdetachstate()
pthread_attr_getfp_np()
pthread_attr_getinheritsched()
pthread_attr_getschedparam()
pthread_attr_getschedpolicy()
pthread_attr_getscope()
pthread_attr_getstackaddr()
pthread_attr_getstackaddr_np()
pthread_attr_getstacksize()
pthread_attr_init()
pthread_attr_setdetachstate()
pthread_attr_setfp_np()
pthread_attr_setinheritsched()
pthread_attr_setschedparam()
pthread_attr_setschedpolicy()
pthread_attr_setscope()
pthread_attr_setstackaddr()
pthread_attr_setstacksize()
pthread_cancel()
pthread_cond_broadcast()
pthread_cond_destroy()
pthread_cond_init()
pthread_cond_signal()
pthread_cond_timedwait()
pthread_cond_wait()
pthread_condattr_destroy()

pthread_condattr_init()
pthread_create()
pthread_equal()
pthread_errno_np()
pthread_exit()
pthread_getschedparam()
pthread_getspecific()
pthread_join()
pthread_key_create()
pthread_key_delete()
pthread_kill()
pthread_mutex_destroy()
pthread_mutex_getprioceiling()
pthread_mutex_init()
pthread_mutex_lock()
pthread_mutex_setprioceiling()
pthread_mutex_timedlock()
pthread_mutex_trylock()
pthread_mutex_unlock()
pthread_mutexattr_destroy()
pthread_mutexattr_getprioceiling()
pthread_mutexattr_getprotocol()
pthread_mutexattr_init()
pthread_mutexattr_setprioceiling()
pthread_mutexattr_setprotocol()
pthread_once()
pthread_resume_np()
pthread_self()
pthread_setcancelstate()
pthread_setcanceltype()
pthread_setschedparam()
pthread_setspecific()
pthread_suspend_np()
pthread_testcancel()
sched_getmask()
sched_setmask()
sched_yield()

Heap Management

memheap_alloc()
memheap_alloc_aligned()
memheap_alloc_pages()
memheap_extend()
memheap_free()
memheap_query()
memheap_realloc()
memheap_replace()

Buffer Pool Management

mempool_alloc()
mempool_destroy()
mempool_free()
mempool_init()
mempool_query()

Signal Handling

kill()
pthread_sigmask()
sigaddset()
sigdelset()
sigemptyset()
sigfillset()
sigismember()
siglongjmp()
sigpending()
sigsetjmp()
sigwait()

Clocks and Timers

alarm()
clock_getres()
clock_gettime()
clock_settime()
nanosleep()
sleep()
timer_create()
timer_delete()
timer_getoverrun()
timer_gettime()
timer_settime()
timertick_notify()

Interrupt and Fault Handling

int_disable()
int_enable()
int_getid()

int_install()
int_query()

Message Queues

mq_close()
mq_freemsg()
mq_getattr()
mq_notify()
mq_open()
mq_receive()
mq_send()
mq_setattr()
mq_timedreceive()
mq_timedsend()
mq_tryrecv()
mq_trysend()
mq_unlink()

Semaphores

sem_close()
sem_destroy()
sem_getvalue()
sem_init()
sem_open()
sem_post()
sem_timedwait()
sem_trywait()
sem_unlink()
sem_wait()

Trace Buffer Support

trace_get()
trace_write()

Miscellaneous Services

getpid()
np_compare_swap()
package_install()
tzset()

Functions in cLib.a

abort()
abs()
asctime()
asctime_r()
atexit()
atof()
atoi()

atol()
bsearch()
calloc()
ctime()
ctime_r()
difftime()
div()
free()
getenv()
gmtime()
gmtime_r()
isalnum()
isalpha()
iscntrl()
isdigit()
isgraph()
islower()
isprint()
ispunct()
isspace()
isupper()
isxdigit()
labs()
ldiv()
localeconv()
localtime()
localtime_r()
longjmp()
malloc()
mblen()
mbstowcs()
mbtowc()
memchr()
memcmp()
memcpy()
memmove()
memset()
mktime()
qsort()
raise()
rand()
rand_r()
realloc()
setenv_np()
setjmp()
setlocale()
signal()

sprintf()
srand()
sscanf()
strcasecmp()
strcat()
strchr()
strcmp()
strcpy()
strcspn()
strerror()
strftime()
strlen()
strncasecmp()
strncat()
strncmp()
strncpy()
strpbrk()
strrchr()
strspn()
strstr()
strtod()
strtok()
strtok_r()
strtol()
strtoul()
time()
tolower()
toupper()
vsprintf()
wcstombs()
wctomb()

Functions in mathLib.a

acos()
arcsc()
asin()
atan()
atan2()
ceil()
cos()
cosh()
exp()
fabs()
floor()
fmod()
frexp()
ldexp()

log()
log10()
modf()
pow()
sin()
sinh()
sqrt()
tan()
tanh()

Functions in fsLib.a

clearerr()
fclose()
fdopen()
feof()
ferror()
fflush()
fgetc()
fgetpos()
fgets()
flockfile()
fopen()
fprintf()
fputc()
fputs()
fread()
freopen()
fs_init()
fscanf()
fseek()
fsetpos()
ftell()
ftrylockfile()
funlockfile()
fwrite()
getc_unlocked()
gets()
perror()
printf()
putc_unlocked()
puts()
rewind()
scanf()
setbuf()
setvbuf()
tmpfile()
tmpnam()

ungetc()
vfprintf()
vprintf()

Functions in rngLib.a

rngBufGet()
rngBufPut()
rngCount()
rngCreate()
rngDelete()
rngFlush()
rngIsEmpty()
rngIsFull()

Functions in queLib.a

queCreate()
queDeq()
queDeqNoCheck()
queEnq()
queEnqFront()
queInit()
queInsert()
queNewer()
queNewest()
queOlder()
queOldest()
queRemove()

Functions in devLib.a

access()
bld_path()
chdir()
chmod()
close()
closedir()
dev_io_init()
device_install()
device_uninstall()
driver_install()
errlog()
falloc()
fcntl()
ffree()
fpathconf()
fstat()
ftruncate()

getcwd()
ioctl()
link()
lseek()
mkdir()
mlock()
mlockall()
mmap()
munlock()
munlockall()
munmap()
open()
opendir()
pathconf()
read()
readdir()
readdir_r()
rename()
rewinddir()
rmdir()
select()
select_notify()
select_redrive()
shm_init()
shm_open()
shm_unlink()
stat()
strategy()
umask()
unlink()
utime()
write()

Functions in ttyLib.a

cfgetispeed()
cfgetospeed()
cfsetispeed()
cfsetospeed()
isatty()
tcdrain()
tcflow()
tcflush()
tcgetattr()
tcgetkey()
tcsendbreak()
tcsetattr()
tcsetkey()

tty_init()

Functions in netLib.a

enet_attach()
gethostbyaddr()
gethostbyname()
gethostlock()
gethostname()
gethostunlock()
getnetbyaddr()
getnetbyname()
getnetlock()
getnetunlock()
getprotobyname()
getprotobynumb()
getprotolock()
getprotounlock()
getservbyname()
getservbyport()
getservlock()
getservunlock()
ifconfig()
inet_addr()
inet_aton()
inet_lnaof()
inet_makeaddr()
inet_netof()
inet_network()
inet_ntoa()
inet_ntoa_r()
net_init()
ping()
recv()
route()
send()
slip_attach()
slip_resume()
slip_suspend()

Functions in tcpiLib.a

accept()
bind()
connect()
enet_arputinput()
enet_arpresolve()
getpeername()

getsockname()
 getsockopt()
 if_attach()
 info_rtable()
 listen()
 m_clalloc()
 m_free()
 m_freem()
 m_prepend()
 m_retry()
 m_retryhdr()
 net_setsoftnet()
 net_splmp()
 net_splx()
 recvfrom()
 recvmmsg()
 sendmsg()
 sendto()
 setsockopt()
 shutdown()
 socket()
 socketpair()
 tcpip_init()
 tok_arpinput()
 tok_arpresolve()

Functions in rpcLib.a

authnone_create()
 authunix_create()
 authunix_create_default()
 callrpc()
 clnt_broadcast()
 clnt_create()
 clnt_pcreateerror()
 clnt_perrno()
 clnt_perror()
 clnt_screateerror()
 clnt_sperrno()
 clnt_sperror()
 clntraw_create()
 clnttcp_create()
 clntudp_bufcreate()
 clntudp_create()
 get_myaddress()
 getrpcbyname()
 getrpcbynumber()
 getrpccent()

getrpcport()
 pmap_getmaps()
 pmap_getport()
 pmap_rmtcall()
 pmap_set()
 pmap_unset()
 portmap_thread()
 registerrpc()
 rpc_thread_init()
 svc_getreq()
 svc_getreqset()
 svc_register()
 svc_run()
 svc_sendreply()
 svc_unregister()
 svcerr_auth()
 svcerr_decode()
 svcerr_noproc()
 svcerr_noprog()
 svcerr_progvers()
 svcerr_systemerr()
 svcerr_weakauth()
 svcfd_create()
 svcraw_create()
 svctcp_create()
 svcudp_bufcreate()
 svcudp_create()
 svcudp_enablecache()
 xdr_accepted_reply()
 xdr_array()
 xdr_authunix_parms()
 xdr_bool()
 xdr_bytes()
 xdr_callhdr()
 xdr_callmsg()
 xdr_char()
 xdr_des_block()
 xdr_double()
 xdr_enum()
 xdr_float()
 xdr_free()
 xdr_int()
 xdr_long()
 xdr_netobj()
 xdr_opaque()
 xdr_opaque_auth()
 xdr_pmap()

xdr_pmaplist()
 xdr_pointer()
 xdr_reference()
 xdr_rejected_reply()
 xdr_replymsg()
 xdr_rmtcall_args()
 xdr_rmtcallres()
 xdr_short()
 xdr_string()
 xdr_u_char()
 xdr_u_int()
 xdr_u_long()
 xdr_u_short()
 xdr_union()
 xdr_vector()
 xdr_void()
 xdr_wrapstring()
 xdrmem_create()
 xdrrec_create()
 xdrrec_endofrecord()
 xdrrec_eof()
 xdrrec_skiprecord()
 xdrstdio_create()
 xprt_register()
 xprt_unregister()

Functions in ftpLib.a

ftp()
 ftpd_start()

Functions in dbLib.a

dbbrkpt_clr()
 dbbrkpt_cont()
 dbbrkpt_disp()
 dbbrkpt_find_inst()
 dbbrkpt_init()
 dbbrkpt_isbp()
 dbbrkpt_next_inst()
 dbbrkpt_set()
 dbbrkpt_tclr()
 dbbrkpt_tset()
 dbbrkpt_write_inst()
 dbmem_disp()
 dbmem_listi()
 dbstepi()
 dbwhere()

thread_debug()
 trace_snapshot()

Functions in kadtLib.a

bprintf()
 bprintf_set()
 cond_dump()
 cond_list()
 flih_list()
 fpreg_dump()
 heap_list()
 ispthreadid()
 issuspended()
 kda_dump()
 library_list()
 mq_dump()
 mq_list()
 mutex_dump()
 mutex_list()
 mutexattr_dump()
 pool_list()
 semaphore_dump()
 semaphore_list()
 signal_list()
 thread_attr_dum()
 thread_dump()
 thread_info_lis()
 thread_list()
 thread_name()
 timer_dump()
 timer_list()

Functions in symLib.a

dbsymname_find()
 statsym_dump()
 statsym_find_ad()
 statsym_find_ex()
 statsym_find_ld()
 statsym_find_na()
 statsym_find_to()
 statsym_init()
 statsym_remove()
 statsym_update()

Functions in ldrLib.a

ld()

ldrDup_name()
ldrEntry_get()
ldrFree()
ldrIsexec()
ldrLink()
ldrLoad()
ldrMsg()
ldrQuery()
ldrResolve()
ldrStatus()
ldrUnlink()
ldrUnload()
ldr_Idinfo()

Functions in mwdctor.o and mwdctorl.o

cpp_init()
cpp_exit

Functions in rmsLib.a

rms_delq()
rms_init()
rms_insq()
rms_oldest()
rms_start()

Functions in nfsLib.a

nfs_authget()
nfs_authset()
nfs_dinit()
nfs_mount()
nfs_showmount()
nfs_umount()

Functions in runLib.a

run()

Functions in ramdLib.a

ramdsk_init()

Functions in rsldLib.a

rsld_start()
rsld_setschedparam()

Functions in tftp.o

tftp()

Functions in tnetdLib.a

telnetd_start()
telnetd_assigntty()

Functions in alignLib.a

align_h()

Functions in cppprtLib.a

cpp_init()
cpp_exit()

Functions in csLib.a

cs_card_status()
cs_get_io_window()
cs_get_mem_window()
cs_init()
cs_int_install()
cs_int_query()
cs_pata_parm_init()

Functions in pataLib.a

pata_init()

Functions in ssLib.a

socket_services()
ss_init()

Appendix B. Sample Device Drivers

OS Open device drivers are written so that the drivers hide the details of device operations from application programs, as described in Chapter 8, "Writing OS Open Device Drivers."

In this appendix, three sample device drivers are provided: the first for a character device, the second for a block device, and the third for a SCSI device.

Sample Asynchronous Device Driver

The following file shows an implementation for a character device driver:

```
/*-----+
|
| File Name: asyncLib.cut
|
| Function: Asynchronous device driver. Character device, example of
| common functions.
|
+-----*/

#include <sys/devLib.h>
#include <sys/devDriver.h>
#include <ppcLib.h>
#include <ioLib.h>
#include <time.h>
#include <asyncLib.h>
#include <rngLib.h>
#include <flih.h>
#include <stdarg.h>
#include <errno.h>
#include <sys/select.h>
#include <fcntl.h>
#include <string.h>
#include <signal.h>
#include <sys/ioctl.h>

#define ASYNCSTACK_SIZE 1024

static int async_configure(void **dds, va_list args);
static int async_open (file_block_t *fb, int oflag, va_list args);
static int async_close (file_block_t *fb);
static int async_read (file_block_t *fb, void *buffer, size_t length);
static int async_write (file_block_t *fb, const void *buffer, size_t length);
static int async_ioctl (file_block_t *fb, int cmd, va_list args);
```

```

static int async_select (file_block_t *fb, int *flags );

static void async_flih (void *arg );
static void async_set_baud_rate(asyncdds_t *dds , unsigned long baud_rate);
static void async_set_dlab (asyncdds_t *dds );
static void async_reset_dlab (asyncdds_t *dds );
static void async_init_dds (asyncdds_t *local_dds, int port_number);
static void async_open_port_def(asyncdds_t *local_dds );
static void async_polled_write (asyncdds_t *dds , char c, int *xoff_recv,
int port);
static int dbprintf (int port, const char *format, va_list arg_list);

static char sccsid[]="@(#) asyncLib.a 1.6 7/6/94";
static asyncdds_t *async_port1=NULL;
static asyncdds_t *async_port2=NULL;
static asyncdds_t async_polled_port;
static driver_t asyncdt={async_configure, async_open, async_close,
async_read, async_write, async_ioctl,
async_select, NULL};

/*-----+
| Async_init.
+-----*/
int async_init(driver_t *dsw,
va_list vargs)
{

flih_t flih_struct;
int rc;

*dsw=asyncdt;
/*-----+
| Install first level interrupt handler.
+-----*/
flih_struct.flih_stack=(void *)malloc(ASYNCSTACK_SIZE);
if (flih_struct.flih_stack==NULL) {
return(-1);
}
flih_struct.flih_stack=(void *)((int)flih_struct.flih_stack+ ASYNCSTACK_SIZE);
flih_struct.flih_function=(void *)async_flih;
flih_struct.arg=(void *)SAND_COM1_IRQ;
rc=sand_int_install(SAND_COM1_IRQ, &flih_struct, NULL);
if (rc!=0) {
(void)free((void *)((int)flih_struct.flih_stack- ASYNCSTACK_SIZE));
return(-1);
}

```

```

}
flih_struct.arg=(void *)SAND_COM2_IRQ;
rc=sand_int_install(SAND_COM2_IRQ, &flih_struct, NULL);
if (rc!=0) {
(void)free((void *)((int)flih_struct.flih_stack- ASYNCSTACK_SIZE));
return(-1);
}
(void)package_install(sccsid, (void *)&async_port1, NULL);
return(0);
}

/*-----+
| Async_configure.
+-----*/
static int async_configure(void **dds,
va_list vargs)
{

int port_number;
int send_rng_size;
int rcv_rng_size;
int rc0, rc1;
sem_t *sem0, *sem1, *sem2, *sem3;
static char name0[2][13]={"Async_sem_01", "Async_sem_11"};
static char name1[2][13]={"Async_sem_02", "Async_sem_12"};
static char name2[2][13]={"Async_sem_03", "Async_sem_13"};
static char name3[2][13]={"Async_sem_04", "Async_sem_14"};
asynccdds_t *local_dds;

/*-----+
| Retrieve arguments from va_list.
+-----*/
port_number=va_arg(vargs, int);
send_rng_size=va_arg(vargs, int);
rcv_rng_size=va_arg(vargs, int);
/*-----+
| Check if we have valid port number, decrement port number to be used as
| a index to array of semaphore names.
+-----*/
if ((port_number>2) || (port_number<1)) {
errno=EINVAL;
return(-1);
}
port_number--;
local_dds=(asynccdds_t *)malloc(sizeof(asynccdds_t));

```

```

if (local_dds==NULL) {
return(-1);
}
/*-----+
| Initialize and create read/write rings and semaphores. Serialization
| semaphores are created with count set to 1.
+-----*/
local_dds->send_ring=rngCreate(send_rng_size);
local_dds->rcv_ring=rngCreate(rcv_rng_size);
sem0=sem_open(name0[port_number], O_CREAT, 0, 1);
sem1=sem_open(name1[port_number], O_CREAT, 0, 1);
sem2=sem_open(name2[port_number], O_CREAT, 0, 0);
sem3=sem_open(name3[port_number], O_CREAT, 0, 0);
rc0=pthread_mutex_init(&local_dds->async_drain_m, NULL);
rc1=pthread_cond_init(&local_dds->async_drain_c, NULL);
if ((sem0==(sem_t *)-1) || (sem1==(sem_t *)-1) || (sem2==(sem_t *)-1) ||
(sem3==(sem_t *)-1) || (local_dds->send_ring==0) ||
(local_dds->rcv_ring==0) || (rc0!=0) || (rc1!=0)) {
if (sem0!=(sem_t *)-1) {
(void)sem_unlink(name0[port_number]);
(void)sem_close(sem0);
}
if (sem1!=(sem_t *)-1) {
(void)sem_unlink(name1[port_number]);
(void)sem_close(sem1);
}
if (sem2!=(sem_t *)-1) {
(void)sem_unlink(name2[port_number]);
(void)sem_close(sem2);
}
if (sem3!=(sem_t *)-1) {
(void)sem_unlink(name3[port_number]);
(void)sem_close(sem3);
}
if (local_dds->send_ring!=0) {
(void)rngDelete(local_dds->send_ring);
}
if (local_dds->rcv_ring!=0) {
(void)rngDelete(local_dds->rcv_ring);
}
if (rc0==0) {
(void)pthread_mutex_destroy(&local_dds->async_drain_m);
}
if (rc1==0) {
(void)pthread_cond_destroy(&local_dds->async_drain_c);
}

```

```

}
(void)free((void *)local_dds);
return(-1);
}
local_dds->ser_send_sem=*sem0;
local_dds->ser_rcv_sem=*sem1;
local_dds->async_send_sem=*sem2;
local_dds->async_rcv_sem=*sem3;
/*-----+
| Initialize rest of the dds structure. Port_open equal to 0 indicates that
| the port is closed.
+-----*/
(void)async_init_dds(local_dds, port_number);
/*-----+
| Place address of the dds in global variable, to be found by FLIH.
+-----*/
if (port_number==0) {
    async_port1=local_dds;
} else {
    async_port2=local_dds;
}
/*-----+
| Return local_dds pointer in *dds.
+-----*/
*dds=(void *)local_dds;
return(0);
}

/*-----+
| Async_open.
+-----*/
static int async_open(file_block_t *fb,
int oflag, va_list vargs)
{

    unsigned short parity;
    unsigned short parity_type;
    unsigned short stop_bits;
    unsigned short data_length;
    unsigned long baud_rate;
    asyncdds_t *dds=(asyncdds_t *)fb->driver_specific;
    unsigned long temp;
    char uart_reg;

    /*-----+

```

```

| Retrieve arguments from va_list.
+-----*/
parity=va_arg(vargs, unsigned short);
parity_type=va_arg(vargs, unsigned short);
stop_bits=va_arg(vargs, unsigned short);
data_length=va_arg(vargs, unsigned short);
baud_rate=va_arg(vargs, unsigned long);
/*-----+
| Verify all parameters.
+-----*/
if ((parity!=asyncParityNone) && (parity!=asyncParityGen_Check)) {
    errno=EINVAL;
    return(-1);
}
if ((parity_type!=asyncParityOdd) && (parity_type!=asyncParityEven)) {
    errno=EINVAL;
    return(-1);
}
if ((stop_bits!=asyncStopBits1) && (stop_bits!=asyncStopBits15) &&
(stop_bits!=asyncStopBits2)) {
    errno=EINVAL;
    return(-1);
}
if ((data_length!=asyncDataBits5) && (data_length!=asyncDataBits6) &&
(data_length!=asyncDataBits7) && (data_length!=asyncDataBits8)) {
    errno=EINVAL;
    return(-1);
}
if ((data_length==asyncDataBits5) && (stop_bits!=asyncStopBits15)) {
    errno=EINVAL;
    return(-1);
}
if ((baud_rate<50) || (baud_rate>256000)) {
    errno=EINVAL;
    return(-1);
}
/*-----+
| Increment open count for this port.
+-----*/
do {
    temp=dds->port_open;
} while (!np_compare_swap((int *)&dds->port_open, (int)temp, (int)temp+1));
/*-----+
| Set baud rate.
+-----*/

```



```

(void)async_set_baud_rate(dds, baud_rate);
dds->baud_rate=baud_rate;
/*-----+
| Set other UART registers. Here we need to convert line control
| characteristics passed to us by the user to LCR register bit fields. Data
| length passed by the user maps into value that needs to be written in the
| line control register when 1 is subtracted.
+-----*/
outbyte(dds->fcr_reg, asyncFCRFifoEnable| asyncFCRClearRcvFifo|
asyncFCRClearXmitFifo| asyncFCRFifoTrigger1);
if (stop_bits==asyncStopBits1) {
stop_bits=asyncLCRStopBitsOne;
} else {
stop_bits=asyncLCRStopBitsTwo;
}
if (parity==asyncParityNone) {
parity=asyncLCRParityDisable;
} else {
parity=asyncLCRParityEnable;
}
if (parity_type==asyncParityOdd) {
parity_type=asyncLCROddParity;
} else {
parity_type=asyncLCREvenParity;
}
outbyte(dds->lcr_reg, data_length-1| stop_bits| parity| parity_type);
outbyte(dds->mcr_reg, asyncMCRRTS| asyncMCRDTR| asyncMCROut2|
asyncMCROut1);
outbyte(dds->ier_reg, asyncIERModem| asyncIERReceive| asyncIERLine);
/*-----+
| Enable interrupts on the interrupt controller.
+-----*/
uart_reg=inbyte(dds->msr_reg);
dds->modem_stat_reg=(unsigned long)uart_reg;
(void)sand_int_enable(dds->intr_level);
return(0);
}

/*-----+
| Async_close.
+-----*/
static int async_close(file_block_t *fb)
{

asyncdds_t *dds=(asyncdds_t *)fb->driver_specific;

```

```

unsigned long temp;

/*-----+
| Decrement open count, and if open count not equal to 0 return. Otherwise
| close the port.
+-----*/
do {
temp=dds->port_open;
} while (!np_compare_swap((int *)&dds->port_open, (int)temp, (int)temp-1));
if (dds->port_open!=0) {
return(0);
}
/*-----+
| Disable interrupts in the UART, DTR, and RTS lines. Unlock read/write
| semaphores and flush ring buffers.
+-----*/
outbyte(dds->ier_reg, asyncIERdisableAll);
outbyte(dds->mcr_reg, asyncMCRdisableAll);
if (rngIsFull(dds->send_ring)==true) {
(void)sem_post(&dds->async_send_sem);
}
if (rngIsEmpty(dds->rcv_ring)==true) {
(void)sem_post(&dds->async_rcv_sem);
}
temp=ppcAndMsr(~ppcMsrEE);
(void)rngFlush(dds->send_ring);
(void)rngFlush(dds->rcv_ring);
(void)ppcMtmsr(temp);
return(0);
}

/*-----+
| Async_read.
+-----*/
static int async_read(file_block_t *fb,
void *buffer, size_t length)
{

asynccdds_t *dds=(asynccdds_t *)fb->driver_specific;
unsigned long msr;
size_t count=0;
char *temp_char_p=(char *)buffer;
size_t rc;

rc=sem_wait(&dds->ser_rcv_sem);

```

```

if (rc!=0) {
return(-1);
}
while (count<length) {
if ((dds->read_error_type!=asyncNoError) || (dds->port_open==0)) {
count=-1;
errno=EIO;
break;
}
msr=ppcAndMsr(~ppcMsrEE);
rc=ringBufGet(dds->rcv_ring, temp_char_p, length-count);
(void)ppcMtmsr(msr);
temp_char_p=&temp_char_p[rc];
count+=rc;
if ((ringIsEmpty(dds->rcv_ring)==true) && (count!=length)) {
rc=sem_wait(&dds->async_rcv_sem);
if (rc!=0) {
count=-1;
break;
}
}
}
(void)sem_post(&dds->ser_rcv_sem);
return(count);
}

/*-----+
| Async_write.
+-----*/
static int async_write(file_block_t *fb,
void *buffer, size_t length)
{

asynccds_t *dds=(asynccds_t *)fb->driver_specific;
unsigned long msr;
size_t count=0;
char *temp_char_p=(char *)buffer;
char uart_reg;
size_t rc;

rc=sem_wait(&dds->ser_send_sem);
if (rc!=0) {
return(-1);
}
while (count<length) {

```

```

if ((dds->write_error_type!=asyncNoError) || (dds->port_open==false)) {
count=-1;
errno=EIO;
break;
}
msr=ppcAndMsr(~ppcMsrEE);
rc=rngBufPut(dds->send_ring, temp_char_p, length-count);
(void)ppcMtmsr(msr);
/*-----+
| Enable transmitter empty interrupts in the UART, if something was placed
| in the ring.
+-----*/
if (rc!=0) {
uart_reg=inbyte(dds->ier_reg);
outbyte(dds->ier_reg, uart_reg| asyncIERTransmit);
}
temp_char_p=&temp_char_p[rc];
count+=rc;
if ((rngIsFull(dds->send_ring)==true) && (count!=length)) {
rc=sem_wait(&dds->async_send_sem);
if (rc!=0) {
count=-1;
break;
}
}
}
(void)sem_post(&dds->ser_send_sem);
return(count);
}

/*-----+
| Async_ioctl.
+-----*/
static int async_ioctl(file_block_t *fb,
int cmd, va_list vargs)
{

asynccdds_t *dds=(asynccdds_t *)fb->driver_specific;
struct timespec t;
unsigned long argument;
unsigned long *argument_p;
char uart_reg;

/*-----+
| Process command.

```

```

+-----*/
switch(cmd) {
case ASYNCBAUDSET:
argument=va_arg(vargs, unsigned long);
(void)async_set_baud_rate(dds, argument);
dds->baud_rate=argument;
break;
case ASYNCBAUDGET:
argument_p=va_arg(vargs, unsigned long *);
*argument_p=dds->baud_rate;
break;
case ASYNCTRIGGET:
argument_p=va_arg(vargs, unsigned long *);
uart_reg=inbyte(dds->fcr_reg);
*argument_p=(unsigned long)((uart_reg&0xC0)>>0x6)+1;
break;
case ASYNCTRIGSET:
uart_reg=inbyte(dds->fcr_reg);
uart_reg&=0x3F;
argument=va_arg(vargs, unsigned long);
switch (argument) {
case asyncFifoTrigger1:
outbyte(dds->fcr_reg, uart_reg);
break;
case asyncFifoTrigger4:
outbyte(dds->fcr_reg, uart_reg+0x40);
break;
case asyncFifoTrigger8:
outbyte(dds->fcr_reg, uart_reg+0x80);
break;
case asyncFifoTrigger14:
outbyte(dds->fcr_reg, uart_reg+0xC0);
break;
}
break;
case ASYNCBREAKSET:
uart_reg=inbyte(dds->lcr_reg);
outbyte(dds->lcr_reg, uart_reg|asyncLCRSetBreak);
break;
case ASYNCBREAKCLR:
uart_reg=inbyte(dds->lcr_reg);
outbyte(dds->lcr_reg, uart_reg&(~asyncLCRSetBreak));
break;
case ASYNCSTICKGET:
argument_p=va_arg(vargs, unsigned long *);

```

```

uart_reg=inbyte(dds->lcr_reg);
*argument_p=(unsigned long)((uart_reg&asyncLCRStickParity)>>0x5);
break;
case ASYNCSTICKZERO:
uart_reg=inbyte(dds->lcr_reg);
outbyte(dds->lcr_reg, uart_reg&(~asyncLCRStickParity));
break;
case ASYNCSTICKONE:
uart_reg=inbyte(dds->lcr_reg);
outbyte(dds->lcr_reg, uart_reg|asyncLCRStickParity);
break;
case ASYNCRERRORGET:
argument_p=va_arg(vargs, unsigned long *);
*argument_p=dds->read_error_type;
dds->read_error_type=asyncNoError;
break;
case ASYNCWERRORGET:
argument_p=va_arg(vargs, unsigned long *);
*argument_p=dds->write_error_type;
dds->write_error_type=asyncNoError;
break;
case ASYNCERROREN:
dds->error_handling=asyncErrorEnable;
break;
case ASYNCERRORDIS:
dds->error_handling=asyncErrorDisable;
dds->read_error_type=asyncNoError;
dds->write_error_type=asyncNoError;
break;
case ASYNCERRORGET:
argument_p=va_arg(vargs, unsigned long *);
*argument_p=dds->error_handling;
break;
case ASYNCDLENGET:
uart_reg=inbyte(dds->lcr_reg);
argument_p=va_arg(vargs, unsigned long *);
*argument_p=(unsigned long)(uart_reg&0x4)+1;
break;
case ASYNCDLENSET:
argument=va_arg(vargs, unsigned long);
uart_reg=inbyte(dds->lcr_reg);
uart_reg&=0xFC;
switch (argument) {
case asyncDataBits5:
outbyte(dds->lcr_reg, uart_reg);

```

```

break;
case asyncDataBits6:
outbyte(dds->lcr_reg, uart_reg+1);
break;
case asyncDataBits7:
outbyte(dds->lcr_reg, uart_reg+2);
break;
case asyncDataBits8:
outbyte(dds->lcr_reg, uart_reg+3);
break;
}
break;
case ASYNCSTOPGET:
uart_reg=inbyte(dds->lcr_reg);
argument_p=va_arg(vargs, unsigned long *);
*argument_p=(unsigned long)((uart_reg&asyncLCRStopBitsTwo)>>0x2)+1;
break;
case ASYNCSTOPSET1:
uart_reg=inbyte(dds->lcr_reg);
outbyte(dds->lcr_reg, uart_reg&(~asyncLCRStopBitsTwo));
break;
case ASYNCSTOPSET1_5:
if ((inbyte(dds->lcr_reg)&asyncLCRWordLengthSelect)==0x0) {
uart_reg=inbyte(dds->lcr_reg);
outbyte(dds->lcr_reg, uart_reg|asyncLCRStopBitsTwo);
} else {
errno=EINVAL;
return(-1);
}
break;
case ASYNCSTOPSET2:
uart_reg=inbyte(dds->lcr_reg);
outbyte(dds->lcr_reg, uart_reg|asyncLCRStopBitsTwo);
break;
case ASYNCPARITYNONE:
uart_reg=inbyte(dds->lcr_reg);
outbyte(dds->lcr_reg, uart_reg&(~asyncLCRParityEnable));
break;
case ASYNCPARITYGEN:
uart_reg=inbyte(dds->lcr_reg);
outbyte(dds->lcr_reg, uart_reg|asyncLCRParityEnable);
break;
case ASYNCPARITYSGET:
uart_reg=inbyte(dds->lcr_reg);
argument_p=va_arg(vargs, unsigned long *);

```

```

*argument_p=(unsigned long)((uart_reg&asyncLCRParityEnable)>>0x3)+1;
break;
case ASYNCPARITYODD:
uart_reg=inbyte(dds->lcr_reg);
outbyte(dds->lcr_reg, uart_reg&(~asyncLCREvenParity));
break;
case ASYNCPARITYEVEN:
uart_reg=inbyte(dds->lcr_reg);
outbyte(dds->lcr_reg, uart_reg|asyncLCREvenParity);
break;
case ASYNCPARITYGET:
uart_reg=inbyte(dds->lcr_reg);
argument_p=va_arg(vargs, unsigned long *);
*argument_p=(unsigned long)((uart_reg&asyncLCREvenParity)>>0x4)+1;
break;
case ASYNCXONENABLE:
dds->xon_xoff_flag=asyncXon_Xoff_enable;
break;
case ASYNCXONDISABLE:
dds->xon_xoff_flag=asyncXon_Xoff_disable;
dds->xoff_sent=false;
uart_reg=inbyte(dds->ier_reg);
outbyte(dds->ier_reg, uart_reg| asyncIERTransmit);
break;
case ASYNCXONGET:
argument_p=va_arg(vargs, unsigned long *);
*argument_p=dds->xon_xoff_flag;
break;
case ASYNCMODEMSTAT:
argument_p=va_arg(vargs, unsigned long *);
*argument_p=dds->modem_stat_reg;
break;
case ASYNCFLUSHIN:
argument=ppcAndMsr(~ppcMsrEE);
(void)rngFlush(dds->rcv_ring);
(void)sem_post(&dds->async_rcv_sem);
(void)ppcMtmsr(argument);
break;
case ASYNCFLUSHOUT:
argument=ppcAndMsr(~ppcMsrEE);
(void)rngFlush(dds->send_ring);
(void)sem_post(&dds->async_send_sem);
(void)ppcMtmsr(argument);
break;
case ASYNCDRAIN:

```



```

while (rnglsEmpty(dds->send_ring)==false) {
(void)pthread_mutex_lock(&dds->async_drain_m);
(void)clock_gettime(CLOCK_REALTIME, &t);
t.tv_sec+=10;
(void)pthread_cond_timedwait(&dds->async_drain_c,
&dds->async_drain_c, &t);
(void)pthread_mutex_unlock(&dds->async_drain_m);
}
break;
case ASYNCSIGBREAK:
case ASYNCSIGBREAK:
case ASYNCSIGBREAK:
dds->async_break_type=cmd;
break;
default:
errno=EINVAL;
return(-1);
}
return(0);
}

/*-----+
| Async_select.
+-----*/
static int async_select(file_block_t *fb,
int *flags)
{

asynccdds_t *dds=(asynccdds_t *)fb->driver_specific;
unsigned long async=*flags&SEL_ASYNC;
unsigned long ret_flags;

dds->select_fb=fb;
ret_flags=~(SEL_ASYNC| SEL_READ| SEL_WRITE| SEL_EXCP);
if ((*flags&SEL_READ)!=0) {
if (rnglsEmpty(dds->rcv_ring)!=true) {
ret_flags=SEL_READ;
} else if (async==SEL_ASYNC) {
dds->select_request|=SEL_READ;
}
}
if ((*flags&SEL_WRITE)!=0) {
if (rnglsFull(dds->send_ring)!=true) {
ret_flags|=SEL_WRITE;
} else if (async==SEL_ASYNC) {

```

```

dds->select_request|=SEL_WRITE;
}
}
if ((*flags&SEL_EXCP)!=0) {
if ((dds->read_error_type!=asyncNoError) ||
(dds->write_error_type!=asyncNoError)) {
ret_flags|=SEL_EXCP;
} else if (async==SEL_ASYNC) {
dds->select_request|=SEL_EXCP;
}
}
}
*flags=(int)ret_flags;
return(0);
}

/*-----+
| Async_flih.
+-----*/
static void async_flih(void *arg)
{

char uart_reg;
char buffer[ASYNC_FIFO_SIZE];
unsigned long rc, temp=false;
asynccdds_t *dds;

if (arg==(void *)SAND_COM1_IRQ) {
dds=async_port1;
} else {
dds=async_port2;
}
while (1) {
/*-----+
| Read interrupt identification register on the UART, and process
| interrupts if necessary.
+-----*/
uart_reg=inbyte(dds->iir_reg);
uart_reg&=asynclIRMask;
if ((uart_reg&asynclIRNoInterrupt)!=0) {
break;
}
switch(uart_reg) {
/*-----+
| Receive data.
+-----*/

```

```

case asyncIIRReceive:
case asyncIIRFifoTimeout:
while (1) {
uart_reg=inbyte(dds->rxtx_reg);
if ((dds->xon_xoff_flag==asyncXon_Xoff_enable) &&
(uart_reg==asyncXOFFchar)) {
uart_reg=inbyte(dds->ier_reg);
dds->xoff_sent=true;
outbyte(dds->ier_reg, uart_reg&(~asyncIERTransmit));
} else if ((dds->xon_xoff_flag==asyncXon_Xoff_enable) &&
(uart_reg==asyncXONchar)) {
uart_reg=inbyte(dds->ier_reg);
dds->xoff_sent=false;
outbyte(dds->ier_reg, uart_reg| asyncIERTransmit);
} else if ((rngIsFull(dds->rcv_ring)==true) &&
(dds->error_handling==asyncErrorEnable)) {
dds->read_error_type=asyncOverrunError;
} else {
if (rngIsEmpty(dds->rcv_ring)==true) {
(void)sem_post(&dds->async_rcv_sem);
}
if (dds->next_break==false) {
temp=true;
(void)rngBufPut(dds->rcv_ring, &uart_reg, 1);
} else {
dds->next_break=false;
}
}
/*-----+
| If there are more characters in UART receive another one.
| LSR data ready bit indicates there is a character to receive.
+-----*/
if ((inbyte(dds->lsrc_reg)&asyncLSRDataReady)!=1) {
break;
}
}
/*-----+
| Call select_notify function if there is read notification
| pending, and byte was read, or if overrun occurred.
+-----*/
if (((dds->select_request&SEL_EXCP)!=0) &&
(dds->read_error_type==asyncOverrunError)) {
dds->select_request&=~SEL_EXCP;
(void)select_notify(dds->select_fb, SEL_EXCP);
}

```

```

if (((dds->select_request&SEL_READ)!=0) && (temp==true)) {
    dds->select_request&=~SEL_READ;
    (void)select_notify(dds->select_fb, SEL_READ);
}
break;
/*-----+
| Send data.
+-----*/
case asyncIIRTransmit:
if (dds->xoff_sent==false) {
    if (rngIsFull(dds->send_ring)==true) {
        (void)sem_post(&dds->async_send_sem);
    }
    if (dds->xon_xoff_flag==asyncXon_Xoff_enable) {
        rc=rngBufGet(dds->send_ring, buffer, 1);
    } else {
        rc=rngBufGet(dds->send_ring, buffer, ASYNC_FIFO_SIZE);
    }
    for (temp=0; temp<rc; temp++) {
        outbyte(dds->rxtx_reg, buffer[temp]);
    }
    if (rngIsEmpty(dds->send_ring)==true) {
        uart_reg=inbyte(dds->ier_reg);
        outbyte(dds->ier_reg, uart_reg&(~asyncIERTransmit));
        (void)pthread_cond_signal(&dds->async_drain_c);
    }
}
/*-----+
| Call select_notify function if there is write notification
| pending.
+-----*/
if ((dds->select_request&SEL_WRITE)!=0) {
    dds->select_request&=~SEL_WRITE;
    (void)select_notify(dds->select_fb, SEL_WRITE);
}
} else {
    uart_reg=inbyte(dds->ier_reg);
    outbyte(dds->ier_reg, uart_reg&(~asyncIERTransmit));
}
break;
/*-----+
| Modem interrupt. One of the serial lines has changed state. This
| is signaled when CTS, DSR, RI and DCD change state.
+-----*/
case asyncIIRModem:
    uart_reg=inbyte(dds->msr_reg);

```

```

if (dds->error_handling==asyncErrorEnable) {
    dds->read_error_type=asyncModemStatusError;
    dds->write_error_type=asyncModemStatusError;
    dds->modem_stat_reg=(unsigned long)uart_reg;
    if (rngIsFull(dds->send_ring)==true) {
        (void)sem_post(&dds->async_send_sem);
    }
    if (rngIsEmpty(dds->rcv_ring)==true) {
        (void)sem_post(&dds->async_rcv_sem);
    }
}
/*-----+
| Call select_notify function if there is error notification
| pending.
+-----*/
if ((dds->select_request&SEL_EXCP)!=0) {
    dds->select_request&=~SEL_EXCP;
    (void)select_notify(dds->select_fb, SEL_EXCP);
}
break;
/*-----+
| Line interrupt.
+-----*/
case asyncIIRLine:
    uart_reg=inbyte(dds->lsrc_reg);
    if ((uart_reg&asyncLSRBreakInterrupt)!=0) {
        dds->next_break=true;
        if (dds->async_break_type==ASYNCERRBREAK) {
            dds->read_error_type=asyncBREAKInterrupt;
            dds->next_break=false;
        } else if (dds->async_break_type==ASYNCISGBREAK) {
            (void)rngFlush(dds->rcv_ring);
            (void)rngFlush(dds->send_ring);
            (void)kill(getpid(), SIGINT);
        }
    } else if (dds->error_handling==asyncErrorEnable) {
        if ((uart_reg&asyncLSRFramingError)!=0) {
            dds->read_error_type=asyncFramingError;
        } else if ((uart_reg&asyncLSRParityError)!=0) {
            dds->read_error_type=asyncParityError;
        } else if ((uart_reg&asyncLSROverrunError)!=0) {
            dds->read_error_type=asyncOverrunError;
        } else if ((uart_reg&asyncLSRRxFifoError)!=0) {
            dds->read_error_type=asyncFIFOError;
        }
    }
}

```

```

if (rngIsEmpty(dds->rcv_ring)==true) {
(void)sem_post(&dds->async_rcv_sem);
}
}
/*-----+
| Call select_notify function if there is error notification
| pending.
+-----*/
if ((dds->select_request&SEL_EXCP)!=0) {
dds->select_request&=~SEL_EXCP;
(void)select_notify(dds->select_fb, SEL_EXCP);
}
break;
default:
break;
}
}
}
}

```

Sample Diskette Device Driver

The following file shows the implementation of a block device driver:

```

/*-----+
|
| File Name: dsktLib.cut
|
| Descriptive name: Diskette device driver. Block device driver, example
| of common functions.
|
+-----*/
#include <sys/types.h>
#include <sys/devLib.h>
#include <sys/devDriver.h>
#include <sys/select.h>
#include <sys/blockdev.h>
#include <sys/ioctl.h>
#include <stdlib.h>
#include <pthread.h>
#include <flih.h>
#include <time.h>
#include <fcntl.h>
#include <errno.h>

```

```

#include <unistd.h>
#include <limits.h>
#include <ioLib.h>
#include <dsktLib.h>
#include <memLib.h>
#include "dsktdrvr.h"

/* forward declarations */
static int dskt_config(void **dds, va_list vargs);
static int dskt_open(file_block_t *fb, int oflag, va_list vargs);
static int dskt_close(file_block_t *fb);
static int dskt_read(file_block_t *fb, void *buffer, size_t length);
static int dskt_write(file_block_t *fb, const void *buffer, size_t length);
static int dskt_ioctl(file_block_t *fb, int cmd, va_list vargs);
static int dskt_select(file_block_t *fb, int *events);
static int dskt_strategy(file_block_t *fb, block_req_t *blk_req);

static driver_t dsktdst = {dskt_config, dskt_open, dskt_close, dskt_read,
dskt_write, dskt_ioctl, dskt_select, dskt_strategy};

/* private types */
/* cleanup structure */
struct cleanup
{
    int cleanup_depth;
    struct cleanup_record
    {
        void (*cln_function)(void *arg1, void *arg2, void *arg3);
        void *arg1;
        void *arg2;
    } cln_array[10];
};

extern char dskt_version[];
/* subsystem wide control entry */
struct dskt_dsa dsktdsa = {dskt_version};

/* file scope statics */
static struct drive *dsktdrive = dsktdsa.dsktdrive;
static void *dskt_buffer; /* global diskette buffer */
static pthread_cond_t dskt_cond;
static pthread_mutex_t dskt_mutex; /* mutex for interrupt processing */
static pthread_mutex_t dskt_mutex;

```

```

/* FLIH stack size */
#define DSKT_INT_STACK_SIZE 1024

/* media arrays */
/* There will be an entry for every type of media supported */
static struct media_info media[3] =
{ /* 2.88 Media */
{
M2PT88MB, /* media type */
512, /* sector size */
36, /* end of track */
79, /* max cylinder */
0x38, /* gap length */
0x03, /* data rate */
0x0A, /* step rate */
1, /* head unload */
1 /* head load */
},
/* 1.44 MB media */

{
M1PT44MB, /* media type */
512, /* sector size */
18, /* end of track */
79, /* max cylinder */
0x1B, /* gap length */
0x00, /* data rate */
0x0D, /* step rate */
1, /* head unload */
1 /* head load */
},

/* 720 K media */
{
MPT720MB, /* media type */
512, /* sector size */
9, /* end of track */
79, /* max cylinder */
0x2A, /* gap length */
0x02, /* data rate */
0x0E, /* step rate */
1, /* head unload */
1 /* head load */
}
}

```



```

}
};
#define MEDIA_TYPES 3

/* service function forward declarations */
void *motor_control(void *arg);
static void dskt_int(void *arg);
static void dskt_cleanup_push(struct cleanup *clean_anchor, void (*cln_fcn)(),
    void *arg1, void *arg2);
static void dskt_cleanup(struct cleanup *clean_anchor);
static int validate_media(int drive);

/*-----
|
| Name: dskt_init
|
| Function:
| Initializes the diskette device driver, sets the device
| specific pointers and returns.
| No effort is made to access the actual drive at this stage of
| initialization.
|
| Implementation notes:
| No serialization protection provided for this essentially one time
| initialization
|
+-----*/
int dskt_init(driver_t *dswitch, va_list vargs)
{
    int rc;
    int irq_level;
    unsigned char preg4;
    flih_t dskt_flih;
    int dskt_intlvl;
    char *tstack;
    struct cleanup dskt_clean = {0};

    /* setup condition variable */
    rc = pthread_cond_init(&dskt_cond, NULL); /* take defaults */
    if (rc != 0)
    {
        errno = rc;
        return(-1);
    }
}

```

```

/* register cond variable cleanup function */
dskt_cleanup_push(&dskt_clean, (void *)pthread_cond_destroy,
(void *)&dskt_cond, NULL);

/* setup associated mutex */
rc = pthread_mutex_init(&dskt_mutex, NULL);
if (rc != 0)
{
dskt_cleanup(&dskt_clean);
errno = rc;
return(-1);
}
/* register dskt mutex cleanup function */
dskt_cleanup_push(&dskt_clean, (void *)pthread_mutex_destroy,
(void *)&dskt_mutex, NULL);
/* mutex associated with interrupt processing */
rc = pthread_mutex_init(&dint_mutex, NULL);
if (rc != 0)
{
dskt_cleanup(&dskt_clean);
errno = rc;
return(-1);
}
/* register dint mutex cleanup function */
dskt_cleanup_push(&dskt_clean, (void *)pthread_mutex_destroy,
(void *)&dint_mutex, NULL);

/* create interrupt handler */
tstack = (char *)malloc(DSKT_INT_STACK_SIZE); /* typical stack*/
if (tstack == NULL)
{
dskt_cleanup(&dskt_clean);
return(-1);
}
dskt_cleanup_push(&dskt_clean, free, (void *)tstack, NULL);

/* register interrupt handler */
dskt_flih.flih_stack = (void *) (tstack + DSKT_INT_STACK_SIZE);
dskt_flih.flih_function = dskt_int;
dskt_flih.arg = (void *) dskt_cond;
rc = sand_int_install(SAND_FLOPPY_IRQ, &dskt_flih, NULL);
if (rc != 0)
{
dskt_cleanup(&dskt_clean);
return(-1);
}

```

```

}
dskt_cleanup_push(&dskt_clean, (void(*) )sand_int_install,
(void *)SAND_FLOPPY_IRQ, NULL);

/* let go of FDC hardware reset */
outbyte(DOR, inbyte(DOR) | RESET_BAR);
/* assert software reset */
outbyte(DSR, SWRESET);
/* enable and unmask interrupt */
sand_int_enable(SAND_FLOPPY_IRQ);
dskt_cleanup_push(&dskt_clean, sand_int_disable,
(void *)SAND_FLOPPY_IRQ, NULL);
/* at this point, the newly reset controller chip will post an
| interrupt and stack interrupt status to be removed
*/
dsktdrive[0].d_flags |= INTERRUPT_PENDING;
rc = wait_for_interrupt(0); /* wait for interrupt on drive 0 */
if (rc != 0) /* timeout */
{
dskt_cleanup(&dskt_clean);
return(-1);
}
/* The diskette controller chip stacks interrupt status out of
reset. The initialization code must pull the interrupt status
for each supported driver (4) until sense_interrupt_status
returns invalid command. Then, the physical cylinder number
is retrieved
*/

/* pull rest of interrupt status */
do
{
rc = fdc_put(SENSE_INTERRUPT_STATUS);
if (rc != 0) break; /* indetirminent error */
rc = fdc_get();
if (rc == -1) break;
if ((rc & ST0_IC_MASK) == ST0_INVALID_CMD)
{
rc = 0; /* got them all */
break;
}
/* now, pull pcn */
rc = fdc_get();
if (rc == -1) break;
} while(1);

```

```

if (rc != 0) /* some sort of error when retrieving status */
{
    dskt_cleanup(&dskt_clean);
    return(-1);
}
/* configure */
if (
    fdc_put(CONFIGURE) || fdc_put(0x00)
    || fdc_put(0x57) /* disable poll, enable implied seek
    FIFO threshold of 8 */
    || fdc_put(0x00)) /* precomp of zero */
{
    /* config failure */
    dskt_cleanup(&dskt_clean);
    return(-1);
}

/* create motor control thread */
rc = pthread_create(&dsktdsa.dskt_thread_id, NULL,
    motor_control, (void *) &dskt_mutex);
if (rc != 0)
{
    dskt_cleanup(&dskt_clean);
    errno = rc;
    return(-1);
}
/* push thread cancelation */
dskt_cleanup_push(&dskt_clean, (void(*)pthread_cancel,
    (void *)&dsktdsa.dskt_thread_id, NULL);
rc = package_install(dskt_version, (void *)&dsktdsa, NULL);
if (rc != 0)
{
    dskt_cleanup(&dskt_clean);
    return(-1);
}
*dswitch = dsktdst; /* set method pointers */
return(0);
}
static void dskt_cleanup_push(struct cleanup *clean_anchor, void (*cln_fcn)(),
    void *arg1, void *arg2)
{
    /* This function maintains an array of cleanup routines and
    | associated arguments.
    */
    int i;

```

```

i = clean_anchor->cleanup_depth++;
clean_anchor->cln_array[i].cln_function = cln_fcn;
clean_anchor->cln_array[i].arg1 = arg1;
clean_anchor->cln_array[i].arg2 = arg2;
}

static void dskt_cleanup(struct cleanup *clean_anchor)
{
    while(clean_anchor->cleanup_depth--)
    {
        /* call cleanup function with arg1, arg2, and NULL */
        clean_anchor->cln_array[clean_anchor->cleanup_depth].cln_function(
            clean_anchor->cln_array[clean_anchor->cleanup_depth].arg1,
            clean_anchor->cln_array[clean_anchor->cleanup_depth].arg2, NULL);
    }
}

/*-----
|
| Name: dskt_config
|
| Function:
| Prepares the device driver for use by a specific drive by
| initializing device specific structure
|
+-----*/
static int dskt_config(void **ds_ptr, va_list vargs)
{
    int rc;
    unsigned int drive;

    drive = va_arg(vargs, unsigned int);
    if (drive > MAX_DRIVE_NUM)
    {
        errno = EINVAL;
        return(-1);
    }
    pthread_mutex_lock(&dskt_mutex);
    memset(&dsktdrive[drive], 0x00, sizeof(struct drive));
    /* invalidate media and clear interrupt pending */
    dsktdrive[drive].d_flags &= ~(MEDIA_VALID | INTERRUPT_PENDING);
    /* set perpendicular recording mode */
    if ( fdc_put(PERPENDICULAR_MODE) || fdc_put(0x80 + (0x4 << drive)))
    {

```

```

pthread_mutex_unlock(&dskt_mutex);
return(-1);
}
*ds_ptr = (void *) drive;
pthread_mutex_unlock(&dskt_mutex);
return(0);
}

/*-----
|
| Name: dskt_open
|
| Function:
| Performs open processing for diskette block device driver
| Actual drive access and media validation is deferred until drive is
| needed. This routine validates open flags and marks if this is to
| be readonly access.
|
+-----*/
static int dskt_open(file_block_t *fb, int oflag, va_list ap)
{
    int rc;
    int drive = (int) fb->driver_specific;
    struct drive *drvp = &dsktdrive[drive];
    int i;
    /* validity checks */
    pthread_mutex_lock(&dskt_mutex);
    if ((drvp->d_flags & DRIVE_IN_USE) != 0)
    {
        errno = EBUSY;
        pthread_mutex_unlock(&dskt_mutex);
        return(-1);
    }
    /* check wp flag */
    if ((oflag & O_RDONLY) != 0 )
    {
        drvp->d_flags |= READ_ONLY_FILE;
    }
    drvp->d_flags |= DRIVE_IN_USE;
    pthread_mutex_unlock(&dskt_mutex);
    return(0);
}

/*-----
|

```

```

| Name: dskt_close
|
| Function:
| Performs close processing for diskette device driver
| Marks drive as unused and invalidates media.
|
+-----*/
static int dskt_close(file_block_t *fb)
{
    int drive = (int) fb->driver_specific;
    struct drive *drv = &dsktdrive[drive];

    /* clean flags in general */
    drv->d_flags = 0;
    /* clean flags in specific */
    drv->d_flags &= ~DRIVE_IN_USE;
    /* no real device specific support needed */
    return(0);
}

/*-----
|
| Name: dskt_read
|
| Function:
| Function not supported for block device
|
+-----*/
static ssize_t dskt_read(file_block_t *fb, void *buffer, size_t length)
{
    errno = EINVAL; /* read not supported for this driver */
    return(-1);
}

/*-----
|
| Name: dskt_write
|
| Function:
| Function not supported for block device
|
+-----*/
static ssize_t dskt_write(file_block_t *fb, const void *buffer, size_t length)
{
    errno = EINVAL; /* read not supported for this driver */

```

```

return(-1);
}
/*-----
|
| Name: dskt_select
|
| Function:
| This function not supported for this device
|
+-----*/
int dskt_select(file_block_t *fb, int *events)
{
    errno = EINVAL;
    return(-1);
}

/*-----
|
| Name: dskt_ioctl
|
| Function:
| Performs ioctl processing for diskette driver system
| The following commands are implemented:
| QDEVATTR - Return certain device attributes
| QDSKTATTR - Return certain media attributes
| BLKMEDIA_CHANGE - Determine if media changed for removable device
|
+-----*/
static int dskt_ioctl(file_block_t *fb, int cmd, va_list ap)
{
    int rc;
    block_dev_attr_t *blkattr;
    dsktattr_t *dsktattr;
    int drive = (int) fb->driver_specific;
    struct drive *drv = &dsktdrive[drive];
    int i;

    switch (cmd)
    {
        case QDEVATTR: /* query device attributes */
            blkattr = va_arg(ap, block_dev_attr_t *);
            *blkattr = REMOVABLE_MEDIA;
            return(0);
        break;
    }

```



```

case QDSKTATTR: /* query media attributes */
    dsktattr = va_arg(ap, dsktattr_t *);
    *dsktattr = 0; /* indicates media is invalid ~ MEDIA_VALID */
    pthread_mutex_lock(&dskt_mutex);
    validate_media(drive);
    if ((drv->d_flags & MEDIA_VALID) == 0)
    {
        pthread_mutex_unlock(&dskt_mutex);
        return(0);
    }
    /* process flags */
    /* set media type and valid flag */
    *dsktattr = drv->media.media_type | DSKT_VALID;
    motor_on(drive);
    if ((inbyte(STATREG_A) & WRITE_PROTECT) == 0)
    {
        *dsktattr |= DSKT_WP;
    }
    pthread_mutex_unlock(&dskt_mutex);
    return(0);

break;

case BLKMEDIA_CHANGE:
    /* check to see if media has been changed since last call */
    pthread_mutex_lock(&dskt_mutex);
    motor_on(drive); /* motor must be on to check changed line */
    /* check changed line on diskette drive */
    if ((inbyte(DIR) & DSKT_CHANGE) == 0) /* No change */
    {
        pthread_mutex_unlock(&dskt_mutex);
        return(0);
    }
    /* Media now invalidated, try to revalidate */
    rc = validate_media(drive);
    pthread_mutex_unlock(&dskt_mutex);
    if (rc == 0) return(1);
    return(rc);
break;

default:
    errno = EINVAL; /* invalid command */
    return(-1);
}

```

```

}

/*-----
|
| Name: dskt_strategy
|
| Function:
| Performs block oriented i/o for diskette
|
+-----*/
static int dskt_strategy(file_block_t *fb, block_req_t *blk_req)
{
    int rc;
    int i;
    int drive = (int) fb->driver_specific;
    struct drive *drv = &dsktdrive[drive];
    int rel_block = blk_req->block_id;
    int isRead;
    size_t transfer_size;
    int retry_count;
    struct dma_stat dm_stat;

#ifdef DEBUG
    char *rtype;
#endif
    /* validity check */
    switch(blk_req->request_type)
    {
        case BLOCK_READ:
            isRead = 1;
#ifdef DEBUG
            rtype = "Read";
#endif
            break;
        case BLOCK_WRITE:
            isRead = 0;
#ifdef DEBUG
            rtype = "Write";
#endif
            break;
        default:
            errno = EINVAL;
            return(-1);
    }
    pthread_mutex_lock(&dskt_mutex);

```

```

if ((drvp->d_flags & MEDIA_VALID) == 0) /* need to validate media */
{
    motor_on(drive);
    rc = validate_media(drive);
    if (rc != 0)
    {
        pthread_mutex_unlock(&dskt_mutex);
        return(-1);
    }
}
/* compute target, converting block ID to cyl:head:sector */
drvp->cylinder = rel_block / (drvp->media.end_of_track * 2);
if (drvp->cylinder > drvp->media.last_cylinder)
{
    errno = EINVAL;
    pthread_mutex_unlock(&dskt_mutex);
    return(-1);
}
/* now, compute residual block */
rel_block -= drvp->cylinder * drvp->media.end_of_track * 2;
if (rel_block >= drvp->media.end_of_track)
{
    drvp->head = 1;
    drvp->sector = rel_block - drvp->media.end_of_track + 1;
}
else
{
    drvp->head = 0;
    drvp->sector = rel_block + 1;
}
#ifdef DEBUG
printf("%s Target RBA %d C:H:S %d:%d:%d\n", rtype, blk_req->block_id,
    drvp->cylinder, drvp->head, drvp->sector);
#endif

motor_on(drive); /* start motor */
transfer_size = drvp->media.sector_size;
retry_count = 3;
/* flush required for 601 */
dcache_flush(blk_req->block_buffer, transfer_size);
do /* retry loop */
{
    if (isRead)
    {
        if (fb->file_mode & O_WRONLY != 0) /* write only diskette device */

```

```

{
    errno = EINVAL;
    pthread_mutex_unlock(&dskt_mutex);
    return(-1);
}

    dma_setup(DMA_FLOPPY_CHANNEL, DMA_IO_TO_MEM,
blk_req->block_buffer,
transfer_size);
}
else /* write case */
{
    /* check if device opened R/O or if media write protected */
    if ((drv->d_flags & READ_ONLY_MEDIA) != 0 ||
(drv->d_flags & READ_ONLY_FILE) != 0 )
    {
        errno = EROFS;
        pthread_mutex_unlock(&dskt_mutex);
        return(-1);
    }
    dma_setup(DMA_FLOPPY_CHANNEL, DMA_MEM_TO_IO,
blk_req->block_buffer,
transfer_size);
}
/* issue read or write to fdc */
/* prime interrupt */
drv->d_flags |= INTERRUPT_PENDING;

/* issue controller command */
if (isRead) rc = fdc_put(READ_DATA);
else rc = fdc_put(WRITE_DATA);
if (rc != 0)
{
    dma_disable(DMA_FLOPPY_CHANNEL);
    pthread_mutex_unlock(&dskt_mutex);
    return(-1);
}
/* issue rest of parameters */
/* controller chip requires a sequence of bytes describing the
operation to be performed */

if (
    fdc_put(drive + (drv->head << 2)) ||
    fdc_put(drv->cylinder) ||
    fdc_put(drv->head) ||

```

```

    fdc_put(drvp->sector) ||
    fdc_put(FDC_SECTOR_CODE) ||
    fdc_put(drvp->media.end_of_track) ||
    fdc_put(drvp->media.gap_length) ||
    fdc_put(BOGUS_DTL)
)
{
    dma_disable(DMA_FLOPPY_CHANNEL);
    pthread_mutex_unlock(&dskt_mutex);
    return(-1);
}
#ifdef DEBUG
printf("Waiting for interrupt in strategy\n");
#endif
rc = wait_for_interrupt(drive);
#ifdef DEBUG
printf("Interrupt complete rc %d status: %#x\n", rc, drvp->status_0);
#endif
if (rc != 0)
{
#ifdef DEBUG
printf("Interrupt complete rc %d status: %#x\n", rc, drvp->status_0);
#endif
pthread_mutex_unlock(&dskt_mutex);
return(-1);
}
if ((drvp->status_0 & ST0_IC_MASK) != ST0_CMD_END )
{
    /* use dskt_buffer as a scratchpad */
    rc = sprintf(blk_req->block_buffer,
        "dsktLib.a status:\n0: %#x\n1: %#x\n2: %#x\ncyl: %d\nhead:i %d\nsector:
        %d\n",
        drvp->status_0,
        drvp->status_1,
        drvp->status_2,
        drvp->status_cyl,
        drvp->status_head,
        drvp->status_sector);
    errlog(dskt_buffer, rc+1);
    /* retry operation */
    /* disable DMA channel, recalibrate drive and try again */
    dma_disable(DMA_FLOPPY_CHANNEL);
    rc = recalibrate(drive);
    if (rc != 0)
    {

```

```

pthread_mutex_unlock(&dskt_mutex);
return(-1);
}
continue; /* iterates on retry loop */
}
/* check DMA ending status */
dma_status(DMA_FLOPPY_CHANNEL, &dm_stat);
if ( dm_stat.current_count != 0xFFFF) /* final count not right */
{
rc = sprintf(
blk_req->block_buffer,
"dsktLib.a DMA error \n Status reg: %x Address: %p Count: %d\n",
dm_stat.stat_reg, dm_stat.current_addr, dm_stat.current_count);
errlog(blk_req->block_buffer, rc+1);
errno = EIO;
pthread_mutex_unlock(&dskt_mutex);
return(-1);
}
if (!isRead) /* all done if write */
{
#ifdef DEBUG
printf("Write operation complete\n");
#endif
pthread_mutex_unlock(&dskt_mutex);
return(0);
}
#ifdef DEBUG
printf("Read operation complete\n");
#endif
pthread_mutex_unlock(&dskt_mutex);
return(0);
} while(--retry_count); /* retry operation */
/* retry loop failed */
errno = EIO;
pthread_mutex_unlock(&dskt_mutex);
return(-1);
}

/*-----
|
| Name: dskt_int
|
| Function:
| diskette drive first level interrupt handler
|

```

```

+-----*/
static void dskt_int(void *arg)
{
    int rc;
    int status;
    int drive;

    while(1)
    {
        if ((inbyte(MSR) & (RQM | DIO)) == (RQM | DIO))/* results pending */
        {
            status = fdc_get();
            if (status == -1) break; /* who knows ????? */
            drive = status & ST0_DS_MASK; /* select drive */
            if (drive > MAX_DRIVE_NUM) break; /* unsupported drive interrupt */
            if ((dsktdrive[drive].d_flags & INTERRUPT_PENDING) == 0)
                /* unexpected interrupt */
            {
                dsktdrive[drive].d_flags |= SEQUENCE_ERROR;
                break;
            }
            /* interrupt on this drive expected at this point */
            dsktdrive[drive].status_0 = status;
            dsktdrive[drive].d_flags &= ~INTERRUPT_PENDING;
            do
            {
                if ((dsktdrive[drive].status_1 = fdc_get()) == -1) break;
                if ((dsktdrive[drive].status_2 = fdc_get()) == -1) break;
                if ((dsktdrive[drive].status_cyl = fdc_get()) == -1) break;
                if ((dsktdrive[drive].status_head = fdc_get()) == -1) break;
                if ((dsktdrive[drive].status_sector = fdc_get()) == -1) break;
                if ((dsktdrive[drive].status_sectsize = fdc_get()) == -1) break;
                pthread_cond_signal(&dskt_cond);
                return; /* one of the "good" returns */
            } while(0);

            dsktdrive[drive].d_flags |= SEQUENCE_ERROR;
            pthread_cond_signal(&dskt_cond);
            break;
            return;
        }
        else /* sense interrupt status because no status pending */
        {
            rc = fdc_put(SENSE_INTERRUPT_STATUS);
            if (rc != 0) break; /* indetirminent error */

```

```

status = fdc_get();
if (status == -1) break; /* who knows ??? */
if ((status & ST0_IC_MASK) == ST0_INVALID_CMD) break; /* false alarm*/
drive = status & ST0_DS_MASK; /* select drive */
if (drive > MAX_DRIVE_NUM) break; /* unsupported drive interrupt */
if ((dsktdrive[drive].d_flags & INTERRUPT_PENDING) == 0)
/* unexpected interrupt */
{
dsktdrive[drive].d_flags |= SEQUENCE_ERROR;
break;
}
/* interrupt on this drive expected at this point */
do
{
dsktdrive[drive].status_0 = status;
if ((dsktdrive[drive].status_cyl = fdc_get()) == -1) break;
dsktdrive[drive].d_flags &= ~INTERRUPT_PENDING;
pthread_cond_signal(&dskt_cond);
return; /* one of the "good" returns */
} while(0);

dsktdrive[drive].d_flags |= SEQUENCE_ERROR;
dsktdrive[drive].d_flags &= ~INTERRUPT_PENDING;
pthread_cond_signal(&dskt_cond);
break;
}
}
return;
}

```

Sample SCSI Device Driver

The following section contains a skeleton SCSI adapter device driver and its include file.

Include File for Skeleton SCSI Adapter Device Driver

```
#ifndef _ex_sadpt_h
#define _ex_sadpt_h

/*-----+
| Function prototypes for the device driver.
+-----*/
int scsiadpt_init(driver_t *dsw, va_list vargs);

#endif /* _ex_sadpt_h */
```

Sample Skeleton SCSI Adapter Device Driver

```
#include <sys/devLib.h>
#include <sys/devDriver.h>
#include <sys/ksyscall.h>
#include <time.h>
#include <flih.h>
#include <stdarg.h>
#include <errno.h>
#include <sys/select.h>
#include <fcntl.h>
#include <string.h>
#include <signal.h>
#include <sys/ioctl.h>
#include <semaphore.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/scsiLib.h>
#include "ex_sadpt.h"

/* Misc SCSI constants */
#define SCSIADPT_MAX_DEVICES      8
#define SCSIADPT_STK_SIZE        1024
#define DEFAULT_PERIOD 100 /* 2.5 MB/s (example default) */
#define BEST_PERIOD 50 /* 5.0 MB/s (example best) */
#define DEFAULT_OFFSET 0 /* Means asynchronous */
#define BEST_OFFSET 0 /* Perhaps, sync is not supported */
#define DEFAULT_WIDTH 0 /* Means narrow */
```

```

#define BEST_WIDTH 0 /* perhaps wide not supported */

/* Adapter Control Structure */

typedef struct scsiadpt_dds {
    scsi_adapter_parm_t adapter_parm;
    sem_t          access_sem;
    int            clean_acc_sem;
    flih_t         flih_struct;
    int            port_open;    /* Count of open requests */
} scsiadpt_dds_t;

/*-----+
| Function prototypes for the SCSI Adapter device driver.
+-----*/

/* Device driver initialization function and methods - prototypes */

static int scsiadpt_configure(void **dds, va_list vargs);
static int scsiadpt_open    (file_block_t *fb, int oflag, va_list vargs);
static int scsiadpt_close   (file_block_t *fb);
static int scsiadpt_read    (file_block_t *fb, void *buffer, size_t length);
static int scsiadpt_write   (file_block_t *fb, const void *buffer, size_t length);
static int scsiadpt_ioctl   (file_block_t *fb, int cmd, va_list vargs);
static int scsiadpt_select  (file_block_t *fb, int *flags);
static int scsiadpt_strategy (file_block_t *fb, block_req_t *blk_req);

/* prototypes for internally used functions */

static void scsiadpt_cleanup(scsiadpt_dds_t *local_dds);
static void scsiadpt_flih(void *arg);
static int init_dds_xfer_parms(scsiadpt_dds_t *local_dds);
static int reset_dds_xfer_parms(scsiadpt_dds_t *local_dds);
static int reset_scsi_chip(scsiadpt_dds_t *local_dds);
static int test_scsi_chip(scsiadpt_dds_t *local_dds);
static int init_scsi_chip(scsiadpt_dds_t *local_dds);
static int reset_bus(scsiadpt_dds_t *local_dds);

static char      sccsid[]="@(#) ex_sadpt.a1.1 3/13/95";

static driver_t   scsiadpt_dt={scsiadpt_configure, scsiadpt_open,
                                scsiadpt_close, scsiadpt_read,
                                scsiadpt_write, scsiadpt_ioctl,
                                scsiadpt_select, scsiadpt_strategy};

```

```

int scsiadpt_init(driver_t *dsw, va_list vargs)
{
    /*-----+
    | Install methods and the package for the adapter device driver.
    +-----*/
    *dsw = scsiadpt_dt;
    package_install(sccsid, NULL, NULL);
    return 0;
}

static int scsiadpt_configure(void **dds, va_list vargs)
{
    /*-----+
    | vargs:
    |   SCSI_initiator_id,
    |   base chip address,
    |   external interrupt level
    |   dma channel
    |   queue depth (number of command allowed to send at a time) - must be 1!
    |
    | algorithm:
    |   check vargs,
    |   allocate and initialize control structure
    |   initialize needed semaphores
    |   install flih
    |   reset and initialize the SCSI chip
    |   test to see if chip responds at passed address
    |   reset SCSI bus
    |   return pointer to allocated control structure
    |
    | Note: If the first varg is -1, this is a request to uninstall
    | the device which will be done as long as there are no opens outstanding.
    +-----*/

    int            scsi_id;
    unsigned char  *chip_address;
    int            interrupt_level;
    int            dma_channel;
    int            queue_depth;
    scsiadpt_dds_t *local_dds;

    /* extract the first variable arguments */
    scsi_id = va_arg(vargs, int);
    /* check for uninstall */
    if (scsi_id == -1)

```

```

{
    local_dds = (scsiadpt_dds_t *) *dds;
    if (local_dds->port_open != 0)
    {
        errno = EINVAL;
        return -1;
    }
    scsiadpt_cleanup(local_dds);
    return 0;
}

/* extract the remaining args */
chip_address = va_arg(vargs, unsigned char *);
interrupt_level = va_arg(vargs, int);
dma_channel = va_arg(vargs, int);
queue_depth = va_arg(vargs, int);
/* ensure (0 <= SCSI_ID < SCSIADPT_MAX_DEVICES), queue depth is 1*/
if (scsi_id < 0 || scsi_id >= SCSIADPT_MAX_DEVICES || queue_depth != 1)
{
    errno = EINVAL;
    return -1;
}

/* Allocate and initialize the control structure. */
local_dds = (scsiadpt_dds_t *) malloc(sizeof(scsiadpt_dds_t));
if (local_dds == NULL)
{
    return -1;
}
local_dds->clean_acc_sem = false;
local_dds->flih_struct.flih_stack = NULL;
local_dds->adapter_parm.scsi_initiator_id = scsi_id;
local_dds->adapter_parm.adapter_base_address = chip_address;
local_dds->adapter_parm.interrupt_level = interrupt_level;
local_dds->adapter_parm.dma_channel = dma_channel;
local_dds->adapter_parm.queue_depth = queue_depth;
local_dds->adapter_parm.max_scsi_devices = SCSIADPT_MAX_DEVICES;
local_dds->port_open = 0;
init_dds_xfer_parms(local_dds);
/* Initialize semaphores. */
if (sem_init(&local_dds->access_sem, 0, queue_depth))
{
    scsiadpt_cleanup(local_dds);
    return -1;
}
local_dds->clean_acc_sem = true;
/* Install first level interrupt handler */

```

```

local_dds->flih_struct.flih_stack = (void *) malloc(SCSIADPT_STK_SIZE);
if (local_dds->flih_struct.flih_stack == NULL)
{
    scsiadpt_cleanup(local_dds);
    return -1;
}
local_dds->flih_struct.flih_stack =
    (void *) ((int) local_dds->flih_struct.flih_stack + SCSIADPT_STK_SIZE);
local_dds->flih_struct.flih_function = (void *) scsiadpt_flih;
local_dds->flih_struct.arg = (void *) local_dds;
if (int_install((event_t) local_dds->adapter_parm.interrupt_level,
                &local_dds->flih_struct, NULL))
{
    scsiadpt_cleanup(local_dds);
    return -1;
}
/* Put code here to configure and enable interrupts as needed */
/* reset and initialize and test the SCSI chip */
reset_scsi_chip(local_dds);
init_scsi_chip(local_dds);
if (test_scsi_chip(local_dds))
{
    scsiadpt_cleanup(local_dds);
    return -1;
}
/* reset the scsi bus */
reset_bus(local_dds);
/* Return local_dds pointer in *dds */
*dds=(void *)local_dds;
return 0;
}

static void scsiadpt_cleanup(scsiadpt_dds_t *local_dds)
{
    if (local_dds != NULL)
    {
        if (local_dds->flih_struct.flih_stack != NULL)
        {
            int_disable(local_dds->adapter_parm.interrupt_level);
            int_install((event_t) local_dds->adapter_parm.interrupt_level,
                        NULL, NULL);

            free((void *)
                ((int) local_dds->flih_struct.flih_stack - SCSIADPT_STK_SIZE));
        }
        if (local_dds->clean_acc_sem)

```

```

        {
            sem_destroy(&local_dds->access_sem);
        }
        free(local_dds);
    }
}

static int scsiadpt_open(file_block_t *fb, int oflag, va_list vars)
{
    /*-----+
    | algorithm:
    | set open flag
    +-----*/
    scsiadpt_dds_t *local_dds = (scsiadpt_dds_t *) fb->driver_specific;
    local_dds->port_open = 1;
    return 0;
}

static int scsiadpt_close(file_block_t *fb)
{
    /*-----+
    | algorithm:
    | clear open flag
    +-----*/
    scsiadpt_dds_t *local_dds = (scsiadpt_dds_t *) fb->driver_specific;
    local_dds->port_open = 0;
    return 0;
}

static int scsiadpt_read(file_block_t *fb, void *buffer, size_t length)
{
    {
        errno = EINVAL;
        return -1;
    }
}

static int scsiadpt_write(file_block_t *fb, const void *buffer, size_t length)
{
    {
        errno = EINVAL;
        return -1;
    }
}

static int scsiadpt_ioctl(file_block_t *fb, int cmd, va_list vars)
{
    {
        int rc = 0;
        int i;
    }
}

```

```

scsiadpt_dds_t      *local_dds = (scsiadpt_dds_t *) fb->driver_specific;
scsi_adapter_parm_t *parm_ptr;
scsi_command_t      *cmd_ptr;

```

```

switch (cmd)
{

```

```

    case RESET_ADAPTER:

```

```

        /*-----+
        | algorithm:
        |   This request IS serialized.
        |   Resets the chip and reset the SCSI bus
        +-----*/
        if (sem_wait(&local_dds->access_sem)) return -1;
        reset_scsi_chip(local_dds);
        init_scsi_chip(local_dds);
        reset_bus(local_dds);
        reset_dds_xfer_parms(local_dds);
        sem_post(&local_dds->access_sem);
        break;

```

```

    case RESET_SCSI_BUS:

```

```

        /*-----+
        | algorithm:
        |   This request IS serialized.
        |   Reset the SCSI bus
        |   reset the dds transfer parms
        +-----*/
        if (sem_wait(&local_dds->access_sem)) return -1;
        reset_bus(local_dds);
        reset_dds_xfer_parms(local_dds);
        sem_post(&local_dds->access_sem);
        break;

```

```

    case EXECUTE_SCSI_COMMAND:

```

```

        /*-----+
        | vargs:
        |   pointer to SCSI command structure
        |
        | algorithm:
        |   This request IS serialized.
        |   Issue and oversee the execution of a SCSI command to a SCSI
        |   target/lun. The target, lun, command and other information
        |   are specified in the SCSI command structure.
        +-----*/
        cmd_ptr = va_arg(vargs, scsi_command_t *);

```

```

if (sem_wait(&local_dds->access_sem)) return -1;
/*
| Put LOTS of code here to execute the SCSI command passed in the
| command structure pointed to by cmd_ptr. Don't forget to qualify
| the rc by the check_expected and trunc_transfer_ok flags in the
| command structure. If the command fails, change the rc and update
| the cmd_ptr->additional_error field.
*/
sem_post(&local_dds->access_sem);
break;

case READ_ADAPTER_PARAMETERS:
/*-----+
| vargs:
|   pointer to adapter parm block
|
| algorithm:
|   This request IS serialized.
|   Copy current adapter parameters into the output parm block.
+-----*/
parm_ptr = va_arg(vargs, scsi_adapter_parm_t *);
if (sem_wait(&local_dds->access_sem)) return -1;
*parm_ptr = local_dds->adapter_parm;
sem_post(&local_dds->access_sem);
break;

case WRITE_ADAPTER_PARAMETERS:
/*-----+
| vargs:
|   pointer to adapter parm block
|
| algorithm:
|   This request IS serialized.
|   Update the current adapter parameters with the data in the
|   input parm block. Do not update read only parms.
+-----*/
parm_ptr = va_arg(vargs, scsi_adapter_parm_t *);
if (sem_wait(&local_dds->access_sem)) return -1;
for (i = 0; i < local_dds->adapter_parm.max_scsi_devices; ++i)
{
    local_dds->adapter_parm.requested_xfer_parm[i] =
        parm_ptr->requested_xfer_parm[i];
}
sem_post(&local_dds->access_sem);
break;

```



```

        default:
            /*-----+
            | Unsupported IOCTL command
            +-----*/
            errno = EINVAL;
            rc = -1;
        }

    return rc;
}

static int scsiadpt_select(file_block_t *fb, int *flags)
{
    errno = EINVAL;
    return -1;
}

static int scsiadpt_strategy(file_block_t *fb, block_req_t *blk_req)
{
    errno = EINVAL;
    return -1;
}

static void scsiadpt_flih(void *arg)
{
    scsiadpt_dds_t *local_dds = (scsiadpt_dds_t *) arg;
    /* First level interrupt handler code goes here */
    return;
}

static int init_dds_xfer_parms(scsiadpt_dds_t *local_dds)
{
    /*
    | This code initializes the dds transfer parms, the requested transfer
    | parms, and the best transfer parms.
    */
    int i;
    for (i = 0; i < local_dds->adapter_parm.max_scsi_devices; ++i)
    {
        local_dds->adapter_parm.current_xfer_parm[i].period =
        DEFAULT_PERIOD;
        local_dds->adapter_parm.current_xfer_parm[i].offset =
        DEFAULT_OFFSET;
        local_dds->adapter_parm.current_xfer_parm[i].width = DEFAULT_WIDTH;
    }
}

```

```

        local_dds->adapter_parm.requested_xfer_parm[i].period =
DEFAULT_PERIOD;
        local_dds->adapter_parm.requested_xfer_parm[i].offset =
DEFAULT_OFFSET;
        local_dds->adapter_parm.requested_xfer_parm[i].width =
DEFAULT_WIDTH;
    }
    local_dds->adapter_parm.best_xfer_parm.period = BEST_PERIOD;
    local_dds->adapter_parm.best_xfer_parm.offset = BEST_OFFSET;
    local_dds->adapter_parm.best_xfer_parm.width = BEST_WIDTH;
    return 0;
}

static int reset_dds_xfer_parms(scsiadpt_dds_t *local_dds)
{
    /*
    | This code resets the dds transfer parms to asynchronous and narrow for
    | each possible target.
    */
    int i;
    for (i = 0; i < local_dds->adapter_parm.max_scsi_devices; ++i)
    {
        local_dds->adapter_parm.current_xfer_parm[i].period =
DEFAULT_PERIOD;
        local_dds->adapter_parm.current_xfer_parm[i].offset =
DEFAULT_OFFSET;
        local_dds->adapter_parm.current_xfer_parm[i].width = DEFAULT_WIDTH;
    }
    return 0;
}

static int reset_scsi_chip(scsiadpt_dds_t *local_dds)
{
    /* Put code here to reset the SCSI adapter. */
    return 0;
}

static int test_scsi_chip(scsiadpt_dds_t *local_dds)
{
    /*
    | Put code here to test access to the SCSI adapter as well as to perform
    | some adapter self test if desired. Return 0 is OK, otherwise return
    | non-zero and set errno to EIO.
    */
    return 0;
}

```

```

}

static int init_scsi_chip(scsiadpt_dds_t *local_dds)
{
    /* Put code here to initialize the SCSI adapter */
    return 0;
}

static int reset_bus(scsiadpt_dds_t *local_dds)
{
    /* Put code here to reset the SCSI bus on this adapter */
    return 0;
}

```


Index

- arithmetic negation operator 18-7
- decrement operator 18-7

Symbols

- != not equal to operator 18-8
- % remainder operator 18-7
- & bitwise AND operator 18-8
- && logical AND operator 18-8
- () operators
 - calling functions 18-7
 - grouping expressions 18-7
- (-) subtraction operator 18-7
- * indirection operator 18-7
- + addition operator 18-7
- ++ increment operator 18-7
- / division operator 18-7
- < less than operator 18-8
- << left shift operator 18-8
- <= less than or equal to operator 18-8
- = assignment operator 18-8
- == equal to operator 18-8
- > arrow operator 18-7
- > greater than operator 18-8
- >= greater than or equal to operator 18-8
- >> right shift operator 18-7
- [] array reference operators 18-7
- \ backslash character 18-6
- ^ bitwise exclusive OR operator 18-8
- | bitwise inclusive OR operator 18-8
- || logical OR operator 18-8
- ~ bitwise negation operator 18-7

A

- address operator 18-7
- AIX commands
 - ar 17-3
 - as-emb 17-2
 - ld 17-4

- m4 17-2
- make 17-9
- xlC 17-2

ANSI C libraries 3-9

applications

- assembler language programs
 - ELF object 17-7
 - XCOFF object 17-2
- booting over TCP/IP 17-13
- building 17-1
- building a loadable module 17-14
- building sample programs 17-12
- building the diskettes 17-14
- compiling 17-13
- loading bootable files 17-13
- sample configuration 17-12
- sample panic module 17-12
- sample source files 17-11
- using host development tools 17-1
- using the AIX C compiler 17-2
- using the AIX linker 17-4
- using the ELF linker 17-8
- using the High C/C++ compiler 17-6
- working with archive files
 - ELF 17-7
 - XCOFF 17-3

ARP processing, example 13-7

ATA/IDE support 11-20

attaching a network interface drivers 13-2

attribute objects, threads 6-9

B

block device drivers

- close() 8-8
- ioctl() 8-9
- open() 8-8
- strategy() 8-9

boot image object

- ELF 17-8

- XCOFF 17-4
- bounded execution 5-2
- breakpoints
 - clearing 18-20
 - listing status 18-21
 - setting 18-19
- building diskettes for applications 17-14
- building loadable application modules 17-14

C

- C interpreter
 - `/* */` 18-10
 - `//` 18-10
 - `{ }` 18-9
 - `char` 18-8
 - comments 18-10
 - constants 18-6
 - `continue` 18-9
 - control specifier 18-9
 - differences 18-6
 - `do` 18-9
 - `double` 18-8
 - `else` 18-9
 - examples 18-11
 - expression operators 18-7
 - `float` 18-8
 - `for` 18-9
 - identifiers 18-7
 - `if` 18-9
 - `int` 18-8
 - `long` 18-8
 - operators
 - binary 18-7
 - unary 18-7
 - restrictions 18-6
 - restrictions and differences 18-12
 - `return` 18-9
 - shell commands 18-11
 - `short` 18-8
 - special characters 18-6
 - storage class specifiers 18-8
 - struct-specifier 18-9

- structure specifier 18-9
- structure specifiers 18-9
- type specifiers 18-8
- `unsigned` 18-8
- user-defined functions 18-10
- `void` 18-8
- `while` 18-9
- C Set ++ 3-21, 17-5
- C++ support 3-21, 17-5
- cancel safe 6-3
- card services 11-1, 11-5, 11-6, 11-21, 11-24, 11-26
- character device drivers
 - `close` 8-5
 - `ioctl` 8-6
 - `open` 8-5
 - `read` 8-6
 - `select` 8-7
 - `write` 8-6
- character handling 3-9
- character sets
 - decimal digits xxi
 - graphic characters xxi
 - hexadecimal digits xxi
 - lowercase letters xxi
 - portable file name characters xxii
 - uppercase letters xxi
- CHMOD command 10-4
- clearing breakpoints 18-20
- cLib.a library 3-9
- commands
 - `arppc` 17-7
- compiling the applications 17-13
- configuring OS Open 4-1
- `continue` command 18-21
- conventions
 - highlighting xx
 - input xix
 - numeric notation xix
- creating OS Open object files
 - ELF 17-6

- XCOFF 17-2
- cross development 1-2, 1-6, 17-1
- csLib 11-1, 11-5, 11-6, 11-21
- csLib.a 3-17

D

- date 3-12
- dbLib.a library 3-19
- debugging a thread 18-23
- debugging support library 3-19
- default thread 18-17
- deterministic execution 5-2
- device and file support library 3-13
- device drivers
 - initializing 8-2
 - installing 8-2
 - sample async driver B-1
 - sample block driver B-20
 - sample SCSI driver B-39
- devLib.a library 3-13
- disassembler, symbolic 18-18
- display 18-16
 - example 18-16
- displaying memory contents 18-19
- DOS commands
 - make 17-9
- DOS file system support library 3-16
- DOS logical file system
 - device driver installation 10-1
 - device installation 10-1
 - I/O control 10-3
 - opening files 10-2
 - reading files 10-3
 - writing files 10-3
- dynamic loader library 3-20
- dynamically loadable object
 - ELF 17-8
 - XCOFF 17-4

E

- ELF 1-2, 1-3

- ELF assembler commands
 - asppc 17-7
- ELF linker commands
 - ldppc 17-8
- ExCA 11-1, 11-5, 11-12
- exchangeable card architecture 11-1,
11-5, 11-12
 - ExCA specification 11-1
- eXtended Common Object File Format 1-3, 17-1
- Extended Link Format 1-2, 1-3

F

- fatLib.a library 3-16
- file blocks 8-3
- file redirection
 - more file 18-4
- file support
- file transfer protocol
 - description 14-1
- file transfer protocol support library 3-18
- FLIH 7-1
- floating point emulation
 - library description 3-21
- FPATHCONF command 10-4
- fpeLib.a library 3-21
- fsLib.a library 3-9
- FSYNC command 10-4
- ftp 14-1
- ftpLib.a library 3-18
- functions for interrupt handlers 7-4

G

- general programming utilities 3-11

H

- High C/C++ compiler commands
 - hcppc 17-6, 17-7
- host development tools
 - AIX assembler 17-2
 - AIX linker 17-4
 - archiver

- ELF 17-7
- XCOFF 17-3
- compiling C source programs
 - ELF object 17-6
 - XCOFF object 17-2
- creating OS Open object files
 - ELF 17-6
 - XCOFF 17-2
- ELF assembler 17-7
- ELF linker 17-8
- using the make command 17-9
- working with object files
 - ELF 17-7
 - XCOFF 17-3
- host system requirements 1-2
- how to use this book xviii
- how to write a device driver 8-2

I

- I/O control, DOS logical file system 10-3
- IBM XL C Compiler/6000 17-1
- implementing a logical file system 9-1
- industry standards
 - ANSI C 2-1
 - draft 2-2
 - POSIX 2-2
 - support for ANSI C Standard 2-2
 - support for POSIX 2-3
- input/output 3-10
- instruction stepping 18-21
- instruction tracing 18-21
- integral device drivers
 - memory locking 3-15
 - null device 3-14
 - shared memory device 3-14
- interrupt and fault handling 3-6
- interrupt handlers
 - execution environment 7-3
 - safe functions 7-4
- ioctl commands
 - CHMOD 10-4
 - FPATHCONF 10-4

- FSYNC 10-4
- LINK 10-3
- MKDIR 10-3
- OPENDIR 10-3
- PATHCONF 10-4
- REaddir 10-3
- READDR_R 10-3
- RENAME 10-3
- REWINDDIR 10-3
- RF_ACCESS 10-3
- RF_FSTAT 10-3
- RF_STAT 10-3
- RMDIR 10-3
- SEEK 10-4
- UNLINK 10-3
- UTIME 10-4

IOCTL processing, network interfaces 13-6

K

- kadtLib.a library 3-20
- kernal abstract data types 3-20
- kernel configuration block 4-1
- kernel trace snapshot 18-24

L

- ldrLib.a library 3-20
- library description
 - cLib.a 3-9
 - dbLib.a 3-19
 - devLib.a 3-13
 - fatLib.a 3-16
 - fpeLib.a 3-21
 - fsLib.a 3-9
 - ftpLib.a 3-18
 - kadtLib.a 3-20
 - ldrLib.a 3-20
 - mathLib.a 3-9
 - netLib.a 3-17
 - nfsLib.a 3-17
 - ramdLib.a 3-16
 - rmsLib.a 3-21
 - rngLib.a 3-16

- rpcLib.a 3-18
- rsldLib.a 3-19
- rtx.o 3-3
- rtxLib.a 3-3
- rtxmin.o 3-3
- runLib.a 3-17
- shell.o 3-19
- symLib.a 3-20
- tcpiLib.a 3-17
- telnet.o 3-18
- ttyLib.a 3-15
- xlclib.a 3-20
- LINK command 10-3
- listi 18-15
 - example 18-15
- listing breakpoint status 18-21
- loading bootable images 17-13
- local debugger 18-13
 - registers 18-17
- logical file system driver installation 10-1
- low-level POSIX I/O support 3-13

M

- make command 17-9
- mathLib.a library 3-9
- memory locking 3-15
- memory management 3-4
- message queue 3-6
- minimal kernel 3-3
- MKDIR command 10-3

N

- netLib.a library 3-17
- network interface drivers
 - ARP processing example 13-7
 - attachment 13-2
 - general concepts 13-1
 - input 13-4
 - IOCTL processing 13-6
 - network software interrupts 13-1
 - output 13-2

- serializing network data structures 13-1
- network support library 3-17
- networking features
 - NetLib.a
 - general information 12-3
 - prerequisite libraries 12-6
 - TcpipLib.a
 - corequisite libraries 12-3
 - general information 12-1
- NFS client support library 3-17, 12-6
- nfsLib.a library 3-17, 12-6
- nonlocal jumps 3-9
- null device driver 3-14

O

- object code, disassembling 18-18
- OPENDIR command 10-3
- opening files, DOS logical file system 10-2
- OpenShell 3-19, 18-1
 - command history 18-3
 - command line editing 18-5
 - examples 18-14
 - features 18-1
 - special file more 18-4
 - symbols 18-13
 - using 18-1
- OpenShell commands
 - at 18-19
 - clear 18-20
 - cont 18-21
 - kernel trace snapshot 18-24
 - listi 18-18
 - nexti 18-21
 - status 18-21
 - where 18-22
- OpenShell prompt 18-5
- OpenShell variables 18-17
- OS Open
 - cross development 1-2, 17-1
 - cross-development environment 1-2, 17-1
 - functions 1-4

- host system requirements 1-2
- overview 1-1
- performance 1-6
- programming language 1-2
- target systems 1-3
- OS Open buffer pool management 3-5
- OS Open debuggers
 - breakpoint status 18-21
 - clearing breakpoints 18-20
 - continue command 18-21
 - debugging a thread 18-23
 - instruction step 18-21
 - instruction trace 18-21
 - kernel trace snapshot 18-24
 - memory display 18-19
 - OpenShell variables 18-17
 - setting breakpoints 18-19
 - stack trace back 18-22
 - symbolic disassembler 18-18
- OS Open heap management 3-4
- OS Open minimal kernel 3-3
- OS Open operating system
- OS Open RAM disk 10-5
- OS Open RAM disk installation 10-5
- OS Open support xxii

P

- panic() function 4-2
- papers, benchmark xxiv
- pataLib.a 3-17
- PATHCONF command 10-4
- pcmcia 11-1
 - ATA/IDE support 11-20
 - card services 11-1, 11-5, 11-6, 11-21, 11-24, 11-26
 - csLib.a 3-17
 - pataLib.a 3-17
 - socket services 11-1, 11-4, 11-6, 11-8, 11-12, 11-13, 11-21, 11-26
 - ssLib.a 3-17
 - support libraries 3-17

- performance 1-6
- Personal Computer Memory Card International Association 11-1
- print 18-16
 - example 18-16
- programming language 1-2
- programming notes 8-9

Q

- queue library 3-16

R

- RAM disk 10-5
- rambuild 10-5
- ramdLib.a library 3-16
- rate monotonic scheduling (RMS)
 - analysis 5-1
 - library description 3-21
 - scheduling algorithms 5-1
- REaddir command 10-3
- READDIR_R command 10-3
- reading files, DOS logical file system 10-3
- reading the syntax diagrams xx
- real-time executive
 - interrupt and fault handling 3-6
 - memory management
 - OS Open heap management 3-4
 - other services 3-8
 - signals 3-5
 - storage management
 - OS Open buffer pool management 3-5
 - thread management 3-4
 - timers and clocks 3-6
- real-time experience
 - memory management 3-4
- real-time programming features
 - bounded execution 5-2
 - deterministic execution 5-2
 - hard real-time systems 5-1
 - RMS analysis 5-1
- related publications xxii

- remote source-line debug library 3-19
- remove 18-11
- RENAME command 10-3
- REWINDDIR command 10-3
- RF_ACCESS command 10-3
- RF_FSTAT command 10-3
- RF_STAT command 10-3
- ring buffer library 3-16
- RMDIR command 10-3
- RMS
 - analysis 5-1
 - library description 3-21
- rmsLib.a library 3-21
- rngLib.a library 3-16
- rpc/xdr protocol support library 3-18
- rpcLib 3-18
- rpcLib.a library 3-18
- rsldLib.a library 3-19
- rtx.o library 3-3
- rtxLib.a library 3-3
- rtxmin.o library 3-3
- run library 3-17
- runLib.a library 3-17

S

- sample asynchronous device driver B-1
- sample configuration 17-12
- sample diskette device driver B-20
- sample panic module 17-12
- sample programs
 - building 17-10
 - executing 17-10, 17-12
- sample SCSI device driver B-39
- SEEK command 10-4
- semaphores 3-7
- set 18-11
- setting breakpoints 18-19
- shared memory device driver 3-14
- shell 18-16
 - C interpreter 18-5

- calling C library functions 18-16
- command line prompt 18-5
 - starting 18-1
 - symbols 18-6
- shell commands
 - display 18-19
 - exit 18-11
 - help 18-11
 - print 18-11
 - remove 18-11
 - set 18-11
 - stepi 18-21
 - unset 18-11
 - whatis 18-11
- shell print options
 - hexadecimal character pointers 18-11
 - hexadecimal characters 18-11
 - hexadecimal integers 18-11
 - octal integers 18-11
- shell.o library 3-19
- signals 3-5
- socket services 11-1, 11-4, 11-6, 11-8, 11-12, 11-13, 11-21, 11-26
- ssLib 11-1, 11-6, 11-12
- ssLib.a 3-17
- stack trace back 18-22
- standards xxiv
- string handling 3-11
- support for real-time systems 5-1
- symbol support library 3-20
- symLib.a library 3-20
- syntax diagrams, how to read xx
- system services
 - description 3-1

T

- TCP/IP protocol support library 3-17, 12-1
- tcipLib.a library 3-17, 12-1
- telnet
 - description 15-1
- Telnet support library 3-18

- telnet.o library 3-18
- tftp 14-1
- thread cancelation 6-2
- thread management 3-4, 6-1
- thread_dump() 18-15
- threads
 - attribute objects 6-9
 - cancelation 6-2
 - condition variable 6-8
 - mutex attribute objects 6-7
 - operations 6-1
 - semaphores 6-9
 - serialization 6-5, 6-11
 - specific data 6-11
 - synchronization 6-7
 - thread management 6-4
 - thread scheduling 6-4
- thread-specific data 6-11
- time 3-12
- timers and clocks 3-6
- trace buffer support 3-7
- trace_snapshot() 3-7, 3-8
- trace_write() 3-8
- tracefmt 3-8
- trc41 3-8
- trcfmt.templ 3-8
- trcrpt 3-8
- trivial file transfer protocol 14-1
- TTY support library 3-15
- ttyLib.a library 3-15

U

- UNLINK command 10-3
- unset 18-11
- using interrupt handlers 7-1
- UTIME command 10-4

V

- variable arguments 3-10

W

- whatis 18-11
- who should use this book xvii
- working with object files
 - ELF 17-7
 - XCOFF 17-3
- writing file system
 - close 9-2
 - ioctl 9-3
 - open 9-2
 - read 9-3
 - select 9-13
 - write 9-3
- writing files, DOS logical file system 10-3
- writing network interface drivers
 - general concepts 13-1
 - mbufs 13-1
 - network software interrupts 13-1
 - serializing network data structures 13-1

X

- XCOFF 1-3, 17-1
- XL C compiler 17-1
- XLC compiler support
 - library description 3-20
- xlclib.a library 3-20

