

Bull

Kernel Extensions and Device Support Programming Concepts

AIX

Bull

Kernel Extensions and Device Support Programming Concepts

AIX

Software

November 1999

BULL ELECTRONICS ANGERS
CEDOC
34 Rue du Nid de Pie – BP 428
49004 ANGERS CEDEX 01
FRANCE

ORDER REFERENCE
86 A2 36JX 02

The following copyright notice protects this book under the Copyright laws of the United States of America and other countries which prohibit such actions as, but not limited to, copying, distributing, modifying, and making derivative works.

Copyright © Bull S.A. 1992, 1999

Printed in France

Suggestions and criticisms concerning the form, content, and presentation of this book are invited. A form is provided at the end of this book for this purpose.

To order additional copies of this book or other Bull Technical Publications, you are invited to use the Ordering Form also provided at the end of this book.

Trademarks and Acknowledgements

We acknowledge the right of proprietors of trademarks mentioned in this book.

AIX[®] is a registered trademark of International Business Machines Corporation, and is being used under licence.

UNIX is a registered trademark in the United States of America and other countries licensed exclusively through the Open Group.

Year 2000

The product documented in this manual is Year 2000 Ready.

The information in this document is subject to change without notice. Groupe Bull will not be liable for errors contained herein, or for incidental or consequential damages in connection with the use of this material.

Contents

Trademarks and Acknowledgements . . . iii

About This Book xv

Who Should Use This Book	xv
How to Use This Book	xv
Overview of Contents	xv
Highlighting	xvi
ISO 9000	xvi
AIX 32-Bit Support for the X/Open UNIX95 Specification	xvi
AIX 32-Bit and 64-Bit Support for the UNIX98 Specification	xvii
Related Publications	xvii
Ordering Publications	xvii

Chapter 1. Kernel Environment. 1

Understanding Kernel Extension Binding	2
Base Kernel Services - the /unix Name Space	2
Using System Calls with Kernel Extensions	3
Using Private Routines	4
Using Libraries	5
Understanding Execution Environments	7
Process Environment	8
Interrupt Environment	8
Understanding Kernel Threads	9
Kernel Threads, Kernel Only Threads, and User Threads	9
Kernel Data Structures	10
Thread Creation, Execution, and Termination	10
Thread Scheduling	10
Thread Signal Handling	10
Using Kernel Processes	11
Introduction to Kernel Processes	11
Accessing Data from a Kernel Process	12
Cross-Memory Services	13
Kernel Process Creation, Execution, and Termination	13
Kernel Process Preemption	14
Kernel Process Signal and Exception Handling	14
Kernel Process Use of System Calls	15
Accessing User-Mode Data While in Kernel Mode	15
Data Transfer Services	15
Using Cross-Memory Kernel Services	16
Understanding Locking	16
Lockl Locks	16
Simple Locks	17
Complex Locks	17
Types of Critical Sections	17
Priority Promotion	17
Locking Strategy in Kernel Mode	17
Understanding Exception Handling	18
Exception Processing	18
Kernel-Mode Exception Handling	18
Implementing Kernel Exception Handlers	20
User-Mode Exception Handling	23

64-bit Kernel Extension Development. 23

Chapter 2. System Calls 25

Differences Between a System Call and a User Function	25
Understanding System Call Execution	25
User Protection Domain	26
Kernel Protection Domain	26
Actions of the System Call Handler	26
Accessing Kernel Data While in a System Call.	27
Preempting a System Call	28
Handling Signals While in a System Call	28
Handling Exceptions While in a System Call	29
Understanding Nesting and Kernel-Mode Use of System Calls	30
Page Faulting within System Calls.	30
Returning Error Information from System Calls	31
System Calls Available to Kernel Extensions	31
System Calls Available to All Kernel Extensions	31
System Calls Available to Kernel Processes Only	32

Chapter 3. Virtual File Systems 33

Logical File System Overview	33
Component Structure of the Logical File System	34
Virtual File System Overview	35
Understanding Virtual Nodes (V-nodes)	35
Understanding Generic I-nodes (G-nodes)	35
Understanding the Virtual File System Interface	36
Understanding Data Structures and Header Files for Virtual File Systems	37
Configuring a Virtual File System	38

Chapter 4. Kernel Services 39

Categories of Kernel Services	39
I/O Kernel Services	39
Block I/O Kernel Services	39
Buffer Cache Kernel Services	40
Character I/O Kernel Services	40
Interrupt Management Services.	40
Memory Buffer (mbuf) Kernel Services	41
DMA Management Kernel Services	42
Block I/O Buffer Cache Kernel Services: Overview	42
Managing the Buffer Cache	43
Using the Buffer Cache write Services	43
Understanding Interrupts.	44
Interrupt Priorities	44
Understanding DMA Transfers	44
Hiding DMA Data	45
Accessing Data While the DMA Operation Is in Progress	45
Kernel Extension and Device Driver Management	
Kernel Services	46
Kernel Extension Loading and Binding Services	46
Other Functions for the Kernel Extension and Device Driver Management Services	46

List of Kernel Extension and Device Driver Management Kernel Services	47
Locking Kernel Services	48
Lock Allocation and Other Services	48
Simple Locks	48
Complex Locks	49
Lockl Locks	50
Atomic Lock Operations	50
Atomic Operations	51
File Descriptor Management Services	51
Logical File System Kernel Services	51
Other Considerations	52
List of Logical File System Kernel Services	52
Memory Kernel Services	53
Memory Management Kernel Services	53
Memory Pinning Kernel Services	53
User Memory Access Kernel Services	54
Virtual Memory Management Kernel Services	54
Cross-Memory Kernel Services	55
Understanding Virtual Memory Manager Interfaces	56
Virtual Memory Objects	56
Addressing Data	57
Moving Data to or from a Virtual Memory Object	57
Data Flushing	57
Discarding Data	57
Protecting Data	58
Executable Data	58
Installing Pager Backends	58
Referenced Routines	58
Services that Support 64-bit Processes	59
Message Queue Kernel Services	60
Network Kernel Services	61
Address Family Domain and Network Interface Device Driver Kernel Services	61
Routing and Interface Address Kernel Services	62
Loopback Kernel Services	62
Protocol Kernel Services	62
Communications Device Handler Interface Kernel Services	63
Process and Exception Management Kernel Services	63
Creating Kernel Processes	63
Creating Kernel Threads	63
Kernel Structures Encapsulation	64
Registering Exception Handlers	64
Signal Management	64
Events Management	64
List of Process, Thread, and Exception Management Kernel Services	65
RAS Kernel Services	66
List of RAS Kernel Services	66
Security Kernel Services	67
Timer and Time-of-Day Kernel Services	67
Time-Of-Day Kernel Services	67
Fine Granularity Timer Kernel Services	67
Timer Kernel Services for Compatibility	68
Watchdog Timer Kernel Services	68
Using Fine Granularity Timer Services and Structures	68
Timer Services Data Structures	68
Coding the Timer Function	69
Using Multiprocessor-Safe Timer Services	69

Virtual File System (VFS) Kernel Services	69
---	----

Chapter 5. Asynchronous I/O Subsystem	71
Asynchronous I/O Overview	71
How do I know if I need to use AIO?	72
How many AIO Servers am I currently using?.	72
How many AIO servers do I need?	73
Prerequisites	73
Functions of Asynchronous I/O	73
Large File-Enabled Asynchronous I/O (AIX Version 4.2.1 or later)	73
Nonblocking I/O	74
Notification of I/O Completion.	74
Cancellation of I/O Requests	75
Asynchronous I/O Subroutines.	75
Order and Priority of Asynchronous I/O Calls	76
Subroutines Affected by Asynchronous I/O	76
Changing Attributes for Asynchronous I/O.	76
64-bit Enhancements	77

Chapter 6. Device Configuration Subsystem	79
Scope of Device Configuration Support	79
Device Configuration Subsystem Overview.	79
General Structure of the Device Configuration Subsystem.	80
High-Level Perspective	83
Device Method Level	84
Low-Level Perspective.	84
Device Configuration Database Overview	84
Basic Device Configuration Procedures Overview.	85
Device Configuration Manager Overview	85
Devices Graph	86
Configuration Rules	86
Invoking the Configuration Manager	87
Device Classes, Subclasses, and Types Overview	87
Writing a Device Method.	88
Invoking Methods	88
Example Methods	88
Understanding Device Methods Interfaces	89
Configuration Manager	89
Run-Time Configuration Commands	90
Understanding Device States	90
Adding an Unsupported Device to the System	92
Modifying the Predefined Database	92
Adding Device Methods	92
Adding a Device Driver	93
Using installp Procedures.	93
Understanding Device Dependencies and Child Devices.	93
Accessing Device Attributes	94
Modifying an Attribute Value	95
Device Dependent Structure (DDS) Overview	95
How the Change Method Updates the DDS	96
Guidelines for DDS Structure	96
Example of DDS.	97
List of Device Configuration Commands	97
List of Device Configuration Subroutines	98

Chapter 7. Communications I/O

Subsystem 99

User-Mode Interface to a Communications PDH . . . 99

Kernel-Mode Interface to a Communications PDH . . . 99

CDLI Device Drivers 100

Communications Physical Device Handler Model Overview 100

Use of mbuf Structures in the Communications PDH 101

Common Communications Status and Exception Codes 101

Status Blocks for Communications Device Handlers Overview 102

CIO_START_DONE 102

CIO_HALT_DONE 102

CIO_TX_DONE 103

CIO_NULL_BLK 103

CIO_LOST_STATUS 103

CIO_ASYNC_STATUS 103

MPQP Device Handler Interface Overview 104

Binary Synchronous Communication (BSC) with the MPQP Adapter 104

Description of the MPQP Card 105

Serial Optical Link Device Handler Overview 108

Special Files 108

Entry Points 108

Configuring the Serial Optical Link Device Driver 109

Physical and Logical Devices 109

Changeable Attributes of the Serial Optical Link Subsystem 110

Forum-Compliant ATM LAN Emulation Device Driver 110

Adding ATM LANE Clients 113

Configuration Parameters for the ATM LANE Device Driver 113

Device Driver Configuration and Unconfiguration 118

Device Driver Open 118

Device Driver Close 118

Data Transmission 118

Data Reception 119

Asynchronous Status 119

Device Control Operations 120

Tracing and Error Logging in the ATM LANE Device Driver 124

Adding an ATM MPOA Client 124

Configuration Parameters for ATM MPOA Client 126

Tracing and Error Logging in the ATM MPOA Client 126

Fiber Distributed Data Interface (FDDI) Device Driver 127

Configuration Parameters for FDDI Device Driver 127

FDDI Device Driver Configuration and Unconfiguration 128

Device Driver Open 128

Device Driver Close 128

Data Transmission 128

Data Reception 129

Reliability, Availability, and Serviceability for FDDI Device Driver 129

High-Performance (8fc8) Token-Ring Device Driver Configuration Parameters for Token-Ring Device 132

Driver 132

Device Driver Configuration and Unconfiguration 132

Device Driver Open 132

Device Driver Close 132

Data Transmission 133

Data Reception 133

Asynchronous Status 133

Device Control Operations 136

Trace Points and Error Log Templates for 8fc8 Token-Ring Device Driver 138

High-Performance (8fa2) Token-Ring Device Driver Configuration Parameters for 8fa2 Token-Ring 141

Device Driver 141

Device Driver Configuration and Unconfiguration 141

Device Driver Open 141

Device Driver Close 142

Data Transmission 142

Data Reception 142

Asynchronous Status 143

Device Control Operations 145

Trace Points and Error Log Templates for 8fa2 Token-Ring Device Driver 147

PCI Token-Ring High Performance (14101800) Device Driver 149

Configuration Parameters 150

Device Driver Configuration and Unconfiguration 151

Device Driver Open 151

Device Driver Close 151

Data Transmission 151

Data Reception 151

Asynchronous Status 152

Device Control Operations 153

Reliability, Availability, and Serviceability (RAS) 155

Ethernet Device Drivers 157

Configuration Parameters 159

Interface Entry Points 164

Asynchronous Status 166

Device Control Operations 168

Reliability, Availability, and Serviceability (RAS) 171

Chapter 8. Graphic Input Devices

Subsystem 179

open and close Subroutines 179

read and write Subroutines 179

ioctl Subroutines 179

Keyboard 179

Mouse 180

Tablet 180

GIO (Graphics I/O) Adapter 180

Dials 180

LPFK 180

Input Ring 181

Management of Multiple Keyboard Input Rings 181

Event Report Formats 181

Keyboard Service Vector	182
Special Keyboard Sequences	183

Chapter 9. Low Function Terminal Subsystem 185

Low Function Terminal Interface Functional Description	185
Configuration	185
Terminal Emulation	185
IOCTLS Needed for AIXwindow Support	186
Low Function Terminal to System Keyboard Interface	186
Low Function Terminal to Display Device Driver Interface	186
Low Function Terminal Device Driver Entry Points	186
Components Affected by the Low Function Terminal Interface	186
Configuration User Commands	186
Display Device Driver	187
Rendering Context Manager	187
Diagnostics	188
Accented Characters	188
List of Diacritics Supported by the HFT LFT Subsystem	188

Chapter 10. Logical Volume Subsystem 191

Direct Access Storage Devices (DASDs).	191
Physical Volumes	191
Physical Volume Implementation Limitations	192
Physical Volume Layout	192
Reserved Sectors on a Physical Volume	192
Sectors Reserved for the Logical Volume Manager (LVM)	193
Understanding the Logical Volume Device Driver	195
Data Structures	195
Top Half of LVDD	196
Bottom Half of the LVDD	196
Interface to Physical Disk Device Drivers	198
Understanding Logical Volumes and Bad Blocks	199
Relocating Bad Blocks	199
Detecting and Correcting Bad Blocks	199
Changing the mwcc_entries Variable	200
Prerequisite Tasks or Conditions	200
Procedure	200

Chapter 11. Printer Addition Management Subsystem 203

Printer Types Currently Supported	203
Printer Types Currently Unsupported	203
Adding a New Printer Type to Your System	203
Additional Steps for Adding a New Printer Type	203
Modifying Printer Attributes	204
Adding a Printer Definition	204
Adding a Printer Formatter to the Printer Backend	205
Understanding Embedded References in Printer Attribute Strings	205

Chapter 12. Small Computer System Interface Subsystem 207

SCSI Subsystem Overview	207
Responsibilities of the SCSI Adapter Device Driver	207
Responsibilities of the SCSI Device Driver	207
Communication between SCSI Devices	208
Understanding SCSI Asynchronous Event Handling	209
Defined Events and Recovery Actions	210
Asynchronous Event-Handling Routine	210
SCSI Error Recovery	211
SCSI Initiator-Mode Recovery When Not Command Tag Queuing	211
SCSI Initiator-Mode Recovery During Command Tag Queuing	212
Analyzing Returned Status	213
Target-Mode Error Recovery	214
A Typical Initiator-Mode SCSI Driver Transaction Sequence	214
Understanding SCSI Device Driver Internal Commands	215
Understanding the Execution of Initiator I/O Requests	215
Spanned (Consolidated) Commands	216
Fragmented Commands	216
Gathered Write Commands	217
SCSI Command Tag Queuing	218
Understanding the sc_buf Structure	218
Fields in the sc_buf Structure	218
Other SCSI Design Considerations	223
Responsibilities of the SCSI Device Driver	223
SCSI Options to the openx Subroutine	223
Using the SC_FORCED_OPEN Option	224
Using the SC_RETAIN_RESERVATION Option	224
Using the SC_DIAGNOSTIC Option	224
Using the SC_NO_RESERVE Option	225
Using the SC_SINGLE Option	225
Closing the SCSI Device	227
SCSI Error Processing	227
Device Driver and Adapter Device Driver Interfaces	227
Performing SCSI Dumps	228
SCSI Target-Mode Overview	229
Configuring and Using SCSI Target Mode	229
Managing Receive-Data Buffers	230
Understanding Target-Mode Data Pacing	230
Understanding the SCSI Target Mode Device Driver Receive Buffer Routine	231
Understanding the tm_buf Structure	232
Understanding the Execution of SCSI Target-Mode Requests	233
Required SCSI Adapter Device Driver ioctl Commands	235
Initiator-Mode ioctl Commands	235
Target-Mode ioctl Commands	237
Target- and Initiator-Mode ioctl Commands	239

Chapter 13. Fibre Channel Protocol for SCSI Subsystem 241

FCP Subsystem Overview	241
Responsibilities of the FCP Adapter Device Driver	241
Responsibilities of the FCP Device Driver	241
Communication between FCP Devices	242
Initiator-Mode Support	242
Understanding FCP Asynchronous Event Handling	242
Defined Events and Recovery Actions	243
Asynchronous Event-Handling Routine.	244
FCP Error Recovery	244
autosense data	245
NACA=1 error recovery.	245
FCP Initiator-Mode Recovery When Not Command Tag Queuing	245
FCP Initiator-Mode Recovery During Command Tag Queuing	246
Analyzing Returned Status	247
A Typical Initiator-Mode FCP Driver Transaction Sequence	248
Understanding FCP Device Driver Internal Commands	249
Understanding the Execution of Initiator I/O Requests	249
Spanned (Consolidated) Commands.	250
Fragmented Commands	250
FCP Command Tag Queuing	251
Understanding the scsi_buf Structure	251
Fields in the scsi_buf Structure	251
Other FCP Design Considerations	257
Responsibilities of the FCP Device Driver	257
FCP Options to the openx Subroutine	257
Using the SC_FORCED_OPEN Option	258
Using the SC_RETAIN_RESERVATION Option	258
Using the SC_DIAGNOSTIC Option.	258
Using the SC_NO_RESERVE Option.	259
Using the SC_SINGLE Option.	259
Closing the FCP Device	261
FCP Error Processing	261
Length of Data Transfer for FCP Commands	261
Device Driver and Adapter Device Driver Interfaces.	262
Performing FCP Dumps	262
Required FCP Adapter Device Driver ioctl Commands	263
Description	263
Initiator-Mode ioctl Commands	263
Initiator-Mode ioctl Command used by FCP Device Drivers	266

Chapter 14. FCP Device Drivers 269

Programming FCP Device Drivers	269
FCP Device Driver Overview	269
FCP Adapter Device Driver Overview	269
FCP Adapter/Device Interface.	270
scsi_buf Structure	270
Adapter/Device Driver Intercommunication	275
FCP Adapter Device Driver Routines	276
config	276
open	276
close	276
openx	276

strategy	277
ioctl	277
start	277
interrupt	277
FCP Adapter ioctl Operations	277
IOCINFO.	277
SCIOLSTART	278
SCIOLSTOP	278
SCIOLEVENT	279
SCIOLINQU.	279
SCIOLSTUNIT	280
SCIOLTUR	281
SCIOLREAD	281
SCIOLRESET	282
SCIOLHALT.	282
SCIOLCMD	283

Chapter 15. Integrated Device Electronics (IDE) Subsystem. 285

Responsibilities of the IDE Adapter Device Driver	285
Responsibilities of the IDE Device Driver	285
Communication Between IDE Device Drivers and IDE Adapter Device Drivers	286
IDE Error Recovery	286
Analyzing Returned Status	286
A Typical IDE Driver Transaction Sequence	287
IDE Device Driver Internal Commands.	288
Execution of I/O Requests	288
Spanned (Consolidated) Commands.	289
Fragmented Commands	290
Gathered Write Commands.	290
ataide_buf Structure	291
Fields in the ataide_buf Structure.	291
Other IDE Design Considerations	293
IDE Device Driver Tasks.	293
Closing the IDE Device	294
IDE Error Processing	294
Device Driver and Adapter Device Driver Interfaces.	294
Performing IDE Dumps	294
Required IDE Adapter Device Driver ioctl Commands	295
ioctl Commands	295

Chapter 16. Serial Direct Access Storage Device Subsystem 299

DASD Device Block Level Description	299
---	-----

Chapter 17. Debugging Tools 301

System Dump	301
Initiating a System Dump	301
Including Device Driver Information in a System Dump	302
Formatting a System Dump	304
The crash Command	305
Addresses in crash	306
Command-line Editing	306
Output Redirection	306
crash Subcommands	307
Low Level Kernel Debugger (LLDB).	332

LLDB Kernel Debug Program	332	ppd Command for the LLDB Kernel Debug Program	355
Loading and Starting the LLDB Kernel Debug Program	333	proc Command for the LLDB Kernel Debug Program	355
Using a Terminal with the LLDB Kernel Debug Program	333	queue Command for the LLDB Kernel Debug Program	356
Entering the LLDB Kernel Debug Program	334	quit Command for the LLDB Kernel Debug Program	357
Debugging Multiprocessor Systems	334	reason Command for the LLDB Kernel Debug Program	357
LLDB Kernel Debug Program Concepts	335	reboot Command for the LLDB Kernel Debug Program	358
LLDB Kernel Debug Program Commands	338	reset Command for the LLDB Kernel Debug Program	358
LLDB Kernel Debug Program Commands grouped in Alphabetical Order	338	screen Command for the LLDB Kernel Debug Program	359
LLDB Kernel Debug Program Commands grouped by Task Category	340	segst64 Command for the LLDB Kernel Debug Program	360
Descriptions of the LLDB Kernel Debug Program Commands	341	set Command for the LLDB Kernel Debug Program	361
alter Command for the LLDB Kernel Debug Program	341	sregs Command for the LLDB Kernel Debug Program	362
back Command for the LLDB Kernel Debug Program	342	sr64 Command for the LLDB Kernel Debug Program	362
break Command for the LLDB Kernel Debug Program	342	st Command for the LLDB Kernel Debug Program	363
breaks Command for the LLDB Kernel Debug Program	343	stack Command for the LLDB Kernel Debug Program	364
buckets Command for the LLDB Kernel Debug Program	344	stc Command for the LLDB Kernel Debug Program	364
clear Command for the LLDB Kernel Debug Program	344	step Command for the LLDB Kernel Debug Program	365
cpu Command for the LLDB Kernel Debug Program	345	sth Command for the LLDB Kernel Debug Program	365
display Command for the LLDB Kernel Debug Program	345	stream Command for the LLDB Kernel Debug Program	366
dmodsw Command for the LLDB Kernel Debug Program	346	swap Command for the LLDB Kernel Debug Program	367
drivers Command for the LLDB Kernel Debug Program	347	sysinfo Command for the LLDB Kernel Debug Program	368
find Command for the LLDB Kernel Debug Program	348	thread Command for the LLDB Kernel Debug Program	368
float Command for the LLDB Kernel Debug Program	349	trace Command for the LLDB Kernel Debug Program	370
fmodsw Command for the LLDB Kernel Debug Program	349	trb Command for the LLDB Kernel Debug Program	371
fs Command for the LLDB Kernel Debug Program	350	tty Command for the LLDB Kernel Debug Program	371
go Command for the LLDB Kernel Debug Program	350	un Command for the LLDB Kernel Debug Program	372
help Command for the LLDB Kernel Debug Program	351	user Command for the LLDB Kernel Debug Program	372
loop Command for the LLDB Kernel Debug Program	351	user64 Command for the LLDB Kernel Debug Program	373
map Command for the LLDB Kernel Debug Program	352	uthread Command for the LLDB Kernel Debug Program	373
mblk Command for the LLDB Kernel Debug Program	353	vars Command for the LLDB Kernel Debug Program	375
mst64 Command for the LLDB Kernel Debug Program	353	vmm Command for the LLDB Kernel Debug Program	375
netdata Command for the LLDB Kernel Debug Program	354		
next Command for the LLDB Kernel Debug Program	354		
origin Command for the LLDB Kernel Debug Program	354		

watch Command for the LLDB Kernel Debug Program	376	ddvb, dddvh, dddvw, dddvd, dddpd, dddph, and dddpw Subcommands	430
xlate Command for the LLDB Kernel Debug Program	376	find and findp Subcommands	430
Maps and Listings as Tools for the LLDB Kernel Debug Program	377	ext and extp Subcommands	431
Compiler Listing	377	Modify Memory Subcommands for the KDB Kernel Debugger and kdb Command	432
Map File	378	m, mw, md, mp, mpw, and mpd Subcommands	432
Using the LLDB Kernel Debug Program	381	mr Subcommand	434
Setting Breakpoints	381	mdvb, mdvh, mdvw, mdvd, mdpb, mdph, mdpw, mdpd Subcommands	434
Viewing and Modifying Global Data	384	Namelist/Symbol Subcommands for the KDB Kernel Debugger and kdb Command	436
Displaying Registers on a Micro Channel Adapter	386	nm and ts Subcommands	436
Stack Trace	387	ns Subcommand	436
Error Messages for the LLDB Kernel Debug Program	388	Watch Break Points Subcommands for the KDB Kernel Debugger and kdb Command	437
KDB Kernel Debugger and Command	390	wr, ww, wrw, cw, lwr, lww, lwrw, and lcw Subcommands	437
KDB Kernel Debugger and kdb Command	390	Miscellaneous Subcommands for the KDB Kernel Debugger and kdb Command	438
The kdb Command	390	time and debug Subcommands	438
KDB Kernel Debugger	391	Conditional Subcommands for the KDB Kernel Debugger and kdb Command	439
Loading and Starting the KDB Kernel Debugger Using a Terminal with the KDB Kernel Debugger.	392	test Subcommand	439
Entering the KDB Kernel Debugger	392	Calculator Converter Subcommands for the KDB Kernel Debugger and kdb Command	439
Debugging Multiprocessor Systems	393	hcal and dcal Subcommands	439
Kernel Debug Program Concepts	393	Machine Status Subcommands for the KDB Kernel Debugger and kdb Command	439
Subcommands for the KDB Kernel Debugger and kdb Command	394	status Subcommand	439
Introduction to Subcommands.	394	switch Subcommand	441
KDB Kernel Debug Program Subcommands grouped in Alphabetical Order	396	Kernel Extension Loader Subcommands for the KDB Kernel Debugger and kdb Command	442
KDB Kernel Debug Subcommands grouped by Task Category	402	lke, stbl, and rmst Subcommands.	442
Basic Subcommands for the KDB Kernel Debugger and kdb Command	410	export table Subcommand	445
h Subcommand.	410	Address Translation Subcommands for the KDB Kernel Debugger and kdb Command	446
his Subcommand	411	tr and tv Subcommands	446
e Subcommand.	412	Process Subcommands for the KDB Kernel Debugger and kdb Command	447
set Subcommand	412	ppda Subcommand	447
f Subcommand	414	intr Subcommand	448
ctx Subcommand	417	mst Subcommand	449
cdt Subcommand	419	proc Subcommand	450
Trace Subcommands for the KDB Kernel Debugger and kdb Command	420	thread Subcommand	453
bt Subcommand	420	ttid and tpid Subcommands	456
ct and cat Subcommands	420	runq, lockq, and sleepq Subcommands	457
bt script Subcommand	421	user Subcommand.	457
bt [cond] Subcommand	421	LVM Subcommands for the KDB Kernel Debugger and kdb Command	459
Breakpoints/Steps Subcommands for the KDB Kernel Debugger and kdb Command	421	pbuf Subcommand	459
b Subcommand.	421	volgrp Subcommand	459
lb Subcommand	422	pvol Subcommand	461
c, lc, and ca Subcommands	423	lvvol Subcommand	461
r and gt Subcommands	424	SCSI Subcommands for the KDB Kernel Debugger and kdb Command	462
n s, S, and B Subcommands	425	ascsi Subcommand	462
Dumps/Display/Decode Subcommands for the KDB Kernel Debugger and kdb Command	426	vscsi Subcommand	464
d, dw, dd, dp, dpw, dpd Subcommands	426	scdisk Subcommand	468
dc and dpc Subcommands	427	Memory Allocator Subcommands for the KDB Kernel Debugger and kdb Command	471
dr Subcommand	428		

heap Subcommand	471	lockanch Subcommand	513
xm Subcommand	473	lockhash Subcommand	514
bucket Subcommand	475	lockword Subcommand	515
kmstats Subcommands	476	vmdmap Subcommand	518
File System Subcommands for the KDB Kernel		vmlocks Subcommand	519
Debugger and kdb Command	477	SMP Subcommands for the KDB Kernel Debugger	
buffer Subcommand	477	and kdb Command	520
hbuffer Subcommand	478	start and stop Subcommands	520
fbuffer Subcommand	478	cpu Subcommand	521
gnode Subcommand	478	bat/Block Address Translation Subcommands for	
gfs Subcommand	479	the KDB Kernel Debugger and kdb Command	522
file Subcommand	479	dbat Subcommand	522
inode Subcommand	480	ibat Subcommand	522
hinode Subcommand	481	mdbat Subcommand	523
icache Subcommand	482	mibat Subcommand	524
rnode Subcommand	483	btac/BRAT Subcommands for the KDB Kernel	
cku Subcommand	483	Debugger and kdb Command	524
vnode Subcommand	484	btac, cbtac, lbtac, lcbtac Subcommands	524
mount Subcommand	484	machdep Subcommands for the KDB Kernel	
specnode Subcommand	485	Debugger and kdb Command	526
devnode Subcommand	486	reboot Subcommand	526
fifonode Subcommand	487	Using the KDB Kernel Debug Program	526
hnode Subcommand	488	Example Files	526
System Table Subcommands for the KDB Kernel		Generating Maps and Listings	527
Debugger and kdb Command	488	Compiler Listing	527
var Subcommand	488	Map File	528
devsw Subcommand	489	Setting Breakpoints	530
timer Subcommand	490	Viewing and Modifying Global Data	534
slk and clk Subcommands	491	Stack Trace	538
iplcb Subcommand	491	demo.c Example File	542
trace Subcommand	492	demokext.c Example File	544
Net Subcommands for the KDB Kernel Debugger		demo.h Example File	546
and kdb Command	494	demokext.exp Example File	546
ifnet Subcommand	494	comp_link Example File	546
tcb Subcommand	495	Error Logging	546
udb Subcommand	495	Precoding Steps to Consider	547
sock Subcommand	496	Coding Steps	548
tcpcb Subcommand	496	Writing to the /dev/error Special File	553
mbuf Subcommand	497	Debug and Performance Tracing	554
VMM Subcommands for the KDB Kernel Debugger		Introduction	554
and kdb Command	497	Using the trace Facility	556
vmker Subcommand	497	Controlling trace	559
rmap Subcommand	498	Producing a trace Report	561
pfhdata Subcommand	499	Defining trace Events	564
vmstat Subcommand	500	Usage Hints	577
vmaddr Subcommand	501	SMIT Trace Hook Groups	579
pdt Subcommand	501	Memory Overlay Detection System (MODS)	579
scb Subcommand	502	AIX Kernel Memory Overlay Detection System	
pft Subcommand	503	(MODS)	579
pte Subcommand	506	Appendix A. Alphabetical List of	
pta Subcommand	506	Kernel Services 583	
ste Subcommand	507	Kernel Services Available in Process and Interrupt	
sr64 Subcommand	508	Environments	583
segst64 Subcommand	508	Kernel Services Available in the Process	
apt Subcommand	509	Environment Only	588
vmwait Subcommand	509	Index 593	
ames Subcommand	510		
zproc Subcommand	511		
vmlog Subcommand	512		
vrlld Subcommand	512		
ipc Subcommand	512		

**Readers' Comments — We'd Like to
Hear from You 599**

About This Book

This book provides information on the kernel programming environment, and about writing system call, kernel service, and virtual file system kernel extensions. Conceptual information on existing kernel subsystems is also provided.

More detailed information on existing kernel services and interface requirements for kernel extensions can be found in *AIX Version 4.3 Technical Reference: Kernel and Subsystems Volume 1*, Order Number SC23-4163, and *AIX Version 4.3 Technical Reference: Kernel and Subsystems Volume 2*, Order Number SC23-4164.

Note: The information in this book can also be found on the *AIX Version 4.3 Extended Documentation CD*. This online documentation is designed for use with an HTML version 3.2 compatible web browser.

Who Should Use This Book

This book is intended for system programmers who are knowledgeable in operating system concepts and kernel programming and want to extend the kernel.

How to Use This Book

This book provides two types of information: (1) an overview of the kernel programming environment and information a programmer needs to write kernel extensions, and (2) information about existing kernel subsystems.

Overview of Contents

This book contains the following chapters and appendixes:

- Chapter 1, "Kernel Environment", provides an overview of programming in the kernel environment, including kernel extension binding, kernel processes, signal handling, and exception handling.
- Chapter 2, "System Calls", contrasts a user function and a system call, and discusses aspects of system call execution.
- Chapter 3, "Virtual File Systems", discusses the components of a virtual file system, and the steps needed to configure it.
- Chapter 4, "Kernel Services", discusses the various types of kernel services.
- Chapter 5, "Asynchronous I/O Subsystem", describes asynchronous I/O.
- Chapter 6, "Device Configuration Subsystem", provides an overview of the configuration process, the routines and databases involved, and the requirements for configuring new devices.
- Chapter 7, "Communications I/O Subsystem", contains some information common to all communications device drivers and some information about specific communications device drivers.
- Chapter 8, "Graphic Input Devices Subsystem", describes the programming interface of the graphic input device driver.
- Chapter 9, "Low Function Terminal (LFT) Subsystem", discusses the component structure of the high function terminal and the concept of the virtual terminal low function terminal.
- Chapter 10, "Logical Volume Subsystem", includes information on physical volumes, the logical volume device driver, and logical volumes and bad blocks.

- Chapter 11, "Printer Addition Management Subsystem", describes the steps involved in adding a printer to the system.
- Chapter 12, "Small Computer System Interface (SCSI) Subsystem", discusses SCSI subsystem architecture and aspects of writing SCSI device drivers.
- Chapter 13, "Fibre Channel Protocol for SCSI Subsystem", describes the interface between a Fibre Channel Protocol (FCP) for SCSI device driver and a FCP adapter device driver.
- Chapter 14, "FCP Device Drivers", includes information on programming FCP device drivers.
- Chapter 15, "Integrated Device Electronics (IDE) Subsystem", discusses IDE subsystem architecture and aspects of writing IDE device drivers.
- Chapter 16, "Serial Direct Access Storage Device Subsystem", includes information on using serial direct access storage devices.
- Chapter 17, "Debugging Tools", includes information on debugging device drivers.
- Appendix A, "Alphabetical List of Kernel Services", lists and summarizes the function of the kernel services. The list is divided based on the execution environment from which each service can be called.

Highlighting

The following highlighting conventions are used in this book:

Bold	Identifies commands, subroutines, keywords, files, structures, directories, and other items whose names are predefined by the system. Also identifies graphical objects such as buttons, labels, and icons that the user selects.
<i>Italics</i>	Identifies parameters whose actual names or values are to be supplied by the user.
Monospace	Identifies examples of specific data values, examples of text similar to what you might see displayed, examples of portions of program code similar to what you might write as a programmer, messages from the system, or information you should actually type.

ISO 9000

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

AIX 32-Bit Support for the X/Open UNIX95 Specification

Beginning with AIX Version 4.2, the operating system is designed to support the X/Open UNIX95 Specification for portability of UNIX-based operating systems. Many new interfaces, and some current ones, have been added or enhanced to meet this specification. Beginning with Version 4.2, AIX is even more open and portable for applications.

At the same time, compatibility with previous AIX releases is preserved. This is accomplished by the creation of a new environment variable, which can be used to set the system environment on a per-system, per-user, or per-process basis.

To determine the proper way to develop a UNIX95-portable application, you may need to refer to the X/Open UNIX95 Specification, which can be obtained on a CD-ROM by ordering the printed copy of *AIX Version 4.3 Commands Reference*, order number SBOF-1877, or by ordering *Go Solo: How to Implement and Go Solo with the Single Unix Specification*, order number SR28-5705, a book which includes the X/Open UNIX95 Specification on a CD-ROM.

AIX 32-Bit and 64-Bit Support for the UNIX98 Specification

Beginning with AIX Version 4.3, the operating system is designed to support the X/Open UNIX98 Specification for portability of UNIX-based operating systems. Many new interfaces, and some current ones, have been added or enhanced to meet this specification. Making AIX Version 4.3 even more open and portable for applications.

At the same time, compatibility with previous AIX releases is preserved. This is accomplished by the creation of a new environment variable, which can be used to set the system environment on a per-system, per-user, or per-process basis.

To determine the proper way to develop a UNIX98-portable application, you may need to refer to the X/Open UNIX98 Specification, which can be obtained on a CD-ROM by ordering the printed copy of *AIX Version 4.3 Commands Reference*, order number SBOF-1877, or by ordering *Go Solo: How to Implement and Go Solo with the Single Unix Specification*, order number SR28-5705, a book which includes the X/Open UNIX98 Specification on a CD-ROM.

Related Publications

The following books contain additional information on kernel extension programming and the existing kernel subsystems:

- *AIX Ethernet Local Broadcast/6000*, Order Number GC23-2439
- *Ethernet HUB Installation*, Order Number GA27-4024
- *Ethernet LAN Adapter Family*, Order Number G221-3457
- *AIX Version 4.3 Guide to Printers and Printing*, Order Number SC23-4130.
- *AIX Version 4 Keyboard Technical Reference*, Order Number SC23-2631.
- *AIX Version 4.3 Problem Solving Guide and Reference*, Order Number SC23-4123.
- *AIX Version 4.3 System Management Guide: Operating System and Devices*, Order Number SC23-4126.
- *AIX Version 4.3 Technical Reference: Kernel and Subsystems Volume 1*, Order Number SC23-4163.
- *AIX Version 4.3 Technical Reference: Kernel and Subsystems Volume 2*, Order Number SC23-4164
- *Token-Ring Network Architecture Reference*, Order Number SC30-3374

Ordering Publications

You can order publications from your sales representative or from your point of sale.

To order additional copies of this book, use order number SC23-4125.

Use *AIX and Related Products Documentation Overview* for information on related publications and how to obtain them.

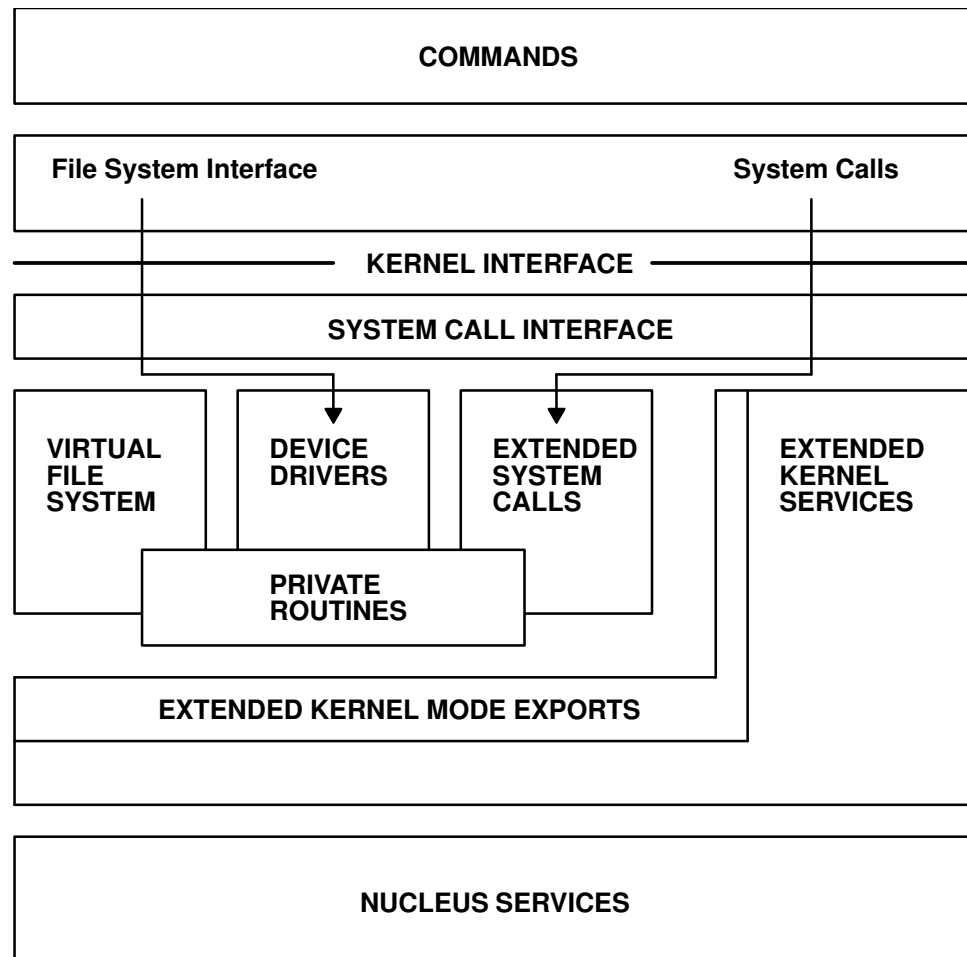
Chapter 1. Kernel Environment

The kernel is dynamically extendable and can be expanded by adding routines that belong to any of the following functional classes:

- System calls
- Virtual file systems
- Kernel Extension and Device Driver Management Kernel Services
- Device Drivers

These kernel extensions can be added at system boot or while the system is in operation.

The Types of Kernel Extensions diagram illustrates the addition of extensions to the kernel environment.



Types of Kernel Extensions

The following kernel-environment programming information is provided to assist you in programming kernel extensions:

- Understanding Kernel Extension Binding
- Understanding Execution Environments

- Understanding Kernel Threads
- Using Kernel Processes
- Accessing User-Mode Data While in Kernel Mode
- Understanding Locking
- Understanding Exception Handling
- 64-bit Kernel Extension Development

A process executing in user mode can customize the kernel by using the **sysconfig** subroutine, if the process has appropriate privilege. In this way, a user-mode process can load, unload, initialize, or terminate kernel routines. Kernel configuration can also be altered by changing tuneable system parameters.

Kernel extensions can also customize the kernel by using kernel services to load, unload, initialize, and terminate dynamically loaded kernel routines; to create and initialize kernel processes; and to define interrupt handlers. Binding of kernel extensions can be performed at link-edit, load, or run time.

Note: Private kernel routines (or kernel services) execute in a privileged protection domain and can affect the operation and integrity of the whole system. See “Kernel Protection Domain” on page 26 for more information.

Understanding Kernel Extension Binding

The following information is provided to assist you in understanding kernel extension binding.

- “Base Kernel Services - the /unix Name Space”
- “Using System Calls with Kernel Extensions” on page 3
- “Using Private Routines” on page 4
- “Using Libraries” on page 5

Base Kernel Services - the /unix Name Space

The kernel provides a set of base kernel services to be used by kernel extensions. (See “Chapter 4. Kernel Services” on page 39.) These services, which are described in the services documentation, are made available to a kernel extension by specifying the **kernex.exp** kernel export file as an import file during the link-edit of the extension. The link-edit operation is performed by using the **ld** command.

The kernel provides a set of base kernel services to be used by kernel extensions. These services, which are described in the services documentation, are made available to a kernel extension by specifying the **kernex.exp** kernel export file as an import file during the link-edit of the extension. The link-edit operation is performed by using the **ld** command.

A kernel extension provides additional kernel services and system calls by supplying an export file when it is link-edited. This export file specifies the symbols to be added to the **/unix** name space, which is the global kernel name space. Symbols that name system calls to be exported must specify one of the **SYSCALL**, **SYSCALL32**, **SYSCALL64**, or **SYSCALL3264** keywords next to the symbol in the export file.

The kernel extension export file must also have **#!/unix** as its first entry. The export file can then be used by other extensions as an import file. The **#!/unix** as the first

entry in an import file specifies that the imported symbols are to come from the `/unix` name space. This entry is ignored when used in an export file. The same file can be used both as the export file for the kernel extension providing the symbols and as the import file for another extension importing one or more of the symbols.

When a new kernel extension is loaded by the `sysconfig` subroutine, any symbols defined in the extension export file at link-edit time are added to the `/unix` kernel name space. The loader can also load additional object files into the kernel to resolve symbols referenced by the new extension. Because these exported symbols are only used to resolve references required during loading of the new extension, these additional object files will not have their own exported symbols added to the name space.

In other words, the kernel name space cannot be expanded without the explicit loading of a kernel object file specifying one or more exported symbols. The symbols added to the kernel name space are available to any subsequently loaded kernel object file as an imported symbol.

An object file explicitly loaded into the kernel exporting symbols into the kernel name space is shared by all kernel extensions. Normally, only one copy of the object file exists in the kernel.

Using System Calls with Kernel Extensions

A restricted set of 32-bit system calls can be used by kernel extensions. A kernel process can use a larger set of system calls than a user process in kernel mode. “System Calls Available to Kernel Extensions” on page 31 specifies which system calls can be used by either type of process. User-mode processes in kernel mode can only use system calls that have all parameters passed by value. Kernel routines running under user-mode processes cannot directly use a system call having parameters passed by reference.

The second restriction is imposed because, when they access a caller’s data, system calls with parameters passed by reference access storage across a protection domain. The cross-domain memory services performing these cross-memory operations support kernel processes as if they, too, accessed storage across a protection domain. However, these services have no way to determine the caller is in the same protection domain when the caller is a user-mode process in kernel mode.

Note: System calls must not be used by kernel extensions executing in the interrupt handler environment.

Kernel extensions can bind to a restricted set of base system calls. Binding is done by specifying the `syscalls.exp` system call export file as an import file when the kernel extension is link-edited. When loading object files into the kernel, the loader needs no protection domain switch to access system calls from the kernel. It binds the system call imports to the function descriptor that provides direct access to the system call. For user-mode programs, the loader binds system call references to a set of function descriptors invoking the system call handler to switch protection domains.

Loading System Calls and Kernel Services

Kernel extensions that provide new system calls or kernel services normally place only a single copy of the routine and its static data in the kernel. When this is the

case, use **SYS_SINGLELOAD sysconfig** operation to load the kernel extension. Because it only loads a new copy if one does not already exist in the kernel, this operation ensures that only a single copy is loaded. For this type of kernel extension, an updated version of the object file is loaded into the kernel only when the current copy has no users and has been unloaded.

If a kernel extension can support multiple versions of itself (particularly its data), the **SYS_KLOAD sysconfig** operation can be used. This operation loads a new copy of the object file even when one or more copies are already loaded. When this operation is used, currently loaded routines bound to the old copy of the object file continue to use the old copy. Any new routines (loaded after the new copy was loaded) are bound to the most recently loaded copy of the kernel extension.

Unloading System Calls and Kernel Services

Kernel extensions that provide new system calls or kernel services can also be unloaded. For each object file loaded, the loader maintains a usage count and a load count. The usage count indicates how many other object files have referenced some exported symbol provided by the kernel extension. The load count indicates how many explicit load requests have been made for each object file.

When an explicit unload of a kernel extension is requested, the load count is decremented. If the load count and the usage count are both equal to 0, the object file is unloaded. However if either the load count or usage count is not equal to 0, the object file is not unloaded. When programs end, the usage counts for kernel extensions that the programs referenced are adjusted. However, no unload of these kernel extensions is performed when the program ends, even if the load and usage counts become 0.

As a result, even though its load count has been decremented to 0 (due to unload requests) and its usage count has reached 0 (because of program terminations), a kernel extension can remain loaded. In this case, the kernel extension's exported symbols are still available for load-time binding unless another unload request for any object file is received. If an explicit unload request (for any program, shared library, or kernel extension) is received, the loader unloads all object files that have both load and usage counts of 0.

The **slibclean** command, which unloads all object files with load and use counts of 0 (zero), can be used to remove object files that are no longer used from both the shared library region and the kernel. Periodically invoking this command reduces the effects of memory fragmentation in the shared library and kernel text regions by removing object files that are no longer required.

Using Private Routines

The other discussions of kernel extension binding have been concerned with importing and exporting symbols *from* and *to* the **/unix** global kernel name space. These symbols are global in the kernel and can be referenced by any routine in the kernel. See "Base Kernel Services - the /unix Name Space" on page 2 for more information.

Kernel extensions can also consist of several separately link-edited object files that are bound at load time. This is particularly useful for device drivers, where one object file contains the top (pageable) half of the driver and a second object file

contains the bottom (pinned) half of the driver. Load-time binding is useful where several kernel extensions use common routines provided in a separate object file.

In both cases, the symbols exported by the private object files should not be added to the global kernel name space. If it is to have certain symbols exported to the global kernel name space and use other symbols only to resolve references to other private object files, the kernel extension should be divided into separately link-edited object files. (One object file would contain the symbols to be exported to the kernel name space, while the other would contain the exported symbols that are considered private.)

For object files that reference each other's symbols, each file should use the other's export file as its own import file during link-edit. The export file for the object file providing the services should specify `#!/path/file` as the first entry in the export file, where *path* specifies the directory path to the object *file*. This provides the exported symbols at load time. This entry is ignored when used as an export file. When used as an import file, however, the entry tells the loader where to find the object file that resolves the imported symbols at load time.

The object file that exports symbols to the kernel name space must specify `#!/unix` as the first entry in its export file. This allows the export file to be used as an import file by other kernel extensions. The object file containing the symbols to be exported to the kernel name space must be the one explicitly loaded into the kernel with the `sysconfig` subroutine. The loader then loads other private object files, as necessary, to resolve imported symbols required for the load.

When, during the same explicit load request, the loader encounters an imported symbol that is resolved by an already loaded object file, the loader does not load a new copy. Instead, it resolves the symbol to the copy of the already loaded object file. This allows for cross-resolving symbols between two or more object files loaded as a result of the same explicit load request.

Note: The loader hashes the path and file name of the object file to determine whether the file has already been loaded during this explicit load request. Another copy of the object file can be loaded if differing path names are used for the same object file and the two names do not hash to the same value.

Object files loaded automatically due to symbol resolution do not have their own exported symbols added to the kernel name space. These symbols remain private to the two or more object files loaded with an explicit load request. In this way, the kernel allows object files to have cross-dependent symbol references, and the loader will correctly resolve them.

Note however that when two separate explicit load requests have private symbols resolved by the same object file, two copies of that object file are loaded into the kernel. Each explicit load resolves its symbols to its own private copy of the object file. The private object files can also be combined into libraries with the `ar` (archive) command.

Using Libraries

A library is a collection of previously link-edited object files or import files and is created by using the `ar` (archive) command. Each object file or import file within the archive (library) is referred to as a member. Program management allows a member (or object file) to be designated as shared when it is link-edited. Libraries with or without shared objects can be created and used by kernel extensions.

However, due to the different programming requirements in the kernel, library services provided for user-mode applications generally should not be used by kernel extensions.

When it resolves a symbol to a library member (or object file) not designated as shared, the linkage editor (**ld** command) binds the required object file into the output object file so that the references will resolve. However, when symbols are resolved to a library member (or object file) designated as shared, the shared object file is not included in the output object file. Instead, the linkage editor adds information to the loader section of the output object file. The loader uses this information at load time to find the location of the shared object file that resolves the symbol.

When these shared object files (normally in libraries) are referenced by user-mode programs, the loader checks the shared library region to determine if the object file is in the shared library region. If it is, the references are resolved to the object file in the shared library region. If the object file has not already been loaded, the loader will load it into the shared library region if the file permissions allow it. In this way, common or shared object files used by user-mode applications can be shared by all user-mode programs in the system.

Unlike user mode, the kernel does not provide a shared library region. Therefore, when a kernel extension that refers to a shared object file is loaded, the loader loads a new copy of the shared object file into the kernel to be used to resolve all references to the object file during the explicit kernel extension load request. However, within the same explicit load request, all references to the *same* object file are resolved to the single copy of the object loaded for the current load request.

The operating system provides the following two libraries that can be used by kernel extensions:

- “libcsys Library”
- “libsys Library” on page 7

libcsys Library

The **libcsys** library is a subset of subroutines found in the user-mode **libc** library that can be used by kernel extensions and consists of the following subroutines:

- **atoi**
- **bcmp**
- **bcopy**
- **bzero**
- **memccpy**
- **memchr**
- **memcmp**
- **memcpy**
- **memmove**
- **memset**
- **ovbcopy**
- **strcat**
- **strchr**
- **strcmp**
- **strcpy**

- **strcspn**
- **strlen**
- **strncat**
- **strncmp**
- **strncpy**
- **strpbrk**
- **strrchr**
- **strspn**
- **strstr**
- **strtok**

Note: In addition to these explicit subroutines, some 64-bit math operators are implemented in **libc** and **libcsys.a**. Consequently, a kernel extension which manipulates 64-bit objects might need to bind with **libcsys.a**. In particular, **struct uio** contains a 64-bit offset.

The **memccpy**, **memcmp**, **memcpy**, and **memmove** memory subroutines are low-level subroutines that the **bcmp**, **bcopy** and **ovbcopy** subroutines use and can be called directly when path length is critical. These subroutines are defined in the **libc** library. The subroutines can be bound to the kernel export by specifying **libcsys.a** as a library when link-editing the kernel extension.

libsys Library

The **libsys** library provides the following set of kernel services:

- **d_align**
- **d_roundup**
- **timeout**
- **timeoutcf**
- **untimeout**

These kernel services, used *by* the extension, must be bound *to* the kernel extension. The kernel services are described as **libsys** services in their respective descriptions.

These services can be bound to the kernel extension by specifying **libsys.a** as an import library when link-editing kernel extension.

Note: The string routines implemented in **libcsys.a** contain processor specific code to enhance performance. These routines access the `"_system_configuration"` structure to determine the processor type. If a kernel extension uses the string routines in **libcsys.a**, then a definition for the `"_system_configuration"` structure must be provided. One way to accomplish this is by specifying `-BI:/lib/syscalls.exp` on the link-edit command line.

Understanding Execution Environments

There are two major environments under which a kernel extension can run:

- "Process Environment" on page 8
- "Interrupt Environment" on page 8

A kernel extension runs in the *process environment* when invoked either by a user process in kernel mode or by a kernel process. A kernel extension is executing in the *interrupt environment* when invoked as part of an interrupt handler (see “Understanding Interrupts” on page 44).

A kernel extension can determine in which environment it is called to run by calling the **getpid** or **thread_self** kernel service. These services respectively return the process or thread identifier of the current process or thread, or a value of -1 if called in the interrupt environment. Some kernel services can be called in both environments, while others can only be called in the process environment.

Note: No floating-point functions can be used in the kernel.

Process Environment

A routine runs in the process environment when it is called by a user-mode process or by a kernel process (see “Using Kernel Processes” on page 11). Routines running in the process environment are executed at an interrupt priority of INTBASE (the least favored priority). A kernel extension running in this environment can cause page faults by accessing pageable code or data. It can also be replaced by another process of equal or higher process priority.

A routine running in the process environment can sleep or be interrupted by routines executing in the interrupt environment. A kernel routine that runs on behalf of a user-mode process can only invoke system calls that have no parameters passed by reference. A kernel process, however, can use all system calls listed in the “System Calls Available to Kernel Extensions” on page 31 if necessary.

Interrupt Environment

A routine runs in the interrupt environment when called on behalf of an interrupt handler. A kernel routine executing in this environment cannot request data that has been paged out of memory and therefore cannot cause page faults by accessing pageable code or data. In addition, the kernel routine has a stack of limited size, is not subject to replacement by another process, and cannot perform any function that would cause it to sleep.

A routine in this environment is only interruptible either by interrupts that have priority more favored than the current priority or by exceptions. These routines cannot use system calls and can use only kernel services available in both the process and interrupt environments.

A process in kernel mode can also put *itself* into an environment similar to the interrupt environment. This action, occurring when the interrupt priority is changed to a priority more favored than INTBASE, can be accomplished by calling the **i_disabledisable_lock** kernel service. A kernel-mode process is sometimes required to do this to serialize access to a resource shared by a routine executing in the interrupt environment. When this is the case, the process operates under most of the same restrictions as a routine executing in the interrupt environment. However, the **e_sleep**, **e_wait**, **e_sleepl**, **et_wait**, **lockl**, and **unlockl** process can sleep, wait, and use locking kernel services if the event word or lock word is pinned.

Note: Locks should only be used when serializing access with respect to other processes. They are not adequate when attempting to serialize access to a resource accessed by a routine executing in the interrupt environment.

Routines executed in this environment can adversely affect system real-time performance and are therefore limited to a specific maximum path length. Guidelines for the maximum path length are determined by the interrupt priority at which the routines are executed. "Understanding Interrupts" on page 44 provides more information.

Understanding Kernel Threads

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

Up to version 3 of AIX, there was no difference between a process and a thread; each process contained a single thread. In AIX Version 4, one process can have multiple threads, with each thread executing different code concurrently, while sharing data and synchronizing much more easily than cooperating processes. Threads require fewer system resources than processes, and can start more quickly.

Although threads are the schedulable entity, they exist in the context of their process. The following list indicates what is managed at process level and shared among all threads within a process:

- Address space
- System resources, like files or terminals
- Signal list of actions.

The process remains the swappable entity. Only a few resources are managed at thread level, as indicated in the following list:

- State
- Stack
- Signal masks.

Kernel Threads, Kernel Only Threads, and User Threads

In AIX there are three kinds of threads:

- Kernel threads
- Kernel-only threads
- User threads.

A *kernel thread* is a kernel entity, like processes and interrupt handlers; it is the entity handled by the system scheduler. A kernel thread runs in user mode environment when executing user functions or library calls; it switches to kernel mode environment when executing system calls.

A *kernel-only thread* is a kernel thread that executes only in kernel mode environment. Kernel-only threads are controlled by the kernel mode environment programmer through kernel services.

User mode programs can access so called *user threads* through a library (such as the **libpthreads.a** threads library). User threads are part of a portable programming model. User threads are mapped to kernel threads by the threads library, in an implementation dependent manner. The threads library uses a proprietary interface

to handle kernel threads. See "Understanding Threads" in *AIX Version 4.3 General Programming Concepts: Writing and Debugging Programs* to get detailed information about the user threads library and their implementation.

All threads discussed in this article are kernel threads; and the information applies only to the kernel mode environment. Kernel threads cannot be accessed from the user mode environment, except through the threads library.

Kernel Data Structures

The kernel maintains thread- and process-related information in two types of structures:

- The **user** structure contains process-related information
- The **uthread** structure contains thread-related information.

These structures cannot be accessed directly by kernel extensions and device drivers. They are encapsulated for portability reasons. Many fields that were previously in the **user** structure are now in the **uthread** structure.

Thread Creation, Execution, and Termination

A process is always created with one thread, called the *initial thread*. The initial thread provides compatibility with previous single-threaded processes. The initial thread's stack is the process stack. See "Kernel Process Creation, Execution, and Termination" on page 13 to get more information about kernel process creation.

Other threads can be created, using a two-step procedure. The **thread_create** kernel service allocates and initializes a new thread, and sets its state to idle. The **kthread_start** kernel service then starts the thread, using the specified entry point routine.

A thread is terminated when it executes a return from its entry point, or when it calls the **thread_terminate** kernel service. Its resources are automatically freed. If it is the last thread in the process, the process ends.

Thread Scheduling

Threads are scheduled using one of the following scheduling policies:

- First-in first-out (FIFO) scheduling policy, with fixed priority. Using the FIFO policy with high favored priorities may lead to bad system performance.
- Round-robin (RR) scheduling policy, quantum based and with fixed priority.
- Default AIX scheduling policy, a non-quantum based round-robin scheduling with fluctuating priority. Priority is modified according to the CPU usage of the thread.

Scheduling parameters can be changed using the **thread_setsched** kernel service. The process-oriented **setpri** system call sets the priority of all the threads within a process. The process-oriented **getpri** system call gets the priority of a thread in the process. The scheduling policy and priority of an individual thread can be retrieved from the **ti_policy** and **ti_pri** fields of the **thrdsinfo** structure returned by the **getthrds** system call.

Thread Signal Handling

The signal handling concepts are the following:

- A signal mask is associated with each thread.

- The list of actions associated with each signal number is shared among all threads in the process.
- If the signal action specifies termination, stop, or continue, the entire process, thus including all its threads, is respectively terminated, stopped, or continued.
- Synchronous signals attributable to a particular thread (such as a hardware fault) are delivered to the thread that caused the signal to be generated.
- Signals can be directed to a particular thread. If the target thread has blocked the signal from delivery, the signal remains pending on the thread until the thread unblocks the signal from delivery, or the action associated with the signal is set to ignore by any thread within the process.

The signal mask of a thread is handled by the **limit_sigs** and **sigsetmask** kernel services. The **kthread_kill** kernel service can be used to direct a signal to a particular thread.

In the kernel environment, when a signal is received, no action is taken (no termination or handler invocation), even for the **SIGKILL** signal. In the kernel environment, a thread is not replaced by signals, even the **SIGKILL** signal. A thread in kernel environment, especially kernel-only threads, must *poll* for signals so that signals can be delivered. Polling ensures the proper kernel-mode serialization.

Signals whose actions are applied at generation time (rather than delivery time) have the same effect regardless of whether the target is in kernel or user mode. A kernel-only thread can poll for unmasked signals that are waiting to be delivered by calling the **sig_chk** kernel service. This service returns the signal number of a pending signal that was not blocked or ignored. The thread then uses the signal number to determine which action should be taken. The kernel does not automatically call signal handlers for a thread in kernel mode as it does for user mode.

See “Kernel Process Signal and Exception Handling” on page 14 to get more information about signal handling at process level.

Using Kernel Processes

A kernel process is a process that is created in the kernel protection domain and always executes in the kernel protection domain. Kernel processes can be used in subsystems, by complex device drivers, and by system calls. They can also be used by interrupt handlers to perform asynchronous processing not available in the interrupt environment. Kernel processes can also be used as device managers where asynchronous I/O and device management is required.

Introduction to Kernel Processes

A kernel process (kproc) exists only in the kernel protection domain and differs from a user process in the following ways:

- It is created using the **creatp** and **initp** kernel services.
- It executes only within the kernel protection domain and has all security privileges.
- It can call a restricted set of system calls and all applicable kernel services (see “System Calls Available to Kernel Extensions” on page 31).

- It has access to the global kernel address space (including the kernel pinned and pageable heaps), kernel code, and static data areas.
- It must poll for signals and can choose to ignore any signal delivered, including a **kill** signal.
- It is not subject to replacement by signals.
- Its text and data areas come from the global kernel heap.
- It cannot use shared libraries as such and has no shared library region (see “Using Libraries” on page 5).
- It has a process-private region containing only the **u-block** (user block) structure and possibly the kernel stack.
- Its *parent process* is the process that issued the **creatp** kernel service to create the process.
- It can change its parent process to the **init** process and can use interrupt disable functions for serialization.
- It can use locking to serialize process-time access to critical data structures.
- It can only be a 32-bit process.

A kernel process controls directly the kernel threads (see “Understanding Kernel Threads” on page 9). Since kernel processes are always in the kernel protection domain, threads within a kernel process are kernel-only threads.

A kernel process inherits the environment of its parent process (the one calling the **creatp** kernel service to create it), but with some exceptions. The kernel process will not have a root directory or a current directory when initialized. All uses of the file system functions must specify absolute path names.

Kernel processes created during phase 1 of system boot must not keep any long-term opens on files until phase 2 of system boot or run time has been reached. This is because Base Operating System changes root file systems between phase 1 and phase 2 of system boot. As a result, the system crashes if any files are open at root file system transition time.

Accessing Data from a Kernel Process

Because kernel processes execute in the more privileged kernel protection domain, a kernel process can access data that user processes cannot. This applies to all kernel data, of which there are three general categories:

- The **user block** data structure

The **u-block** (or **u-area**) structure exists for kernel processes and contains roughly the same information for kernel processes as for user-mode processes. A kernel process must use kernel services to query or manipulate data from the **u-area** to maintain modularity and increase portability of code to other platforms.
- The stack for a kernel process

To ensure binary compatibility with older applications, each kernel process has a stack called the *process stack*. This stack is used by the process initial thread.

The location of the stack for a kernel process is implementation-dependent. This stack can be located in global memory or in the process-private segment of the kernel process. A kernel process must not assume automatically that its stack is located in global memory.
- Global kernel memory

A kernel process can also access global kernel memory as well as allocate and de-allocate memory from the kernel heaps. Because it runs in the kernel protection domain, a kernel process can access any valid memory location within the global kernel address space. Memory dynamically allocated from the kernel heaps by the kernel process must be freed by the kernel process before exiting. Unlike user-mode processes, memory that is dynamically allocated by a kernel process is not freed automatically upon process exit.

Cross-Memory Services

Kernel processes must be provided with a valid cross-memory descriptor to access address regions outside the kernel global address space or kernel process address space. For example, if a kernel process is to access data from a user-mode process, the system call using the process must obtain a cross-memory descriptor for the user-mode region to be accessed. Calling the **xmattach** or **xmattach64** kernel service provides a descriptor that can then be made available to the kernel process.

The kernel process should then call the **xmemin** and **xmemout** kernel services to access the targeted cross-memory data area. When the kernel process has completed its operation on the memory area, the cross-memory descriptor must be detached by using the **xmdetach** kernel service.

Kernel Process Creation, Execution, and Termination

A kernel process is created by a kernel-mode routine by calling the **creatp** kernel service. This service allocates and initializes a process block for the process and sets the new process state to idle. This new kernel process does not run until it is initialized by the **initp** kernel service, which must be called in the same process that created the new kernel process (with the **creatp** service). The **creatp** kernel service returns the process identifier for the new kernel process.

The process is created with one kernel-only thread, called the *initial thread*. See “Understanding Kernel Threads” on page 9 to get more information about threads.

After the **initp** kernel service has completed the process initialization, the initial thread is placed on the run queue. On the first dispatch of the newly initialized kernel process, it begins execution at the entry point previously supplied to the **initp** kernel service. The initialization parameters were previously specified in the call to the **initp** kernel service.

A kernel process terminates when it executes a return from its main entry routine. A process should never exit without both freeing all dynamically allocated storage and releasing all locks owned by the kernel process.

When kernel processes exit, the parent process (the one calling the **creatp** and **initp** kernel services to create the kernel process) receives the **SIGCHLD** signal, which indicates the end of a child process. However, it is sometimes undesirable for the parent process to receive the **SIGCHLD** signal due to ending a process. In this case, the **kproc** can call the **setpinit** kernel service to designate again the **init** process as its parent. The **init** process cleans up the state of all its child processes that have become zombie processes. A kernel process can also issue the **setsid** subroutine call to change its session. Signals and job control affecting the parent process session do not affect the kernel process.

Kernel Process Preemption

A kernel process is initially created with the same process priority as its parent. It can therefore be replaced by a more favored kernel or user process. It does not have higher priority just because it is a kernel process. Kernel processes can use the **setpri** or **nice** subroutines to modify their execution priority.

The kernel process can use the locking kernel services to serialize access to critical data structures. This use of locks does not guarantee that the process will not be replaced, but it does ensure that another process trying to acquire the lock waits until the kernel process owning the lock has released it.

Using locks, however, does not provide serialization if a kernel routine can access the critical data while executing in the interrupt environment. Serialization with interrupt handlers must be handled by using locking together with interrupt control. The **disable_lock** and **unlock_enable** kernel services should be used to serialize with interrupt handlers.

Kernel processes must ensure that their maximum path lengths adhere to the specifications for interrupt handlers when executing at an interrupt priority more favored than INTBASE. This ensures that system real-time performance is not degraded.

Kernel Process Signal and Exception Handling

Signals are delivered to exactly one thread within the process which has not blocked the signal from delivery. If all threads within the target process have blocked the signal from delivery, the signal remains pending on the process until a thread unblocks the signal from delivery, or the action associated with the signal is set to ignore by any thread within the process. See “Thread Signal Handling” on page 10 to get more information about signal handling by threads.

Signals whose action is applied at generation time (rather than delivery time) have the same effect regardless of whether the target is a kernel or user process. A kernel process can poll for unmasked signals that are waiting to be delivered by calling the **sig_chk** kernel service. This service returns the signal number of a pending signal that was not blocked or ignored. The kernel process then uses the signal number to determine which action should be taken. The kernel does not automatically call signal handlers for a kernel process as it does for user processes.

A kernel process should also use the exception-catching facilities (**setjmpx**, and **clrjmpx**) available in kernel mode to handle exceptions that can be caused during run time of the kernel process. Exceptions received during the execution of a kernel process are handled the same as exceptions that occur in any kernel-mode routine.

Unhandled exceptions that occur in kernel mode (in any user process while in kernel mode, in an interrupt handler, or in a kernel process) result in a system crash. To avoid crashing the system due to unhandled exceptions, kernel routines should use the **setjmpx**, **clrjmpx**, and **longjmpx** kernel services to handle exceptions that may possibly occur during run time. Refer to “Understanding Exception Handling” on page 18 for more details on handling exceptions.

Kernel Process Use of System Calls

System calls made by kernel processes do not result in a change of protection domain since the kernel process is already within the kernel protection domain. Routines in the kernel (including routines executing in a kernel process) are bound by the loader to the system call and not to the system call handler. When system calls use kernel services to access user-mode data, these kernel services recognize that the system call is running within a kernel process instead of a user process and correctly handle the data accesses.

However, the error information returned from a kernel process system call must be accessed differently than for a user process. A kernel process must use the **getuerror** kernel service to retrieve the system call error information normally provided in the **errno** global variable for user-mode processes. In addition, the kernel process can use the **setuerror** kernel service to set the error information to 0 before calling the system call. The return code from the system call is handled the same for all processes.

Kernel processes can use only a restricted set of the base system calls found in the **syscalls.exp** export file. "System Calls Available to Kernel Extensions" on page 31 shows system calls available to kernel processes.

Accessing User-Mode Data While in Kernel Mode

Kernel extensions use a set of kernel services to access data that is in the user-mode protection domain (see "User Protection Domain" on page 26). These services ensure that the caller has the authority to perform the desired operation at the time of data access. These services also prevent system crashes in a system call when accessing user-mode data. These services can only be called when running in the process environment of the process that contains the user-mode data. See "Process Environment" on page 8 for more information.

Data Transfer Services

The following list shows user-mode data access kernel services (primitives):

Kernel Service	Purpose
suword, suword64	Stores a word of data in user memory.
fubyte, fubyte64	Fetches, or retrieves, a byte of data from user memory.
fuword, fuword64	Fetches, or retrieves, a word of data from user memory.
copyin, copyin64	Copies data between user and kernel memory.
copyout, copyout64	Copies data between user and kernel memory.
copyinstr, copyinstr64	Copies a character string (including the terminating null character) from user to kernel space.

Additional kernel services allow data transfer between user mode and kernel mode when a **uio** structure is used, describes the user-mode data area to be accessed. (Note that this only works for 32-bit processes or with remapped addresses for 64-bit processes.) Following is a list of services that typically are used between the file system and device drivers to perform device I/O:

Kernel Service	Purpose
uiomove	Moves a block of data between kernel space and a space defined by a uio structure.
ureadc	Writes a character to a buffer described by a uio structure.

Kernel Service	Purpose
<code>uwritec</code>	Retrieves a character from a buffer described by a <code>uio</code> structure.

Using Cross-Memory Kernel Services

Occasionally, access to user-mode data is required when not in the environment of the user-mode process that has addressability to the data. Such cases occur when the data is to be accessed asynchronously. Examples of asynchronous accessing include:

- Direct memory access to the user data by I/O devices
- Data access by interrupt handlers
- Data access by a kernel process

In these circumstances, the kernel cross-memory services are required to provide the necessary access. The `xmattach` or `xmattach64` kernel services allow a cross-memory descriptor to be obtained for the data area to be accessed. This service must be called in the process environment of the process containing the data area.

After a cross-memory descriptor has been obtained, the `xmemin` and `xmemout` kernel services can be used to access the data area outside the process environment containing the data. When access to the data area is no longer required, the access must be removed by calling the `xmdetach` kernel service. Kernel extensions should use these services only when absolutely necessary. Because of the machine dependencies of cross-memory operations, using them increases the difficulty of porting the kernel extension to other machine platforms.

Understanding Locking

The following information is provided to assist you in understanding locking.

Lockl Locks

The *lockl locks* (previously called *conventional locks*) are provided for compatibility only and should not be used in new code: simple or complex locks should be used instead. These locks are used to protect a critical section of code which accesses a resource such as a data structure or device, serializing access to the resource. Every thread which accesses the resource must acquire the lock first, and release the lock when finished.

A conventional lock has two states: locked or unlocked. In the *locked* state, a thread is currently executing code in the critical section, and accessing the resource associated with the conventional lock. The thread is considered to be the owner of the conventional lock. No other thread can lock the conventional lock (and therefore enter the critical section) until the owner unlocks it; any thread attempting to do so must wait until the lock is free. In the *unlocked* state, there are no threads accessing the resource or owning the conventional lock.

Lockl locks are recursive and, unlike simple and complex locks, can be awakened by a signal.

Simple Locks

A *simple* lock provides exclusive-write access to a resource such as a data structure or device. Simple locks are not recursive and have only two states: locked or unlocked.

Complex Locks

A *complex* lock can provide either shared or exclusive access to a resource such as a data structure or device. Complex locks are not recursive by default (but can be made recursive) and have three states: exclusive-write, shared-read, or unlocked.

If several threads perform read operations on the resource, they must first acquire the corresponding lock in shared-read mode. Since no threads are updating the resource, it is safe for all to read it. Any thread which writes to the resource must first acquire the lock in exclusive-write mode. This guarantees that no other thread will read or write the resource while it is being updated.

Types of Critical Sections

There are two types of critical sections which must be protected from concurrent execution in order to serialize access to a resource:

thread-thread	These critical sections must be protected (by using the locking kernel services) from concurrent execution by multiple processes or threads. See “Locking Kernel Services” on page 48 for more information.
thread-interrupt	These critical sections must be protected (by using the disable_lock and unlock_enable kernel services) from concurrent execution by an interrupt handler and a thread or process.

Priority Promotion

When a lower priority thread owns a lock which a higher-priority thread is attempting to acquire, the owner has its priority promoted to that of the most favored thread waiting for the lock. When the owner releases the lock, its priority is restored to its normal value. Priority promotion ensures that the lock owner can run and release its lock, so that higher priority processes or threads do not remain blocked on the lock.

Locking Strategy in Kernel Mode

Attention: A kernel extension should not attempt to acquire the kernel lock if it owns any other lock. Doing so can cause unpredictable results or system failure.

A linear hierarchy of locks exists. This hierarchy is imposed by software convention, but is not enforced by the system. The lock **kernel_lock** variable, which is the global kernel lock, has the coarsest granularity. Other types of locks have finer granularity. The following list shows the ordering of locks based on granularity:

- The **kernel_lock** global kernel lock

Note: Avoid using the **kernel_lock** global kernel lock variable in new code since it is only included for compatibility purposes and may be removed from future versions.

- File system locks (private to file systems)

- Device driver locks (private to device drivers)
- Private fine-granularity locks

Locks should generally be released in the reverse order from which they were acquired; all locks must be released before a kernel process exits or leaves kernel mode. Kernel mode processes do not receive any signals while they hold any lock.

Understanding Exception Handling

Exception handling involves a basic distinction between *interrupts* and *exceptions*:

- An interrupt is an asynchronous event and is not associated with the instruction that is executing when the interrupt occurs.
- An exception is a synchronous event and is directly caused by the instruction that is executing when the exception occurs.

The computer hardware generally uses the same mechanism to report both interrupts and exceptions. The machine saves and modifies some of its state and forces a branch to a particular location. When decoding the reason for the machine interrupt, the interrupt handler determines whether the event is an interrupt or an exception, then processes the event accordingly.

Note: Ordinary page faults are treated more like interrupts than exceptions. The only difference between a page-fault interrupt and other interrupts is that the interrupted program is not dispatchable until the page fault is resolved.

Exception Processing

When an exception occurs, the current instruction stream cannot continue. If you ignore the exception, the results of executing the instruction may become undefined. Further execution of the program may cause unpredictable results. The kernel provides a default exception-handling mechanism by which an instruction stream (a process- or interrupt-level program) can specify what action is to be taken when an exception occurs. Exceptions are handled differently depending on whether they occurred while executing in kernel mode (see “Kernel-Mode Exception Handling”) or user mode (see “User-Mode Exception Handling” on page 23).

Default Exception-Handling Mechanism

If no exception handler is currently defined when an exception occurs, typically one of two things happens:

- If the exception occurs while a process is executing in user mode, the process is sent a signal relevant to the type of exception.
- If the exception occurs while in kernel mode, the system halts.

Kernel-Mode Exception Handling

Exception handling in kernel mode extends the **setjump** and **longjump** subroutines context-save-and-restore mechanism by providing **setjmpx** and **longjmpx** kernel services to handle exceptions. The traditional system mechanism is extended by allowing these exception handlers (or context-save checkpoints) to be stacked on a per-process or per-interrupt handler basis.

This stacking mechanism allows the execution point and context of a process or interrupt handler to be restored to a point in the process or interrupt handler, *at the point of return from the **setjmpx** kernel service*. When execution returns to this point, the return code from **setjmpx** kernel service indicates the type of exception that occurred so that the process or interrupt handler state can be fully restored. Appropriate retry or recovery operations are then invoked by the software performing the operation.

When an exception occurs, the kernel first-level exception handler gets control. The first-level exception determines what type of exception has occurred and saves information necessary for handling the specific type of exception. For an I/O exception, the first-level handler also enables again the programmed I/O operations.

The first-level exception handler then modifies the saved context of the interrupted process or interrupt handler. It does so to execute the **longjmpx** kernel service when the first-level exception handler returns to the interrupted process or interrupt handler.

The **longjmpx** kernel service executes in the environment of the code that caused the exception and restores the current context from the topmost jump buffer on the stack of saved contexts. As a result, the state of the process or interrupt handler that caused the exception is restored to the point of the return from the **setjmpx** kernel service. (The return code, nevertheless, indicates that an exception has occurred.)

The process or interrupt handler software should then check the return code and invoke exception handling code to restore fully the state of the process or interrupt handler. Additional information about the exception can be obtained by using the **getexcept** kernel service.

User-Defined Exception Handling

A typical exception handler should do the following:

- Perform any necessary clean-up such as freeing storage or segment registers and releasing other resources.
- If the exception is recognized by the current handler and can be handled entirely within the routine, the handler should establish itself again by calling the **setjmpx** kernel service. This allows normal processing to continue.
- If the exception is not recognized by the current handler, it must be passed to the previously stacked exception handler. The exception is passed by calling the **longjmpx** kernel service, which either calls the previous handler (if any) or takes the system's default exception-handling mechanism.
- If the exception is recognized by the current handler but cannot be handled, it is treated as though it is unrecognized. The **longjmpx** kernel service is called, which either passes the exception along to the previous handler (if any) or takes the system default exception-handling mechanism.

When a kernel routine that has established an exception handler completes normally, it must remove its exception handler from the stack (by using the **clrjmpx** kernel service) before returning to its caller.

Note: When the **longjmpx** kernel service invokes an exception handler, that handler's entry is automatically removed from the stack.

Implementing Kernel Exception Handlers

The following information is provided to assist you in implementing kernel exception handlers.

setjmpx, longjmpx, and clrjmpx Kernel Services

The **setjmpx** kernel service provides a way to save the following portions of the program state at the point of a call:

- Nonvolatile general registers
- Stack pointer
- TOC pointer
- Interrupt priority number (**intpri**)
- Ownership of kernel-mode lock

This state can be restored later by calling the **longjmpx** kernel service, which accomplishes the following tasks:

- Reloads the registers (including TOC and stack pointers).
- Enables or disables to the correct interrupt level.
- Conditionally acquires or releases the kernel-mode lock.
- Forces a branch back to the point of original return from the **setjmpx** kernel service.

The **setjmpx** kernel service takes the address of a jump buffer (a **label_t** structure) as an explicit parameter. This structure can be defined anywhere including on the stack (as an automatic variable). After noting the state data in the jump buffer, the **setjmpx** kernel service pushes the buffer onto the top of a stack that is maintained in the machine-state save structure.

The **longjmpx** kernel service is used to return to the point in the code at which the **setjmpx** kernel service was called. Specifically, the **longjmpx** kernel service returns to the most recently created jump buffer, as indicated by the top of the stack anchored in the machine-state save structure.

The parameter to the **longjmpx** kernel service is an exception code that is passed to the resumed program as the return code from the **setjmp** kernel service. The resumed program tests this code to determine the conditions under which the **setjmpx** kernel service is returning. If the **setjmpx** kernel service has just saved its jump buffer, the return code is 0. If an exception *has* occurred, the program is entered by a call to the **longjmpx** kernel service, which passes along a return code that is *not* equal to 0.

Note: Only the resources listed here are saved by the **setjmpx** kernel service and restored by the **longjmpx** kernel service. Other resources, in particular segment registers, are not restored. A call to the **longjmpx** kernel service, by definition, returns to an earlier point in the program. The program code must free any resources that are allocated between the call to the **setjmpx** kernel service and the call to the **longjmpx** kernel service.

If the exception handler stack is empty when the **longjmpx** kernel service is issued, there is no place to jump to and the system default exception-handling mechanism is used. If the stack is not empty, the context that is defined by the topmost jump buffer is reloaded and resumed. The topmost buffer is then removed from the stack.

The **clrjmpx** kernel service removes the top element from the stack as placed there by the **setjmpx** kernel service. The caller to the **clrjmpx** kernel service is expected to know exactly which jump buffer is being removed. This should have been established earlier in the code by a call to the **setjmpx** kernel service. Accordingly, the address of the buffer is required as a parameter to the **clrjmpx** kernel service. It can then perform consistency checking by asserting that the address passed is indeed the address of the top stack element.

Exception Handler Environment

The stacked exception handlers run in the environment in which the exception occurs. That is, an exception occurring in a process environment causes the next dispatch of the process to run the exception handler on the top of the stack of exception handlers for that process. An exception occurring in an interrupt handler causes the interrupt handler to return to the context saved by the last call to the **setjmpx** kernel service made by the interrupt handler.

Note: An interrupt handler context is newly created each time the interrupt handler is invoked. As a result, exception handlers for interrupt handlers must be registered (by calling the **setjmpx** kernel service) each time the interrupt handler is invoked. Otherwise, an exception detected during execution of the interrupt handler will be handled by the default handler.

Restrictions on Using the setjmpx Kernel Service

Process and interrupt handler routines registering exception handlers with the **setjmpx** kernel service must not return to their caller before removing the saved jump buffer or buffers from the list of jump buffers. A saved jump buffer can be removed by invoking the **clrjmpx** kernel service in the reverse order of the **setjmpx** calls. The saved jump buffer must be removed before return because the routine's context no longer exists once the routine has returned to its caller.

If, on the other hand, an exception does occur (that is, the return code from **setjmpx** kernel service is nonzero), the jump buffer is automatically removed from the list of jump buffers. In this case, a call to the **clrjmpx** kernel service for the jump buffer must not be performed.

Care must also be taken in defining variables that are used after the context save (the call to the **setjmpx** service), and then again by the exception handler. Sensitive variables of this nature must be restored to their correct value by the exception handler when an exception occurs.

Note: If the last value of the variable is desired at exception time, the variable data type must be declared as "volatile."

Exception handling is concluded in one of two ways. Either a registered exception handler handles the exception and continues from the saved context, or the default exception handler is reached by exhausting the stack of jump buffers.

Exception Codes

The `/usr/include/sys/except.h` file contains a list of code numbers corresponding to the various types of hardware exceptions. When an exception handler is invoked (the return from the **setjmpx** kernel service is not equal to 0), it is the responsibility of the handler to test the code to ensure that the exception is one the routine can handle. If it is not an expected code, the exception handler must:

- Release any resources that would not otherwise be freed (buffers, segment registers, storage acquired using the **xmalloc** routines).
- Call the **longjmpx** kernel service, passing it the exception code as a parameter.

Thus, when an exception handler does not recognize the exception for which it has been invoked, it passes the exception on to the next most recent exception handler. This continues until an exception handler is reached that recognizes the code and can handle it. Eventually, if no exception handler can handle the exception, the stack is exhausted and the system default action is taken.

In this manner, a component can allocate resources (after calling the **setjmpx** kernel service to establish an exception handler) and be assured that the resources will later be released. This ensures the exception handler gets a chance to release those resources regardless of what events occur before the instruction stream (a process- or interrupt-level code) is terminated.

By coding the exception handler to recognize what exception codes it can process, (rather than encoding this knowledge in the stack entries), a powerful and simple-to-use mechanism is created. Each handler need only investigate the exception code that it receives rather than just assuming that it was invoked because a particular exception has occurred to implement this scheme, the set of exception codes used cannot have duplicates.

Exceptions generated by hardware use one of the codes in the **/usr/include/sys/except.h** file. However, the **longjmpx** kernel service can be invoked by any kernel component, and any integer can serve as the exception code. A mechanism similar to the old-style **setjmp** and **longjmp** kernel services can be implemented on top of the **setjmpx/longjmpx** stack by using exception codes outside the range of those used for hardware exceptions.

To implement this old-style mechanism, a unique set of exception codes is needed. These codes must not conflict with either the pre-assigned hardware codes or codes used by any other component. A simple way to get such codes is to use the addresses of unique objects as code values.

For example, a program that establishes an exception handler might compare the exception code to the address of its own entry point (that is, by using its function descriptor). Later on in the calling sequence, after any number of intervening calls to the **setjmpx** kernel service by other programs, a program can issue a call to the **longjmpx** kernel service and pass the address of the agreed-on function descriptor as the code. This code is only recognized by a single exception handler. All the intervening ones just clean up their resources and pass the code to the **longjmpx** kernel service again.

Addresses of function descriptors are not the only possibilities for unique code numbers. For example, addresses of external variables can also be used. By using addresses that are resolved to unique values by the binder and loader, the problem of code-space collision is transformed into a problem of external-name collision. This problem is easier to solve, and is routinely solved whenever the system is built. By comparison, pre-assigning exception numbers by using **#define** statements in a header file is a much more cumbersome and error-prone method.

Hardware Detection of Exceptions

Each of the exception types results in a hardware interrupt. For each such interrupt, a first-level interrupt handler (FLIH) saves the state of the executing

program and calls a second-level handler (SLIH). The SLIH is passed a pointer to the machine-state save structure and a code indicating the cause of the interrupt.

When a SLIH determines that a hardware interrupt should actually be considered a synchronous exception, it sets up the machine-state save to invoke the **longjmpx** kernel service, and then returns. The FLIH then resumes the instruction stream at the entry to the **longjmpx** service.

The **longjmpx** service then invokes the top exception handler on the stack or takes the system default action as previously described.

User-Mode Exception Handling

Exceptions that occur in a user-mode process and are not automatically handled by the kernel cause the user-mode process to be signaled. If the process is in a state in which it cannot take the signal, it is terminated and the information logged. Kernel routines can install user-mode exception handlers that catch exceptions before they are signaled to the user-mode process.

The **uexadd** and **uexdel** kernel services allow system-wide user-mode exception handlers to be added and removed.

The most recently registered exception handler is the first called. If it cannot handle the exception, the next most recent handler on the list is called, and this second handler attempts to handle the exception. If this attempt fails, successive handlers are tried, until the default handler is called, which generates the signal.

Additional information about the exception can be obtained by using the **getexcept** kernel service.

64-bit Kernel Extension Development

AIX kernel extensions run in 32-bit mode, even when processing requests made by 64-bit applications. This allows kernel extensions which were built for AIX releases which only supported 32-bit systems to run on AIX releases which support both 32-bit and 64-bit systems. For these old kernel extensions and all kernel extensions which have not been designed to work with 64-bit applications, only 32-bit applications can be supported. A 64-bit application will fail to link if it attempts to make use of a system call from a kernel extension that has not been modified to support 64-bit applications.

A kernel extension can indicate that it supports 64-bit applications by setting the **SYS_64BIT** flag when it is loaded using the **sysconfig** routine.

Kernel extension support for 64-bit applications has two aspects.

The first aspect is the use of new kernel services for working with the 64-bit user address space. The new 64-bit services for examining and manipulating the 64-bit address space are **as_att64**, **as_det64**, **as_geth64**, **as_puth64**, **as_seth64**, and **as_getsrval64**. The new services for copying data to or from 64-bit address spaces are **copyin64**, **copyout64**, **copyinstr64**, **fubyte64**, **fuword64**, **subyte64**, and **suword64**. The new service for doing cross-memory attaches to memory in a 64-bit address space is **xmattach64**. The new services for creating real memory mappings

are **rmmmap_create64** and **rmmmap_remove64**. The major difference between all these services and their 32-bit counterparts is that they use 64 bit user addresses rather than 32 bit user addresses.

The new service for determining whether a process (and its address space) is 32-bit or 64-bit is **IS64U**.

The second aspect of supporting 64-bit applications is taking 64 bit user data passed to system calls and transforming that data to 32 bit data which can be passed through the system call handler to the kernel extension. If the types of the parameters passed to a system call are all 32 bit sized or smaller (in 64-bit compilation mode), then no extra work is required. However, if 64-bit data (long or long long C types) or addresses are passed to the system call, then the data must be split and passed in two parameters or transformed into a 32-bit value.

To assist in using 64-bit user addresses in kernel mode, a set of "remapping" services are provided which transform the 64 bit user addresses into 32 bit addresses which can be used by most of the old 32-bit user address manipulation services in the kernel. In this way, kernel code which has not been modified to work with 64-bit user addresses can always use the 32 bit address (either from a 32-bit application or as a transformed 64-bit address from a 64-bit application). Services such as **copyin**, **copyout**, **xmattach**, **fuword**, **fubyte**, **suword**, and **subyte** will correctly work with the 32-bit address by transforming it back to a 64-bit address.

The remapping services consist of a set of routines which can be called (in 64-bit execution mode) from a library and a matching set of routines which can be called (in 32-bit mode) from a kernel extension. The library routines determine which 32-bit addresses should be matched to each 64-bit address and package this information in a data structure to be passed to the kernel. The library then passes all the parameters (including the 32-bit addresses which were associated with the 64-bit addresses) through the 64-bit system call to the kernel extension.

The kernel extension passes the remapping information (created by the library remapping routine) to the remapping kernel service, which saves it for use by address space services for the duration of that system call.

A limited number of addresses can be remapped because remapping must be done on a segment (256K bytes) basis. One address remapping can use multiple of these segments, depending on the location of the address relative to a segment boundary and depending on the length of the address range being remapped. Multiple address remappings can utilize a single segment of remapping if they all happen to fall in the range of a single segment.

To simplify the use of these remapping services, a set of macros have been declared (in **sys/remap.h**) which hide many of the underlying details. These macros should be used to avoid mistakes in the use of the underlying services. **REMAP_DCL** is used to declare the data structures used by the other services. **REMAP_SETUP** and **REMAP_SETUP_WITH_LEN** are used to fill in the data structures with the 64-bit addresses and lengths. **REMAP**, **REMAP_VOID**, **REMAP_IDENTITY**, and **REMAP_IDENTITY_VOID** are called by the library to determine the remappings. **REMAP_64** and **REMAP_64_VOID** are called by the kernel extension to register the remapping information with the kernel.

Chapter 2. System Calls

In the operating system, a system call is a routine that crosses a protection domain. Adding system calls is one of several ways to extend the functions provided by the kernel. System calls provide user-mode access to special kernel functions.

The distinction between a system call and an ordinary function call is only important in the kernel programming environment. User-mode application programs are not usually aware of this distinction between system calls and ordinary function calls in the operating system.

Operating system functions are made available to the application program in the form of programming libraries (see “Using Libraries” on page 5). A set of library functions found in a library such as **libc** can have functions that perform some user-mode processing and then internally start a system call. In other cases, the system call can be directly exported by the library without a user-mode layer.

In this way, operating system functions available to application programs can be split or moved between user-mode functions and kernel-mode functions as required for different releases or machine platforms. Such movement does not affect the application program.

Differences Between a System Call and a User Function

A system call differs from a user function in several key ways:

- A system call has more privilege than a normal subroutine. A system call runs with kernel-mode privilege in the kernel protection domain.
- System call code and data are located in global kernel memory.
- System call routines can create and use kernel processes to perform asynchronous processing.
- System calls cannot use shared libraries or any symbols not found in the kernel protection domain.

Understanding System Call Execution

The system call handler gains control when a user program starts a system call. The system call handler changes the protection domain from the caller protection domain, *user*, to the system call protection domain, *kernel*, and switches to a protected stack.

The system call handler then calls the function supporting the system call. The loader maintains a table of the currently defined system calls for this purpose.

The system call runs within the calling process, but with more privilege than the calling process. This is because the protection domain has changed from *user* to *kernel*.

The system call function returns to the system call handler when it has performed its operation. The system call handler then restores the state of the process and returns to the user program.

There are two major protection domains in the operating system: the *user mode protection domain* and the *kernel mode protection domain*.

User Protection Domain

Programs that run in the *user protection domain* include those running within user processes and those within real-time processes. This protection domain implies that code runs in user execution mode and has:

- Read/write access to user data in the process private region
- Read access to the user text and shared text regions
- Access to shared data regions using the shared memory functions.

Programs running in the user protection domain do not have access to the kernel or kernel data segments except indirectly through the use of system calls. A program in this protection domain can only affect its own execution environment and runs in the processor unprivileged state.

Kernel Protection Domain

Programs that run in the *kernel protection domain* include interrupt handlers, kernel processes, the base kernel, and kernel extensions (device drivers, system calls, and file systems). This protection domain implies that code runs in kernel execution mode and has the following access:

- Read/write access to the global kernel address space
- Read/write access to the kernel data in the process private region when running within a process.

User data within the process address space must be accessed using kernel services. Programs running in this protection domain can affect the execution environments of all programs because they:

- Can access global system data.
- Can use kernel services.
- Are exempt from all security restraints.
- Run in the processor privileged state.

All kernel extensions run in the kernel protection domain as described above. The use of a system call by a user-mode process allows a kernel function to be called from user mode. Access to functions that directly or indirectly invoke system calls is typically provided by programming libraries providing access to operating system functions (see “Using Libraries” on page 5).

Actions of the System Call Handler

When a call is made in user mode that starts a system call, the system call handler is invoked. This system call handler switches the protection domain from user to kernel and performs the following steps:

1. Sets privileged access to the process private address region.
2. Sets privileged access to the kernel address regions.
3. Sets the `ut_error` field in the **uthread** structure to 0.
4. Switches to the kernel stack.
5. Starts the specified kernel function (the target of the system call).

On return from the specified kernel function, the system call handler performs the following steps before returning to the caller:

1. Switches back to the user stack.
2. Updates the thread-specific **errno** variable if the `ut_error` field is not equal to 0.
3. Clears the privileged access to the kernel address regions.
4. Clears the privileged access to the process private region.
5. Performs signal processing if a signal is pending.

The system call (and associated kernel function) runs within the context of the calling process, but with more privilege than the user-mode caller. This is because the system call handler has changed the protection domain from user state to kernel state. When the kernel function that was the target of the system call has performed the requested operation (or encountered an error), it returns to the system call handler. When this happens, the system call handler restores the state and protection domain back to user mode and returns control to the user program.

Accessing Kernel Data While in a System Call

Kinds of Kernel Data

Attention: Incorrectly modifying fields in kernel or user block structures can cause unpredictable results or system crashes while running in the kernel protection domain.

A system call can access data that the caller cannot because the system call is running in a more privileged protection domain. This applies to all kernel data, of which there are three general categories:

- The user block data structure:

System calls should use the available kernel services and system calls to access or modify data traditionally found in the **u area** (**u**ser structure) and in the thread-specific **uthread** structures. For example, the system call handler uses the thread's `ut_error` system call error field to set the thread-specific **errno** variable before returning to user mode. This field can be read or set by using the **getuerror** and **setuerror** kernel services. The current process ID can be obtained by using the **getpid** kernel service, and the current thread ID can be obtained by using the **thread_self** kernel service.

- Global memory

System calls can also access global memory such as the kernel and kernel data regions. These regions contain the code and static data for the system call as well as the rest of the kernel.

- The stack for a system call:

A system call routine runs on a protected stack that depends on the thread. This stack allows the system call handler to safely start a system call even when the caller does not have a valid stack pointer initialized. It also allows system calls to access privileged information with automatic variables without exposing the information to the caller.

Passing Parameters to System Calls

The fact that a system call does not run on the same stack as the caller imposes one limitation. System calls are limited in the number of parameters they can use.

The operating system linkage convention passes some parameters in registers and the rest on the stack. The system call handler ensures that the first 8 words of the parameter list are accessible to the system call. All other parameters are not accessible.

For some languages, various types of parameters can take more than one word in the parameter list. The writer of a system call should be familiar with the way parameters are passed by the compiler and conform to this 8-word limit.

Preempting a System Call

The kernel allows a thread to be preempted by a more favored thread even when starting a system call. This is not typical of most operating systems. The kernel makes this change to enhance support for real-time processes and large multiuser systems.

System calls should use the **lockl** and **unlockl** kernel services “Locking Kernel Services” on page 48 to serialize access to any global data that they access. Remember that all of the system call static data is located in global memory and therefore must be accessed serially.

The **lockl** kernel service ensures locking kernel services ensure that the owner of a lock runs with the most favored priority of any of the waiters of that lock. It does this by assigning to the lock owner the thread priority of the most favored waiter for the lock. This mechanism is similar to the standard operating system sleep priority. However, the thread priority must be assigned when the resource is allocated since the system call can be inactivated by preemption, as well as by calling sleep. Unlocking the lock restores the thread priority.

Note that a thread can be preempted even when it owns a lock. The lock only ensures that another thread that tries to lock the resource waits until the owner of the resource unlocks it. A system call must never return with a lock locked. By convention, a locking hierarchy is followed to prevent deadlocks. “Understanding Locking” on page 16 provides more information on locking.

Handling Signals While in a System Call

Signals can be generated asynchronously or synchronously with respect to the process that receives the signal. An asynchronously generated signal is one that results from some action external to a process. It is not directly related to the current instruction stream of that process. Generally these are generated by other processes for interprocess communication or by device drivers.

A synchronously generated signal is one that results from the current instruction stream of the process. These signals cause interrupts. Examples of such cases are the calling of an illegal instruction, or an attempted data access to nonexistent address space. These are often referred to as exceptions.

Delivery of Signals to a System Call

The kernel delays the delivery of all signals, including **SIGKILL**, when starting a system call, device driver, or other kernel extension. The signal takes effect upon leaving the kernel and returning from the system call. This happens when the process returns to the user protection domain, just before running the first instruction at the caller return address. Signal delivery for kernel processes is described in “Using Kernel Processes” on page 11.

Asynchronous Signals and Wait Termination

An asynchronous signal can alter the operation of a system call or kernel extension by terminating a long wait. Kernel services such as `e_block_thread`, `e_sleep_thread`, and `et_wait` all support terminating a wait by a signal. These services provide three options:

- The **short-wait** option of not terminating the wait due to a signal
- Terminating the wait by return from the kernel service with a return code of **interrupted-by-signal**
- Calling the **longjmpx** kernel service to resume at a previously saved context in the event of a signal

The **sleep** kernel service, provided for compatibility, also supports the **PCATCH** and **SWAKEONSIG** options to control the response to a signal during the **sleep** function.

Previously, the kernel automatically saved context on entry to the system call handler. As a result, any long (interruptible) sleep not specifying the **PCATCH** option returned control to the saved context when a signal interrupted the wait. The system call handler then set the **errno** global variable to **EINTR** and returned a return code of -1 from the system call.

The kernel, however, requires each system call that can directly or indirectly issue a **sleep** call without the **PCATCH** option to set up a saved context using the **setjmpx** kernel service. This is done to avoid overhead for system calls that handle waits terminated by signals. Using the **setjmpx** service, the system can set up a saved context sets the system call return code to a -1 and the `ut_error` field to **EINTR**, if a signal interrupts a long wait not specifying **return-from-signal**.

It is probably faster and more robust to specify **return-from-signal** on all long waits and use the return code to control the system call return.

Stacking Saved Contexts for Nested `setjmpx` Calls

The kernel supports nested calls to the **setjmpx** kernel service. It implements the stack of saved contexts by maintaining a linked list of context information anchored in the machine state save area. This area is in the user block structure for a process. Interrupt handlers have special machine state save areas.

An initial context is set up for each process by the **initp** kernel service for kernel processes and by the **fork** subroutine for user processes. The process terminates if that context is resumed.

Handling Exceptions While in a System Call

Exceptions are interrupts detected by the processor as a result of the current instruction stream. They therefore take effect synchronously with respect to the current process.

The default exception handling normally generates a signal if the process is in a state where signals are delivered without delay. If delivery of a signal can be delayed, however, default exception handling causes a dump.

Alternative Exception Handling Using the `setjmpx` Kernel Service

For certain types of exceptions, a system call can specify unique exception-handler routines through calls to the `setjmpx` service. The exception handler routine is saved as part of the stacked saved context. Each exception handler is passed the exception type as a parameter.

The exception handler returns a value that can specify any of the following:

- The process should resume with the instruction that caused the exception.
- The process should return to the saved context that is on the top of the stack of contexts.
- The exception handler did not handle the exception.

In that case, the next exception handler in the stack of contexts is called. If none of the stacked exception handlers handle the exception, the kernel performs default exception handling. The `setjmpx` and `longjmpx` kernel services help implement exception handlers.

Understanding Nesting and Kernel-Mode Use of System Calls

The operating system supports nested system calls with some restrictions. System calls (and any other kernel-mode routines running under the process environment of a user-mode process) can use system calls that pass all parameters by value. System calls and other kernel-mode routines must not start system calls that have one or more parameters passed by reference. Doing so can result in a system crash. This is because system calls with reference parameters assume that the referenced data area is in the user protection domain. As a result, these system calls must use special kernel services to access the data. However, these services are unsuccessful if the data area they are trying to access is not in the user protection domain.

This restriction does not apply to kernel processes. User-mode data access services can distinguish between kernel processes and user-mode processes in kernel mode. As a result, these services can access the referenced data areas accessed correctly when the caller is a kernel process.

Kernel processes cannot call the `fork` or `exec` system calls, among others. A list of the base operating system calls available to system calls or other routines in kernel mode is provided in “System Calls Available to Kernel Extensions” on page 31.

Page Faulting within System Calls

Attention: A page fault that occurs while external interrupts are disabled results in a system crash. Therefore, a system call should be programmed to ensure that its code, data, and stack are pinned before it disables external interrupts.

Most data accessed by system calls is pageable by default. This includes the system call code, static data, dynamically allocated data, and stack. As a result, a system call can be preempted in two ways:

- By a more favored process, or by an equally favored process when a time slice has been exhausted
- By losing control of the processor when it page faults

In the latter case, even less-favored processes can run while the system call is waiting for the paging I/O to complete.

Returning Error Information from System Calls

Error information returned by system calls differs from that returned by kernel services that are not system calls. System calls typically provide a return code of 0 if no error has occurred, or -1 if an error has occurred. In the latter case, the error value is placed in the `ut_error` field of the thread's **uthread** structure. In some cases, when data is returned by the return code, a data value of -1 indicates error. Or alternatively, a value of NULL can indicate error, depending on the interface and function definition of the system call.

In any case, when an error condition is to be returned, the `ut_error` field should be updated by the system call prior to returning from the system call function. The `ut_error` field is accessed by using the **getuerror** and **setuerror** kernel services.

Before actually calling the system call function, the system call handler sets the `ut_error` field to 0. Upon return from the system call function, the system call handler copies the value found in `ut_error` into the thread-specific **errno** variable if `ut_error` was nonzero. After setting the **errno** variable, the system call handler returns to user mode with the return code provided by the system call function.

Kernel-mode callers of system calls must be aware of this return code convention and use the **getuerror** kernel service to obtain the error value when an error indication is returned by the system call. When system calls are nested, the system call function called by the system call handler can return the error value provided by the nested system call function or can replace this value with a new one by using the **setuerror** kernel service.

System Calls Available to Kernel Extensions

The following system calls are grouped according to which subroutines call them:

- "System Calls Available to All Kernel Extensions"
- "System Calls Available to Kernel Processes Only" on page 32

Note: System calls are not available to interrupt handlers.

System Calls Available to All Kernel Extensions

gethostid	Gets the unique identifier of the current host.
getpgrp	Gets the process ID, process group ID, and parent process ID.
getppid	Gets the process ID, process group ID, and parent process ID.
getpri	Returns the scheduling priority of a process.
getpriority	Gets or sets the <i>nice</i> value.
semget	Gets a set of semaphores.
seteuid	Sets the process user IDs.
setgid	Sets the process group IDs.
sethostid	Sets the unique identifier of the current host.
setpgrp	Sets the process group IDs.
setppid	Sets the process group IDs.
setpri	Sets a process scheduling priority to a constant value.
setpriority	Gets or sets the <i>nice</i> value.
setreuid	Sets the process user IDs.
setsid	Creates a session and sets the process group ID.

setuid	Sets the process user IDs.
ulimit	Sets and gets user limits.
umask	Sets and gets the value of the file-creation mask.

System Calls Available to Kernel Processes Only

disclaim	Disclaims the content of a memory address range.
getdomainname	Gets the name of the current domain.
getgroups	Gets the concurrent group set of the current process.
gethostname	Gets the name of the local host.
getpeername	Gets the name of the peer socket.
getrlimit	Controls maximum system resource consumption.
getrusage	Displays information about resource use.
getsockname	Gets the socket name.
getsockopt	Gets options on sockets.
gettimer	Gets and sets the current value for the specified systemwide timer.
resabs	Manipulates the expiration time of interval timers.
resinc	Manipulates the expiration time of interval timers.
restimer	Gets and sets the current value for the specified systemwide timer.
semctl	Controls semaphore operations.
semop	Performs semaphore operations.
setdomainname	Sets the name of the current domain.
setgroups	Sets the concurrent group set of the current process.
sethostname	Sets the name of the current host.
setrlimit	Controls maximum system resource consumption.
settimer	Gets and sets the current value for the specified systemwide timer.
shmat	Attaches a shared memory segment or a mapped file to the current process.
shmctl	Controls shared memory operations.
shmdt	Detaches a shared memory segment.
shmget	Gets shared memory segments.
sigaction	Specifies the action to take upon delivery of a signal.
sigprocmask	Sets the current signal mask.
sigstack	Sets and gets signal stack context.
sigsuspend	Atomically changes the set of blocked signals and waits for a signal.
sysconfig	Provides a service for controlling system/kernel configuration.
times	Displays information about resource use.
uname	Gets the name of the current system.
unamex	Gets the name of the current system.
usrinfo	Gets and sets user information about the owner of the current process.
utimes	Sets file access and modification times.

Chapter 3. Virtual File Systems

The virtual file system (VFS) interface, also known as the *v-node* interface, provides a bridge between the physical and logical file systems. The information that follows discusses the virtual file system interface, its data structures, and its header files, and explains how to configure a virtual file system.

There are two essential components in the file system:

Logical file system	Provides support for the system call interface.
Physical file system	Manages permanent storage of data.

The interface between the physical and logical file systems is the virtual file system interface (see “Understanding the Virtual File System Interface” on page 36). This interface allows support for multiple concurrent instances of physical file systems, each of which is called a file system implementation. The file system implementation can support storing the file data in the local node or at a remote node.

The virtual file system interface is usually referred to as the *v-node* interface (see “Virtual File System Overview” on page 35). The *v-node* structure is the key element in communication between the virtual file system and the layers that call it (see “Understanding Virtual Nodes (V-nodes)” on page 35).

Both the virtual and logical file systems exist across all of this operating system family’s platforms.

Logical File System Overview

The *logical file system* is the level of the file system at which users can request file operations by system call. This level of the file system provides the kernel with a consistent view of what may be multiple physical file systems and multiple file system implementations. As far as the logical file system is concerned, file system types, whether local, remote, or strictly logical, and regardless of implementation, are indistinguishable.

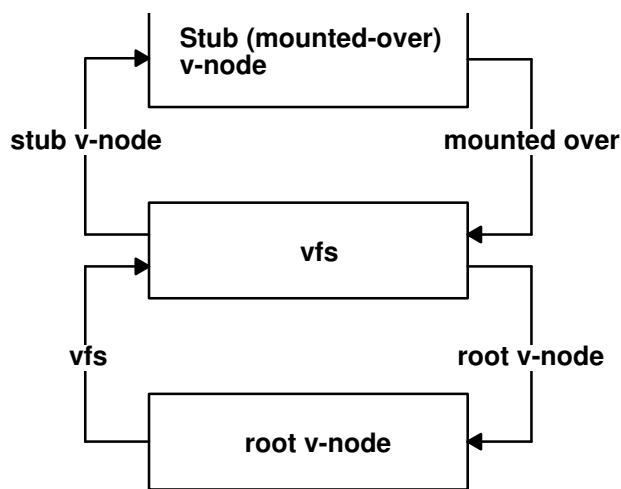
A consistent view of file system implementations is made possible by the virtual file system abstraction (see “Virtual File System Overview” on page 35). This abstraction specifies the set of file system operations that an implementation must include in order to carry out logical file system requests (see “Requirements for a File System Implementation” on page 36). Physical file systems can differ in how they implement these predefined operations, but they must present a uniform interface to the logical file system.

Each set of predefined operations implemented constitutes a virtual file system. As such, a single physical file system can appear to the logical file system as one or more separate virtual file systems.

Virtual file system operations are available at the logical file system level through the *virtual file system switch*. This array contains one entry for each virtual file system, with each entry holding entry point addresses for separate operations. Each file system type has a set of entries in the virtual file system switch.

The logical file system and the virtual file system switch support other operating system file-system access semantics. This does not mean that only other operating system file systems can be supported. It does mean, however, that a file system implementation must be designed to fit into the logical file system model. Operations or information requested from a file system implementation need be performed only to the extent possible.

Logical file system can also refer to the tree of known path names in force while the system is running. A virtual file system (such as shown in the following figure) that is mounted onto the logical file system tree itself becomes part of that tree. In fact, a single virtual file system can be mounted onto the logical file system tree at multiple points, so that nodes in the virtual subtree have multiple names. Multiple mount points allow maximum flexibility when constructing the logical file system view.



Virtual File System Mount Point

Component Structure of the Logical File System

The logical file system is divided into the following components:

- System calls

Implement services exported to users. System calls that carry out file system requests do the following:

- Map the user’s parameters to a file system object. This requires that the system call component use the v-node (virtual node) component to follow the object’s path name (see “Understanding Virtual Nodes (V-nodes)” on page 35). In addition, the system call must resolve a file descriptor or establish implicit (mapped) references using the open file component.
- Verify that a requested operation is applicable to the type of the specified object.
- Dispatch a request to the file system implementation to perform operations.

- Logical file system file routines

Manage open file table entries and per-process file descriptors. An open file table entry records the authorization of a process’s access to a file system object. A user can refer to an open file table entry through a file descriptor or by accessing the virtual memory to which the file was mapped. The logical file system routines are those kernel services, such as `fp_ioctl` and `fp_select`, that begin with the prefix `fp_`.

- v-nodes
Provide system calls with a mechanism for local name resolution. Local name resolution allows the logical file system to access multiple file system implementations through a uniform name space.

Virtual File System Overview

The virtual file system is an abstraction of a physical file system implementation. It provides a consistent interface to multiple file systems, both local and remote. This consistent interface allows the user to view the directory tree on the running system as a single entity even when the tree is made up of a number of diverse file system types. The interface also allows the logical file system code in the kernel to operate without regard to the type of file system being accessed.

A virtual file system can also be viewed as a subset of the logical file system tree, that part belonging to a single file system implementation. A virtual file system can be physical (the instantiation of a physical file system), remote, or strictly logical. In the latter case, for example, a virtual file system need not actually be a true file system or entail any underlying physical storage device.

A virtual file system mount point (see figure on page 34) grafts a virtual file system subtree onto the logical file system tree. This mount point ties together a mounted-over v-node (virtual node) and the root of the virtual file system subtree. A mounted-over, or stub, v-node points to a virtual file system, and the mounted VFS points to the v-node it is mounted over.

Understanding Virtual Nodes (V-nodes)

A *virtual node* (v-node) represents access to an object within a virtual file system. V-nodes are used only to translate a path name into a generic node (g-node) (see “Understanding Generic I-nodes (G-nodes)”).

A v-node is either created or used again for every reference made to a file by path name. When a user attempts to open or create a file, if the VFS containing the file already has a v-node representing that file, a use count in the v-node is incremented and the existing v-node is used. Otherwise, a new v-node is created.

Every path name known to the logical file system can be associated with, at most, one file system object. However, each file system object can have several names. Multiple names appear in the following cases:

- The object can appear in multiple virtual file systems. This can happen if the object (or an ancestor) is mounted in different virtual file systems using a local file-over-file or directory-over-directory mount.
- The object does not have a unique name within the virtual file system. (The file system implementation can provide synonyms. For example, the use of links causes files to have more than one name. However, opens of synonymous paths do not cause multiple v-nodes to be created.)

Understanding Generic I-nodes (G-nodes)

A *generic i-node* (g-node) is the representation of an object in a file system implementation. There is a one-to-one correspondence between a g-node and an object in a file system implementation. Each g-node represents an object owned by the file system implementation.

Each file system implementation is responsible for allocating and destroying g-nodes. The g-node then serves as the interface between the logical file system and the file system implementation (see “Logical File System Overview” on page 33). Calls to the file system implementation serve as requests to perform an operation on a specific g-node.

A g-node is needed, in addition to the file system i-node, because some file system implementations may not include the concept of an i-node. Thus the g-node structure substitutes for whatever structure the file system implementation may have used to uniquely identify a file system object.

The logical file system relies on the file system implementation to provide valid data for the following fields in the g-node:

gn_type Identifies the type of object represented by the g-node.
gn_ops Identifies the set of operations that can be performed on the object.

Understanding the Virtual File System Interface

Operations that can be performed upon a virtual file system and its underlying objects are divided into two categories. Operations upon a file system implementation as a whole (not requiring the existence of an underlying file system object) are called **vfs** operations (see “Chapter 3. Virtual File Systems” on page 33). Operations upon the underlying file system objects are called v-node (virtual node) operations (see “Understanding Virtual Nodes (V-nodes)” on page 35). Before writing specific virtual file system operations, it is important to note the requirements for a file system implementation.

Requirements for a File System Implementation

File system implementations differ in how they implement the predefined operations. However, the logical file system expects that a file system implementation meets the following criteria:

- All **vfs** and v-node operations must supply a return value:
 - A return value of 0 indicates the operation was successful.
 - A nonzero return value is interpreted as a valid error number (taken from the `/usr/include/sys/errno.h` file) and returned through the system call interface to the application program.
- All **vfs** operations must exist for each file system type, but can return an error upon startup. The following are the necessary **vfs** operations:
 - **vfs_cntl**
 - **vfs_mount**
 - **vfs_root**
 - **vfs_statfs**
 - **vfs_sync**
 - **vfs_unmount**
 - **vfs_vget**

Important Data Structures for a File System Implementation

There are two important data structures used to represent information about a virtual file system, the **vfs** structure and the v-node. Each virtual file system has a

vfs structure in memory that describes its type, attributes, and position in the file tree hierarchy. Each file object within that virtual file system can be represented by a v-node.

The **vfs** structure contains the following fields:

vfs_flag	Contains the state flags: VFS_DEVMOUNT Indicates whether the virtual file system has a physical mount structure underlying it. VFS_READONLY Indicates whether the virtual file system is mounted read-only.
vfs_type	Identifies the type of file system implementation. Possible values for this field are described in the <code>/usr/include/sys/vmount.h</code> file.
vfs_ops	Points to the set of operations for the specified file system type.
vfs_mntdover	Points to the mounted-over v-node.
vfs_data	Points to the file system implementation data. The interpretation of this field is left to the discretion of the file system implementation. For example, the field could be used to point to data in the kernel extension segment or as an offset to another segment.
vfs_mdata	Records the user arguments to the <code>mount</code> call that created this virtual file system. This field has a time stamp. The user arguments are retained to implement the <code>mntctl</code> call, which replaces the <code>/etc/mnttab</code> table.

Understanding Data Structures and Header Files for Virtual File Systems

These are the data structures used in implementing virtual file systems:

- The **vfs** structure contains information about a virtual file system as a single entity (see “Important Data Structures for a File System Implementation” on page 36).
- The **vnnode** structure contains information about a file system object in a virtual file system (see “Understanding Virtual Nodes (V-nodes)” on page 35). There can be multiple v-nodes for a single file system object.
- The **gnode** structure contains information about a file system object in a physical file system (see “Understanding Generic I-nodes (G-nodes)” on page 35). There is only a single g-node for a given file system object.
- The **gfs** structure contains information about a file system implementation. This is distinct from the **vfs** structure, which contains information about an instance of a virtual file system.

The header files contain the structure definitions for the key components of the virtual file system abstraction. Understanding the contents of these files and the relationships between them is essential to an understanding of virtual file systems. The following are the necessary header files:

- `sys/vfs.h`
- `sys/gfs.h`
- `sys/vnode.h`
- `sys/vmount.h`

Configuring a Virtual File System

The kernel maintains a table of active file system types (see “Chapter 3. Virtual File Systems” on page 33). A file system implementation must be registered with the kernel before a request to mount a virtual file system (VFS) of that type can be honored (see “Virtual File System Overview” on page 35). Two kernel services, **gfsadd** and **gfsdel**, are supplied for adding a file system type to the **gfs** file system table.

These are the steps that must be followed to get a file system configured.

1. A user-level routine must call the **sysconfig** subroutine requesting that the code for the virtual file system be loaded.
2. The user-level routine must then request, again by calling the **sysconfig** subroutine, that the virtual file system be configured. The name of a VFS-specific configuration routine must be specified.
3. The virtual file system-specific configuration routine calls the **gfsadd** kernel service to have the new file system added to the **gfs** table. The **gfs** table that the configuration routine passes to the **gfsadd** kernel service contains a pointer to an initialization routine. This routine is then called to do any further virtual file system-specific initialization.
4. The file system is then operational.

Chapter 4. Kernel Services

Kernel services are routines that provide the runtime kernel environment to programs executing in kernel mode. Kernel extensions call kernel services, which resemble library routines. In contrast, application programs call library routines.

Callers of kernel services execute in kernel mode. They therefore share with the kernel the responsibility for ensuring that system integrity is not compromised.

“System Calls Available to Kernel Extensions” on page 31 lists the system calls that kernel extensions are allowed to use.

Categories of Kernel Services

Following are the categories of kernel services:

- “I/O Kernel Services”
- “Kernel Extension and Device Driver Management Kernel Services” on page 46
- “Locking Kernel Services” on page 48
- “Logical File System Kernel Services” on page 51
- “Memory Kernel Services” on page 53
- “Message Queue Kernel Services” on page 60
- “Network Kernel Services” on page 61
- “Process and Exception Management Kernel Services” on page 63
- “RAS Kernel Services” on page 66
- “Security Kernel Services” on page 67
- “Timer and Time-of-Day Kernel Services” on page 67
- “Virtual File System (VFS) Kernel Services” on page 69

I/O Kernel Services

The I/O kernel services fall into the following categories:

- “Buffer Cache Kernel Services” on page 40
- “Character I/O Kernel Services” on page 40
- “Interrupt Management Services” on page 40
- “Memory Buffer (mbuf) Kernel Services” on page 41
- “DMA Management Kernel Services” on page 42

Block I/O Kernel Services

The Block I/O kernel services are:

- | | |
|----------------|---|
| iodone | Performs block I/O completion processing. |
| iowait | Waits for block I/O completion. |
| uphysio | Performs character I/O for a block device using a uio structure. |

Buffer Cache Kernel Services

The “Block I/O Buffer Cache Kernel Services: Overview” on page 42 describes how to manage the buffer cache with the Buffer Cache kernel services. The Buffer Cache kernel services are:

bawrite	Writes the specified buffer’s data without waiting for I/O to complete.
bdwrite	Releases the specified buffer after marking it for delayed write.
bflush	Flushes all write-behind blocks on the specified device from the buffer cache.
binval	Invalidates all of the specified device’s blocks in the buffer cache.
blkflush	Flushes the specified block if it is in the buffer cache.
bread	Reads the specified block’s data into a buffer.
breada	Reads in the specified block and then starts I/O on the read-ahead block.
brelease	Frees the specified buffer.
bwrite	Writes the specified buffer’s data.
clrbuf	Sets the memory for the specified buffer structure’s buffer to all zeros.
getblk	Assigns a buffer to the specified block.
geteblk	Allocates a free buffer.
geterror	Determines the completion status of the buffer.
purblk	Purges the specified block from the buffer cache.

Character I/O Kernel Services

The Character I/O kernel services are:

getc	Retrieves a character from a character list.
getcb	Removes the first buffer from a character list and returns the address of the removed buffer.
getcbp	Retrieves multiple characters from a character buffer and places them at a designated address.
getcf	Retrieves a free character buffer.
getc	Returns the character at the end of a designated list.
pincl	Manages the list of free character buffers.
putc	Places a character at the end of a character list.
putcb	Places a character buffer at the end of a character list.
putcbp	Places several characters at the end of a character list.
putcf	Frees a specified buffer.
putcfl	Frees the specified list of buffers.
putc	Places a character on a character list.
waitcfree	Checks the availability of a free character buffer.

Interrupt Management Services

The operating system provides the following set of kernel services for managing interrupts. See “Understanding Interrupts” on page 44 for a description of these services:

i_clear	Removes an interrupt handler from the system.
i_reset	Resets the system’s hardware interrupt latches.
i_sched	Schedules off-level processing.
i_mask	Disables an interrupt level.
i_unmask	Enables an interrupt level.
i_disable	Disables all of the interrupt levels at a particular interrupt priority and all interrupt levels at a less-favored interrupt priority.

i_enable Enables all of the interrupt levels at a particular interrupt priority and all interrupt levels at a more-favored interrupt priority.

Memory Buffer (mbuf) Kernel Services

The Memory Buffer (mbuf) kernel services provide functions to obtain, release, and manipulate memory buffers, or **mbufs**. These **mbuf** services provide the means to easily work with the **mbuf** data structure, which is defined in the `/usr/include/sys/mbuf.h` file. Data can be stored directly in an **mbuf**'s data portion or in an attached external cluster. **Mbufs** can also be chained together by using the `m_next` field in the **mbuf** structure. This is particularly useful for communications protocols that need to add and remove protocol headers.

The Memory Buffer (**mbuf**) kernel services are:

m_adj	Adjusts the size of an mbuf chain.
m_clattach	Allocates an mbuf structure and attaches an external cluster.
m_cat	Appends one mbuf chain to the end of another.
m_clgetm	Allocates and attaches an external buffer.
m_collapse	Guarantees that an mbuf chain contains no more than a given number of mbuf structures.
m_copydata	Copies data from an mbuf chain to a specified buffer.
m_copym	Creates a copy of all or part of a list of mbuf structures.
m_dereg	Deregisters expected mbuf structure usage.
m_free	Frees an mbuf structure and any associated external storage area.
m_freem	Frees an entire mbuf chain.
m_get	Allocates a memory buffer from the mbuf pool.
m_getclr	Allocates and zeros a memory buffer from the mbuf pool.
m_getclustm	Allocates an mbuf structure from the mbuf buffer pool and attaches a cluster of the specified size.
m_ghdr	Allocates a header memory buffer from the mbuf pool.
m_pullup	Adjusts an mbuf chain so that a given number of bytes is in contiguous memory in the data area of the head mbuf structure.
m_reg	Registers expected mbuf usage.

In addition to the **mbuf** kernel services, the following macros are available for use with **mbufs**:

m_clget	Allocates a page-sized mbuf structure cluster.
m_copy	Creates a copy of all or part of a list of mbuf structures.
m_getclust	Allocates an mbuf structure from the mbuf buffer pool and attaches a page-sized cluster.
M_HASCL	Determines if an mbuf structure has an attached cluster.
DTOM	Converts an address anywhere within an mbuf structure to the head of that mbuf structure.
MTOCL	Converts a pointer to an mbuf structure to a pointer to the head of an attached cluster.
MTOD	Converts a pointer to an mbuf structure to a pointer to the data stored in that mbuf structure.
M_XMEMD	Returns the address of an mbuf cross-memory descriptor.

DMA Management Kernel Services

The operating system kernel provides several services for managing direct memory access DMA channels and performing DMA operations. “Understanding DMA Transfers” on page 44 provides additional kernel services information.

The services provided are:

d_align	Assists in the allocation of DMA buffers.
d_cflush	Flushes the processor and I/O controller (IOCC) data caches when using the long term DMA_WRITE_ONLY mapping of direct memory access (DMA) buffers approach to the bus device DMA.
d_clear	Frees a DMA channel.
d_complete	Cleans up after a DMA transfer.
d_init	Initializes a DMA channel.
d_map_init	Allocates and initializes resources for performing DMA with PCI and ISA devices.
d_mask	Disables a DMA channel.
d_master	Initializes a block-mode DMA transfer for a DMA master.
d_move	Provides consistent access to system memory accessed asynchronously by a device and the processor on the system.
d_roundup	Assists in allocation of DMA buffers.
d_slave	Initializes a block-mode DMA transfer for a DMA slave.
d_unmask	Enables a DMA channel.

Block I/O Buffer Cache Kernel Services: Overview

The Block I/O Buffer Cache services are provided to support user access to device drivers through block I/O special files. This access is required by the operating system file system for mounts and other limited activity, as well as for compatibility services required when other file systems are installed on these kinds of systems. These services are not used by the operating system’s JFS (journal file system), NFS (Network File System), or CDRFS (CD-ROM file system) when processing standard file I/O data. Instead they use the virtual memory manager and pager to manage the system’s memory pages as a buffer cache.

For compatibility support of other file systems and block special file support, the buffer cache services serve two important purposes:

- They ensure that multiple processes accessing the same block of the same device in multiprogrammed fashion maintain a consistent view of the data in the block.
- They increase the efficiency of the system by keeping in-memory copies of blocks that are frequently accessed.

The Buffer Cache services use the **buf** structure or buffer header as their main data-tracking mechanism. Each buffer header contains a pair of pointers that maintains a doubly-linked list of buffers associated with a particular block device. An additional pair of pointers maintain a doubly-linked list of blocks available for use again on another operation. Buffers that have I/O in progress or that are busy for other purposes do not appear in this available list.

Kernel buffers are discussed in more detail in “Introduction to Kernel Buffers” in *AIX Version 4.3 Technical Reference: Kernel and Subsystems Volume 1*.

See “Block I/O Kernel Services” on page 39 for a list of these services.

Managing the Buffer Cache

Fourteen kernel services provide management of this block I/O buffer cache mechanism. The **getblk** kernel service allocates a buffer header and a free buffer from the buffer pool. Given a device and block number, the **getblk** and **bread** kernel services both return a pointer to a buffer header for the block. But the **bread** service is guaranteed to return a buffer actually containing a current data for the block. In contrast, the **getblk** service returns a buffer that contains the data in the block only if it is already in memory.

In either case, the buffer and the corresponding device block are made busy. Other processes attempting to access the buffer must wait until it becomes free. The **getblk** service is used when:

- A block is about to be rewritten totally.
- Its previous contents are not useful.
- No other processes should be allowed to access it until the new data has been placed into it.

The **breada** kernel service is used to perform read-ahead I/O and is similar to the **bread** service except that an additional parameter specifies the number of the block on the same device to be read asynchronously after the requested block is available. The **brelease** kernel service makes the specified buffer available again to other processes.

Using the Buffer Cache write Services

There are three slightly different write routines. All of them take a buffer pointer as a parameter and all logically release the buffer by placing it on the free list. The **bwrite** service puts the buffer on the appropriate device queue by calling the device's strategy routine. The **bwrite** service then waits for I/O completion and sets the caller's error flag, if required. This service is used when the caller wants to be sure that I/O takes place synchronously, so that any errors can be handled immediately.

The **bawrite** service is an asynchronous version of the **bwrite** service and does not wait for I/O completion. This service is normally used when the overlap of processing and device I/O activity is desired.

The **bdwrite** service does not start any I/O operations, but marks the buffer as a delayed write and releases it to the free list. Later, when the buffer is obtained from the free list and found to contain data from some other block, the data is written out to the correct device before the buffer is used. The **bdwrite** service is used when it is undetermined if the write is needed immediately.

For example, the **bdwrite** service is called when the last byte of the write operation associated with a block special file falls short of the end of a block. The **bdwrite** service is called on the assumption that another write will soon occur that will use the same block again. On the other hand, as the end of a block is passed, the **bawrite** service is called, because it is assumed the block will not be accessed again soon. Therefore, the I/O processing can be started as soon as possible.

Note that the **getblk** and **bread** services dedicated the specified block to the caller while making other processes wait, while the **brelease**, **bwrite**, **bawrite**, or **bdwrite** services must eventually be called to free the block for use by other processes.

Understanding Interrupts

Each hardware interrupt has an interrupt level and an interrupt priority. The interrupt level defines the source of the interrupt. There are basically two types of interrupt levels: system and bus. The system bus interrupts are generated from the Micro Channel bus and system I/O. Examples of system interrupts are the timer and serial link interrupts.

The interrupt level of a system interrupt is defined in the `sys/intr.h` file. The interrupt level of a bus interrupt is one of the resources managed by the bus configuration methods.

Interrupt Priorities

The interrupt priority defines which of a set of pending interrupts is serviced first. INTMAX is the most favored interrupt priority and INTBASE is the least favored interrupt priority. The interrupt priorities for bus interrupts range from INTCLASS0 to INTCLASS3. The rest of the interrupt priorities are reserved for the base kernel. Interrupts that cannot be serviced within the time limits specified for bus interrupts qualify as off-level interrupts.

A device's interrupt priority is selected based on two criteria: its maximum interrupt *latency* requirements and the device driver's interrupt *execution time*. The interrupt latency requirement is the maximum time within which an interrupt must be serviced. (If it is not serviced in this time, some event is lost or performance is degraded seriously.) The interrupt execution time is the number of machine cycles required by the device driver to service the interrupt. Interrupts with a short interrupt latency time must have a short interrupt service time.

The general rule for interrupt service times is based on the following interrupt priority table:

Interrupt Priority Versus Interrupt Service Times

Priority	Service Time (machine cycles)
INTCLASS0	200 cycles
INTCLASS1	400 cycles
INTCLASS2	600 cycles
INTCLASS3	800 cycles

The valid interrupt priorities are defined in the `/usr/include/sys/intr.h` file.

See "Interrupt Management Services" on page 40 for a list of these services.

Understanding DMA Transfers

A device driver must call the `d_slave` service to set up a DMA slave transfer or call the `d_master` service to set up a DMA master transfer. The device driver then sets up the device to perform the DMA transfer. The device transfers data when it is available and interrupts the processor upon completion of the DMA transfer. The device driver then calls the `d_complete` service to clean up after the DMA transfer. This process is typically repeated each time a DMA transfer is to occur.

Hiding DMA Data

In this system, data can be located in the processor cache, system memory, or DMA buffer. The DMA services have been carefully written to ensure that data is moved between these three locations correctly. The **d_master** and **d_slave** services flush the data from the processor cache to system memory. They then hide the page, preventing data from being placed back into the processor cache. The hardware moves the data between system memory, the DMA buffers, and the device. The **d_complete** service flushes data from the DMA buffers to system memory and unhides the buffer.

A count is maintained of the number of times a page is hidden for DMA. A page is not actually hidden except when the count goes from 0 to 1 and is not unhidden except when the count goes from 1 to 0. Therefore, the users of the services must make sure to have the same number of calls to both the **d_master** and **d_complete** services. Otherwise, the page can be incorrectly unhidden and data lost. This count is intended to support operations such as logical volume mirrored writes.

All pages containing user data must be hidden while DMA operations are being performed on them. This is required to ensure that data is not lost by being put in more than one of these locations.

DMA operations can be carefully performed on kernel data without hiding the pages containing the data. The **DMA_WRITE_ONLY** flag, when specified to the **d_master** service, causes it *not* to flush the processor cache or hide the pages. The same flag when specified to the **d_complete** service causes it *not* to unhide the pages. This flag requires that the caller has carefully flushed the processor cache using the **vm_cflush** service. Additionally, the caller must carefully allocate complete pages for the data buffer and carefully split them up into transfers. Transferred pages must each be aligned at the start of a DMA buffer boundary, and no other data can be in the same DMA buffers as the data to be transferred. The **d_align** and **d_roundup** services help ensure that the buffer allocation is correct.

The **d_align** service (provided in **libsys.a**) returns the alignment value required for starting a buffer on a processor cache line boundary. The **d_roundup** service (also provided in **libsys.a**) can be used to round the desired DMA buffer length up to a value that is an integer number of cache lines. These two services allow buffers to be used for DMA to be aligned on a cache line boundary and allocated in whole multiples of the cache line size so that the buffer is not split across processor cache lines. This reduces the possibility of consistency problems because of DMA and also minimizes the number of cache lines that must be flushed or invalidated when used for DMA. For example, these services can be used to provide alignment as follows:

```
align = d_align();
buffer_length = d_roundup(required_length);
buf_ptr = xmalloc(buffer_length, align, kernel_heap);
```

Note: If the kernel heap is used for DMA buffers, the buffer must be pinned using the **pin** kernel service before being utilized for DMA. Alternately, the memory could be requested from the pinned heap.

Accessing Data While the DMA Operation Is in Progress

Data must be carefully accessed when a DMA operation is in progress. The **d_move** service provides a means of accessing the data while a DMA transfer is

being performed on it. This service accesses the data through the same system hardware as that used to perform the DMA transfer. The `d_move` service, therefore, cannot cause the data to become inconsistent. This service can also access data hidden from normal processor accesses.

See “DMA Management Kernel Services” on page 42 for a list of these services.

Kernel Extension and Device Driver Management Kernel Services

The kernel provides a relatively complete set of program and device driver management services. These services include general kernel extension loading and binding services and device driver binding services. Also provided are services that allow kernel extensions to be notified of base kernel configuration changes, user-mode exceptions, and system-wide process state changes.

Kernel Extension Loading and Binding Services

The `kmod_load`, `kmod_entrypt`, and `kmod_unload` services provide kernel extension loading and binding services. The `sysconfig` subroutine makes these services available to user-mode programs. However, kernel-mode callers executing in a kernel process environment can also use them. These services provide the same kernel object-file load, unload, and query functions provided by the `sysconfig` subroutine as well as the capability to obtain a module’s entry point with the kernel module ID assigned to the module.

The `kmod_load`, `kmod_entrypt`, and `kmod_unload` services can be used to dynamically alter the set of routines loaded into the kernel based on system configuration and application demand. Subsystems and device drivers can use these services to load large, seldom-used routines on demand. Device driver binding services include the `devswadd`, `devswdel`, `devswqry` services, which are used to add or remove a device driver entry from the dynamically managed device switch table. They also query for information concerning a specific entry in the device switch table.

Other Functions for the Kernel Extension and Device Driver Management Services

Some kernel extensions may be sensitive to the settings of base kernel runtime configurable parameters that are found in the `var` structure defined in the `/usr/include/sys/var.h` file. These parameters can be set during system boot or runtime by a privileged user performing system configuration commands that use the `sysconfig` subroutine to alter values in the `var` structure. Kernel extensions may register or remove a configuration notification routine with the `cfgnadd` and `cfgndel` kernel services. This routine is called each time the `sysconfig` subroutine is used to change base kernel tunable parameters found in the `var` structure.

In addition, the `prochadd` and `prochdel` kernel services allow kernel extensions to be notified when any process in the system has a state transition, such as being created, exiting, being swapped in or swapped out. The `uexadd` and `uexdel` kernel services give kernel extensions the capability to intercept user-mode exceptions. These user-mode exception handlers may use this capability to dynamically reassign access to single-use resources or to clean up after some particular user-mode error. The associated `uexblock` and `uexclear` services can be used by these handlers to block and resume process execution when handling these exceptions.

The **pio_assist** and **getexcept** kernel services are typically used by device drivers to obtain detailed information about exceptions that occur during I/O bus access. The **getexcept** service can also be used by any exception handler requiring more information about an exception that has occurred. The **selreg** kernel service is used by file select operations to register unsatisfied asynchronous poll or select event requests with the kernel. The **selnotify** kernel service replaces the traditional operating system's **selwakeup** kernel function and is used by device drivers supporting the **poll** or **select** subroutines when asynchronous event notification is requested. The **iostadd** and **iostdel** services are used by tty and disk device drivers to register device activity reporting structures to be used by the **iostat** and **vmstat** commands.

Finally, the **getuerror** and **setuerror** services can be used by kernel extensions that provide or use system calls to access the `u_t_error` field for the current process thread's **uthread** structure. This is typically used by kernel extensions providing system calls to return error codes, and is used by other kernel extensions to check error codes upon return from a system call (since there is no **errno** global variable in the kernel).

List of Kernel Extension and Device Driver Management Kernel Services

The Kernel Program/Device Driver Management kernel services are:

cfgnadd	Registers a notification routine to be called when system-configurable variables are changed.
cfgndel	Removes a notification routine for receiving broadcasts of changes to system configurable variables.
devdump	Calls a device driver dump-to-device routine.
devstrat	Calls a block device driver's strategy routine.
devswadd	Adds a device entry to the device switch table.
devswchg	Alters a device switch entry point in the device switch table.
devswdel	Deletes a device driver entry from the device switch table.
devswqry	Checks the status of a device switch entry in the device switch table.
getexcept	Allows kernel exception handlers to retrieve additional exception information.
getuerror	Allows kernel extensions to retrieve the current value of the <code>u_error</code> field.
iostadd	Registers an I/O statistics structure used for updating I/O statistics reported by the iostat subroutine.
iostdel	Removes the registration of an I/O statistics structure used for maintaining I/O statistics on a particular device.
kmod_entrypt	Returns a function pointer to a kernel module's entry point.
kmod_load	Loads an object file into the kernel or queries for an object file already loaded.
kmod_unload	Unloads a kernel object file.
pio_assist	Provides a standardized programmed I/O exception handling mechanism for all routines performing programmed I/O.
prochadd	Adds a systemwide process state-change notification routine.
prochdel	Deletes a process state change notification routine.
selreg	Registers an asynchronous poll or select request with the kernel.
selnotify	Wakes up processes waiting in a poll or select subroutine or the fp_poll kernel service.
setuerror	Allows kernel extensions to set the <code>u_error</code> field in the <code>u</code> area.
uexadd	Adds a systemwide exception handler for catching user-mode process exceptions.

uexblock	Makes a process nonrunnable when called from a user-mode exception handler.
uexcLEAR	Makes a process blocked by the uexblock service runnable again.
uexdel	Deletes a previously added system-wide user-mode exception handler.

Locking Kernel Services

The following information is provided to assist you in understanding the locking kernel services:

- “Lock Allocation and Other Services”
- “Simple Locks”
- “Complex Locks” on page 49
- “Lockl Locks” on page 50
- “Atomic Lock Operations” on page 50
- “Atomic Operations” on page 51

Lock Allocation and Other Services

The following lock allocation services allocate and free internal operating system memory for simple and complex locks, or check if the caller owns a lock:

lock_alloc	Allocates system memory for a simple or complex lock.
lock_free	Frees the system memory of a simple or complex lock.
lock_mine	Checks whether a simple or complex lock is owned by the caller.

Simple Locks

Simple locks are exclusive-write, non-recursive locks which protect thread-thread or thread-interrupt critical sections. Simple locks are preemptable, meaning that a kernel thread can be preempted by another, higher priority kernel thread while it holds a simple lock. The simple lock kernel services are:

simple_lock_init	Initializes a simple lock.
simple_lock, simple_lock_try	Locks a simple lock.
simple_unlock	Unlocks a simple lock.

On a multiprocessor system, simple locks which protect thread-interrupt critical sections must be used in conjunction with interrupt control in order to serialize execution both within the executing processor and between different processors. On a uniprocessor system interrupt control is sufficient; there is no need to use locks. The following kernel services provide appropriate locking calls for the system on which they are executed:

disable_lock	Raises the interrupt priority, and locks a simple lock if necessary.
unlock_enable	Unlocks a simple lock if necessary, and restores the interrupt priority.

Using the **disable_lock** and **unlock_enable** kernel services to protect thread-interrupt critical sections (instead of calling the underlying interrupt control and locking kernel services directly) ensures that multiprocessor-safe code does not make unnecessary locking calls on uniprocessor systems.

Simple locks are spin locks; a kernel thread which attempts to acquire a simple lock may spin (busy-wait: repeatedly execute instructions which do nothing) if the lock is not free. The table shows the behavior of kernel threads and interrupt handlers which attempt to acquire a busy simple lock.

Result of Attempting to Acquire a Busy Simple Lock		
Caller	Owner is Running	Owner is Sleeping
Thread (with interrupts enabled)	Caller spins initially; it sleeps if the maximum spin threshold is crossed.	Caller sleeps immediately.
Interrupt handler or thread (with interrupts disabled)	Caller spins until lock is freed.	Caller spins until lock is freed (must not happen).

Note: On uniprocessor systems, the maximum spin threshold is set to one, meaning that a kernel thread will never spin waiting for a lock.

A simple lock that protects a thread-interrupt critical section must never be held across a sleep, otherwise the interrupt could spin for the duration of the sleep, as shown in the table. This means that such a routine must not call any external services which may result in a sleep. In general, using any kernel service which is callable from process level may result in a sleep, as can accessing unpinned data. These restrictions do not apply to simple locks that protect thread-thread critical sections.

The lock word of a simple lock must be located in pinned memory if simple locking services are called with interrupts disabled.

Complex Locks

Complex locks are read-write locks which protect thread-thread critical sections. Complex locks are preemptable, meaning that a kernel thread can be preempted by another, higher priority kernel thread while it holds a complex lock. The complex lock kernel services are:

lock_init	Initializes a complex lock.
lock_islocked	Tests whether a complex lock is locked.
lock_done	Unlocks a complex lock.
lock_read, lock_try_read	Locks a complex lock in shared-read mode.
lock_read_to_write, lock_try_read_to_write	Upgrades a complex lock from shared-read mode to exclusive-write mode.
lock_write, lock_try_write	Locks a complex lock in exclusive-write mode.
lock_write_to_read	Downgrades a complex lock from exclusive-write mode to shared-read mode.
lock_set_recursive	Prepares a complex lock for recursive use.
lock_clear_recursive	Prevents a complex lock from being acquired recursively.

By default, complex locks are not recursive (they cannot be nested). A complex lock can become recursive through the **lock_set_recursive** kernel service. A recursive complex lock is not freed until **lock_done** is called once for each time that the lock was locked.

Complex locks are spin locks; a kernel thread which attempts to acquire a complex lock may spin (busy-wait: repeatedly execute instructions which do nothing) if the lock is not free. The table shows the behavior of kernel threads which attempt to acquire a busy complex lock:

Result of Attempting to Acquire a Busy Complex Lock		
Current Lock Mode	Owner is Running	Owner is Sleeping
Exclusive-write	Caller spins initially, but sleeps if the maximum spin threshold is crossed, or if the owner later sleeps.	Caller sleeps immediately.
Shared-read being acquired for exclusive-write	Caller spins initially; it sleeps if the maximum spin threshold is crossed.	
Shared-read being acquired for shared-read	Lock granted immediately	

Notes:

1. On uniprocessor systems, the maximum spin threshold is set to one, meaning that a kernel thread will never spin waiting for a lock.
2. The concept of a single owner does not apply to a lock held in shared-read mode.

Lockl Locks

Note: Lockl locks (previously called conventional locks) are only provided to ensure compatibility with existing code. New code should use simple or complex locks.

Lockl locks are exclusive-access and recursive locks. The lockl lock kernel services are:

lockl Locks a conventional lock.
unlockl Unlocks a conventional lock.

A thread which tries to acquire a busy lockl lock sleeps immediately.

The lock word of a lockl lock must be located in pinned memory if the lockl service is called with interrupts disabled.

Atomic Lock Operations

Atomic lock operations are services that read or write single word variables; on multiprocessor systems, they also protect against concurrent access using *import and export fences* (or synchronization instructions). They can be used to implement higher level locking services and are mainly intended to support the building of user mode lock services when POSIX 1003.1c mutexes are not appropriate. Thus, the following atomic lock operations are provided as user subroutines:

_check_lock Conditionally updates a single word variable atomically, issuing an *import fence* for multiprocessor systems. The kernel service **compare_and_swap** is similar, but does not issue an import fence, and therefore is inappropriate for updates of lock words on multiprocessor systems.

<code>_clear_lock</code>	Atomically writes a single word variable, issuing an <i>export fence</i> for multiprocessor systems.
<code>_safe_fetch</code>	Atomically reads and returns a single word variable protected by an <i>export fence</i> . The read is safe for multiprocessor systems.

Single word variables accessed by atomic lock operations must be aligned on a full word boundary, and must be located in pinned memory if called with interrupts disabled.

Atomic Operations

Atomic operations are sequences of instructions which guarantee atomic accesses and updates of shared single word variables. This means that atomic operations cannot protect accesses to complex data structures in the way that locks can, but they provide a very efficient way of serializing access to a single word.

The atomic operation kernel services are:

<code>fetch_and_add</code>	Increments a single word variable atomically.
<code>fetch_and_and</code> , <code>fetch_and_or</code>	Manipulates bits in a single word variable atomically.
<code>compare_and_swap</code>	Conditionally updates or returns a single word variable atomically.

Single word variables accessed by atomic operations must be aligned on a full word boundary, and must be located in pinned memory if atomic operation kernel services are called with interrupts disabled.

File Descriptor Management Services

The File Descriptor Management services are supplied by the logical file system for creating, using, and maintaining file descriptors. These services allow for the implementation of system calls that use a file descriptor as a parameter, create a file descriptor, or return file descriptors to calling applications. The following are the File Descriptor Management services:

<code>ufdcreate</code>	Allocates and initializes a file descriptor.
<code>ufdhold</code>	Increments the reference count on a file descriptor.
<code>ufdrele</code>	Decrements the reference count on a file descriptor.
<code>ufdgetf</code>	Gets a file structure pointer from a held file descriptor.
<code>getufdflags</code>	Gets the flags from a file descriptor.
<code>setufdflags</code>	Sets flags in a file descriptor.

Logical File System Kernel Services

The Logical File System services (also known as the `fp_services`) allow processes running in kernel mode to open and manipulate files in the same way that user-mode processes do. Data access limitations make it unreasonable to accomplish these tasks with system calls, so a subset of the file system calls has been provided with an alternate kernel-only interface.

The Logical File System services are one component of the logical file system, which provides the functions required to map system call requests to virtual file system requests. The logical file system is responsible for resolution of file names and file descriptors. It tracks all open files in the system using the file table. The Logical File System services are lower level entry points into the system call support within the logical file system.

Routines in the kernel that must access data stored in files or that must set up paths to devices are the primary users of these services. This occurs most commonly in device drivers, where a lower level device driver must be accessed or where the device requires microcode to be downloaded. Use of the Logical File System services is not, however, restricted to these cases.

A process can use the Logical File System services to establish access to a file or device by calling:

- The **fp_open** service with a path name to the file or device it must access.
- The **fp_opendev** service with the device number of a device it must access.
- The **fp_getf** service with a file descriptor for the file or device. If the process wants to retain access past the duration of the system call, it must then call the **fp_hold** service to acquire a private file pointer.

These three services return a file pointer that is needed to call the other Logical File System services. The other services provide the functions that are provided by the corresponding system calls.

Other Considerations

The Logical File System services are available only in the process environment (see “Process Environment” on page 8).

In addition, calling the **fp_open** service at certain times can cause a deadlock. The lookup on the file name must acquire file system locks. If the process is already holding any lock on a component of the path, the process will be deadlocked. Therefore, do not use the **fp_open** service when the process is already executing an operation that holds file system locks on the requested path. The operations most likely to cause this condition are those that create files.

List of Logical File System Kernel Services

These are the Logical File System kernel services:

fp_access	Checks for access permission to an open file.
fp_close	Closes a file.
fp_fstat	Gets the attributes of an open file.
fp_getdevno	Gets the device number or channel number for a device.
fp_getf	Retrieves a pointer to a file structure.
fp_hold	Increments the open count for a specified file pointer.
fp_ioctl	Issues a control command to an open device or file.
fp_lseek	Changes the current offset in an open file.
fp_llseek	Changes the current offset in an open file. Used to access offsets beyond 2GB.
fp_open	Opens a regular file or directory.
fp_opendev	Opens a device special file.
fp_poll	Checks the I/O status of multiple file pointers/descriptors and message queues.
fp_read	Performs a read on an open file with arguments passed.

fp_readv	Performs a read operation on an open file with arguments passed in iovec elements.
fp_rwuio	Performs read or write on an open file with arguments passed in a uio structure.
fp_select	Provides for cascaded, or redirected, support of the select or poll request.
fp_write	Performs a write operation on an open file with arguments passed.
fp_writev	Performs a write operation on an open file with arguments passed in iovec elements.

Memory Kernel Services

The Memory kernel services provide kernel extensions with the ability to:

- Dynamically allocate and free memory
- Pin and unpin code and data
- Access user memory and transfer data between user and kernel memory
- Create, reference, and change virtual memory objects

The following information is provided to assist you in learning more about memory kernel services:

- “Memory Management Kernel Services”
- “Memory Pinning Kernel Services”
- “User Memory Access Kernel Services” on page 54
- “Virtual Memory Management Kernel Services” on page 54
- “Cross-Memory Kernel Services” on page 55

Memory Management Kernel Services

The Memory Management services are:

init_heap	Initializes a new heap to be used with kernel memory management services.
xmalloc	Allocates memory.
xmfree	Frees allocated memory.

Memory Pinning Kernel Services

The Memory Pinning services are:

pin	Pins the address range in the system (kernel) space.
pincode	Pins the code and data associated with an object file.
pinu	Pins the specified address range in user or system memory.
unpin	Unpins the address range in system (kernel) address space.
unpincode	Unpins the code and data associated with an object file.
unpinu	Unpins the specified address range in user or system memory.
xmempin	Pins the specified address range in user or system memory, given a valid cross-memory descriptor.
xmemunpin	Unpins the specified address range in user or system memory, given a valid cross-memory descriptor.

User Memory Access Kernel Services

In a system call or kernel extension running under a user process, data in the user process can be moved in or out of the kernel using the **copyin** and **copyout** services. The **uiomove** service is used for scatter and gather operations. If user data is to be referenced asynchronously, such as from an interrupt handler or a kernel process, the cross memory services must be used.

The User Memory Access kernel services are:

copyin, copyin64	Copies data between user and kernel memory.
copyinstr, copyinstr64	Copies a character string (including the terminating null character) from user to kernel space.
copyout, copyout64	Copies data between user and kernel memory.
fubyte, fubyte64	Fetches, or retrieves, a byte of data from user memory.
fuword, fuword64	Fetches, or retrieves, a word of data from user memory.
subyte, subyte64	Stores a byte of data in user memory.
suword, suword64	Stores a word of data in user memory.
uiomove	Moves a block of data between kernel space and a space defined by a uio structure.
ureadc	Writes a character to a buffer described by a uio structure.
uwritec	Retrieves a character from a buffer described by a uio structure.

Virtual Memory Management Kernel Services

These services are described in more detail in “Understanding Virtual Memory Manager Interfaces” on page 56. The Virtual Memory Management services are:

as_att, as_att64	Allocates and Maps a specified region in the current user address space.
as_det, as_det64	Unmaps and deallocates a region in the specified address space that was mapped with the as_att or as_att64 kernel service.
as_geth, as_geth64	Obtains a handle to the virtual memory object for the specified address given in the specified address space. The virtual memory object is protected.
as_getsrval, as_getsrval64	Obtains a handle to the virtual memory object for the specified address given in the specified address space.
as_puth as_puth64	Indicates that no more references will be made to a virtual memory object that was obtained using the as_geth or as_geth64 kernel service.
as_seth, as_seth64	Maps a specified region in the specified address space for the specified virtual memory object.
getadsp	Obtains a pointer to the current process’s address space structure for use with the as_att and as_det kernel services.
io_att	Selects, allocates, and maps a region in the current address space for I/O access.
io_det	Unmaps and deallocates the region in the current address space at the given address.
vm_att	Maps a specified virtual memory object to a region in the current address space.
vm_cflush	Flushes the processor’s cache for a specified address range.
vm_det	Unmaps and deallocates the region in the current address space that contains a given address.

vm_handle	Constructs a virtual memory handle for mapping a virtual memory object with specified access level.
vm_makep	Makes a page in client storage.
vm_mount	Adds a file system to the paging device table.
vm_move	Moves data between a virtual memory object and a buffer specified in the uio structure.
vm_protectp	Sets the page protection key for a page range.
vm_qmodify	Determines whether a mapped file has been changed.
vm_release	Releases virtual memory resources for the specified address range.
vm_releasep	Releases virtual memory resources for the specified page range.
vm_uiomove	Moves data between a virtual memory object and a buffer specified in the uio structure.
vm_umount	Removes a file system from the paging device table.
vm_write	Initiates page-out for a page range in the address space.
vm_writep	Initiates page-out for a page range in a virtual memory object.
vms_create	Creates a virtual memory object of the type and size and limits specified.
vms_delete	Deletes a virtual memory object.
vms_iowait	Waits for the completion of all page-out operations for pages in the virtual memory object.

Cross-Memory Kernel Services

The cross-memory services allow data to be moved between the kernel and an address space other than the current process address space. A data area within one region of an address space is attached by calling the **xmattach** or **xmattach64** service. As a result, the virtual memory object cannot be deleted while data is being moved in or out of pages belonging to it. A cross-memory descriptor is filled out by the **xmattach** or **xmattach64** service. The attach operation must be done while under a process. When the data movement is completed, the **xmdetach** service can be called. The detach operation can be done from an interrupt handler.

The **xmemin** service can be used to transfer data from an address space to kernel space. The **xmemout** service can be used to transfer data from kernel space to an address space. These routines may be called from interrupt handler level routines if the referenced buffers are in memory.

Cross-memory services provide the **xmemdma** or **xmemdma64** service to prepare a page for DMA processing. The **xmemdma** and **xmemdma64** services flush any data from cache into memory and hides the page. They do this by making processor access to the page not valid. Any processor references to the page result in page faults with the referencing process waiting on the page to be unhidden. The **xmemdma** or **xmemdma64** service returns the real address of the page for use in preparing DMA address lists. When the DMA transfer is completed, the **xmemdma** or **xmemdma64** service must be called again to unhide the page.

The **xmemdma64** service is identical to the cache-consistent version of **xmemdma**, except that it returns a 64-bit real address. The **xmemdma64** service can be called from the process or interrupt environments. It is also present on 32-bit PowerPC platforms to allow a single device driver or kernel extension binary to work on 32-bit or 64-bit platforms with no change and no run-time checks.

Data movement by DMA or an interrupt handler requires that the pages remain in memory. This is ensured by pinning the data areas using the **pinu** service. This can only be done under a process, since the memory pinning services page-fault on pages not present in memory.

The **unpinu** service unpins pinned pages. This can be done by an interrupt handler if the data area is the global kernel address space. It must be done under the process if the data area is in user process space.

The Cross-Memory services are:

xmattach	Attaches to a user buffer for cross-memory operations.
xmdetach	Detaches from a user buffer used for cross-memory operations.
xmemin	Performs a cross-memory move by copying data from the specified address space to kernel global memory.
xmemout	Performs a cross-memory move by copying data from kernel global memory to a specified address space.
xmemdma	Prepares a page for DMA I/O or processes a page after DMA I/O is complete.
xmemdma64	Prepares a page for DMA I/O or processes a page after DMA I/O is complete. Returns 64-bit real address.

Understanding Virtual Memory Manager Interfaces

The virtual memory manager supports functions that allow a wide range of kernel extension data operations.

Virtual Memory Objects

A *virtual memory object* is an abstraction for the contiguous data that can be mapped into a region of an address space. As a data object, it is independent of any address space. The data it represents can be in memory or on an external storage device. The data represented by the virtual memory object can be shared by mapping the virtual memory object into each address space sharing the access, with the access capability of each mapping represented in that address space map.

File systems use virtual memory objects so that the files can be referenced using a mapped file access method. The map file access method represents the data through a virtual memory object, and allows the virtual memory manager to handle page faults on the mapped file. When a page fault occurs, the virtual memory manager calls the services supplied by the service provider (such as a virtual file system) to get and put pages. A data provider (such as a file system) maintains any data structures necessary to map between the virtual memory object offset and external storage addressing.

The data provider creates a virtual memory object when it has a request for access to the data. It deletes the virtual memory object when it has no more clients referencing the data in the virtual memory object.

The **vms_create** service is called to create virtual memory objects. The **vms_delete** service is called to delete virtual memory objects.

Addressing Data

Data in a virtual memory object is made addressable in user or kernel processes through the **shmat** subroutine. A kernel extension uses the **vm_att** kernel service to select and allocate a region in the current (per-process kernel) address space.

The per-process kernel address space initially sees only global kernel memory and the per-process kernel data. The **vm_att** service allows kernel extensions to allocate additional regions. However, this augmented per-process kernel address space does not persist across system calls. The additional regions must be re-allocated with each entry into the kernel protection domain.

The **vm_att** service takes as an argument a virtual memory handle representing the virtual memory object and the access capability to be used. The **vm_handle** service constructs the virtual memory handles.

When the kernel extension has finished processing the data mapped into the current address space, it should call the **vm_det** service to deallocate the region and remove access.

Moving Data to or from a Virtual Memory Object

A data provider (such as a file system) can call the **vm_makep** service to cause a memory page to be instantiated. This permits a page of data to be moved into a virtual memory object page without causing the virtual memory manager to page in the previous data contents from an external source. This is an operation on the virtual memory object, not an address space range.

The **vm_move** and **vm_uiomove** kernel services move data between a virtual memory object and a buffer specified in a **uio** structure. This allows data providers (such as a file system) to move data to or from a specified buffer to a designated offset in a virtual memory object. This service is similar to **uiomove** service, but the trusted buffer is replaced by the virtual memory object, which need not be currently addressable.

Data Flushing

A kernel extension can initiate the writing of a data area to external storage with the **vm_write** kernel service, if it has addressability to the data area. The **vm_writep** kernel service can be used if the virtual memory object is not currently addressable.

If the kernel extension needs to ensure that the data is moved successfully, it can wait on the I/O completion by calling the **vms_iowait** service, giving the virtual memory object as an argument.

Discarding Data

The pages specified by a data range can be released from the underlying virtual memory object by calling the **vm_release** service. The virtual memory manager deallocates any associated paging space slots. A subsequent reference to data in the range results in a page fault.

A virtual memory data provider can release a specified range of pages in a virtual memory object by calling the **vm_releasep** service. The virtual memory object need not be addressable for this call.

Protecting Data

The **vm_protectp** service can change the storage protect keys in a page range in one client storage virtual memory object. This only acts on the resident pages. The pages are referred to through the virtual memory object. They do not need to be addressable in the current address space. A client file system data provider uses this protection to detect stores to in-memory data, so that mapped files can be extended by storing into them beyond their current end of file.

Executable Data

If the data moved is to become executable, any data remaining in processor cache must be guaranteed to be moved from cache to memory. This is because the retrieval of the instruction does not need to use the data cache. The **vm_cflush** service performs this operation.

Installing Pager Backends

The kernel extension data providers must provide appropriate routines to be called by the virtual memory manager. These routines move a page-sized block of data into or out of a specified page. These services are also referred to as *pager backends*.

For a local device, the device strategy routine is required. A call to the **vm_mount** service is used to identify the device (through a **dev_t** value) to the virtual memory manager.

For a remote data provider, the routine required is a strategy routine, which is specified in the **vm_mount** service. These strategy routines must run as interrupt-level routines. They must not page fault, and they cannot sleep waiting for locks.

When access to a remote data provider or a local device is removed, the **vm_umount** service must be called to remove the device entry from the virtual memory manager's paging device table.

Referenced Routines

The virtual memory manager exports these routines exported to kernel extensions:

Services That Manipulate Virtual Memory Objects

vm_att	Selects and allocates a region in the current address space for the specified virtual memory object.
vms_create	Creates virtual memory object of the specified type and size limits.
vms_delete	Deletes a virtual memory object.
vm_det	Unmaps and deallocates the region at a specified address in the current address space.
vm_handle	Constructs a virtual memory handle for mapping a virtual memory object with a specified access level.
vms_iowait	Waits for the completion of all page-out operations in the virtual memory object.
vm_makep	Makes a page in client storage.

Services That Manipulate Virtual Memory Objects

vm_move	Moves data between the virtual memory object and buffer specified in the uio structure.
vm_protectp	Sets the page protection key for a page range.
vm_releasep	Releases page frames and paging space slots for pages in the specified range.
vm_uiomove	Moves data between the virtual memory object and buffer specified in the uio structure.
vm_writep	Initiates page-out for a page range in a virtual memory object.

Services That Support Address Space Operations

as_att	Selects, allocates, and maps a region in the specified address space for the specified virtual memory object.
as_det	Unmaps and deallocates a region in the specified address space that was mapped with the as_att kernel service.
as_geth	Obtains a handle to the virtual memory object for the specified address given in the specified address space. The virtual memory object is protected.
as_getsrval	Obtains a handle to the virtual memory object for the specified address given in the specified address space.
as_puth	Indicates that no more references will be made to a virtual memory object that was obtained using the as_geth kernel service.
as_seth	Maps a specified region in the specified address space for the specified virtual memory object.
getadsp	Obtains a pointer to the current process's address space structure for use with the as_att and as_det kernel services.
vm_cflush	Flushes cache lines for a specified address range.
vm_release	Releases page frames and paging space slots for the specified address range.
vm_write	Initiates page-out for an address range.

The following Memory-Pinning kernel services also support address space operations. They are the **pin**, **pinu**, **unpin**, and **unpinu** services.

Services That Support Cross-Memory Operations

Cross Memory Services are listed in "Memory Kernel Services" on page 53.

Services that Support the Installation of Pager Backends

vm_mount	Allocates an entry in the paging device table.
vm_umount	Removes a file system from the paging device table.

Services that Support 64-bit Processes

as_att64	Allocates and maps a specified region in the current user address space.
as_det64	Unmaps and deallocates a region in the current user address space that was mapped with the as_att64 kernel service.
as_geth64	Obtains a handle to the virtual memory object for the specified address.
as_puth64	Indicates that no more references will be made to a virtual memory object using the as_geth64 kernel service.
as_seth64	Maps a specified region for the specified virtual memory object.
as_getsrval64	Obtains a handle to the virtual memory object for the specified address.

IS64U	Determines if the current user address space is 64-bit or not.
remap_64	Register the input remapping of one or more addresses for the duration of a system call for 64-bit process.
as_remap64	Remaps an additional 64-bit address to a 32-bit address that can be used by the kernel.
as_unremap64	Returns the 64-bit original or unremapped address associated with a 32-bit remapped address.
rnmap_create64	Defines an effective address to real address translation region for either 64-bit or 32-bit effective addresses.
rnmap_remove64	Destroys an effective address to real address translation region.
xmattach64	Attaches to a user buffer for cross-memory operations.
copyin64	Copies data between user and kernel memory.
copyout64	Copies data between user and kernel memory.
copyinstr64	Copies data between user and kernel memory.
fubyte64	Retrieves a byte of data from user memory.
fuword64	Retrieves a word of data from user memory.
subyte64	Stores a byte of data in user memory.
suword64	Stores a word of data in user memory.

Message Queue Kernel Services

The Message Queue kernel services provide the same message queue functions to a kernel extension as the **msgctl**, **msgget**, **msgsnd**, and **msgrcv** subroutines make available to a program executing in user mode. Parameters have been added for moving returned information to an explicit parameter to free the return codes for error code usage. Instead of the error information available in the **errno** global variable (as in user mode), the Message Queue services use the service's return code. The error values are the same, except that a memory fault error (**EFAULT**) cannot occur because message buffer pointers in the kernel address space are assumed to be valid.

The Message Queue services can be called only from the process environment (see "Understanding Execution Environments" on page 7) because they prevent the caller from specifying kernel buffers. These services can be used as an Interprocess Communication mechanism to other kernel processes or user-mode processes. See "Kernel Extension and Device Driver Management Kernel Services" on page 46 for more information on the functions that these services provide.

There are four Message Queue services available from the kernel:

kmsgctl	Provides message-queue control operations.
kmsgget	Obtains a message-queue identifier.
kmsgrcv	Reads a message from a message queue.
kmsgsnd	Sends a message using a previously defined message queue.

Network Kernel Services

The Network kernel services are divided into:

- “Address Family Domain and Network Interface Device Driver Kernel Services”
- “Routing and Interface Address Kernel Services” on page 62
- “Loopback Kernel Services” on page 62
- “Protocol Kernel Services” on page 62
- “Communications Device Handler Interface Kernel Services” on page 63

Address Family Domain and Network Interface Device Driver Kernel Services

The Address Family Domain and Network Interface Device Driver services enable address family domains (Protocols) and network interface drivers to add and remove themselves from network switch tables.

The **if_attach** service and **if_detach** services add and remove network interfaces from the Network Interface List. Protocols search this list to determine an appropriate interface on which to transmit a packet.

Protocols use the **add_input_type** and **del_input_type** services to notify network interface drivers that the protocol is available to handle packets of a certain type. The Network Interface Driver uses the **find_input_type** service to distribute packets to a protocol.

The **add_netisr** and **del_netisr** services add and delete network software interrupt handlers. Address families add and delete themselves from the Address Family Domain switch table by using the **add_domain_af** and **del_domain_af** services. The Address Family Domain switch table is a list of all available protocols that can be used in the **socket** subroutine.

The Address Family Domain and Network Interface Device Driver services are:

add_domain_af	Adds an address family to the Address Family domain switch table.
add_input_type	Adds a new input type to the Network Input table.
add_netisr	Adds a network software interrupt service to the Network Interrupt table.
del_domain_af	Deletes an address family from the Address Family domain switch table.
del_input_type	Deletes an input type from the Network Input table.
del_netisr	Deletes a network software interrupt service routine from the Network Interrupt table.
find_input_type	Finds the given packet type in the Network Input Interface switch table and distributes the input packet according to the table entry for that type.
if_attach	Adds a network interface to the network interface list.
if_detach	Deletes a network interface from the network interface list.
ifunit	Returns a pointer to the ifnet structure of the requested interface.
schednetisr	Schedules or invokes a network software interrupt service routine.

Routing and Interface Address Kernel Services

The Routing and Interface Address services provide protocols with a means of establishing, accessing, and removing routes to remote hosts or gateways. Routes bind destinations to a particular network interface.

The interface address services accept a destination address or network and return an associated interface address. Protocols use these services to determine if an address is on a directly connected network.

The Routing and Interface Address services are:

ifa_ifwithaddr	Locates an interface based on a complete address.
ifa_ifwithdstaddr	Locates the point-to-point interface with a given destination address.
ifa_ifwithnet	Locates an interface on a specific network.
if_down	Marks an interface as down.
if_nostat	Zeroes statistical elements of the interface array in preparation for an attach operation.
rtalloc	Allocates a route.
rtfree	Frees the routing table entry
rtinit	Sets up a routing table entry, typically for a network interface.
rtredirect	Forces a routing table entry with the specified destination to go through the given gateway.
rtrequest	Carries out a request to change the routing table.

Loopback Kernel Services

The Loopback services enable networking code to be exercised without actually transmitting packets on a network. This is a useful tool for developing new protocols without introducing network variables. Loopback services can also be used to send packets to local addresses without using hardware loopback.

The Loopback services are:

loifp	Returns the address of the software loopback interface structure.
looutput	Sends data through a software loopback interface.

Protocol Kernel Services

Protocol kernel services provide a means of finding a particular address family as well as a raw protocol handler. The raw protocol handler basically passes raw packets up through sockets so that a protocol can be implemented in user space.

The Protocol kernel services are:

pfctlinput	Starts the ctlinput function for each configured protocol.
pffindproto	Returns the address of a protocol switch table entry.
raw_input	Builds a raw_header structure for a packet and sends both to the raw protocol handler.
raw_usrreq	Implements user requests for raw protocols.

Communications Device Handler Interface Kernel Services

The Communications Device Handler Interface services provide a standard interface between network interface drivers and communications device handlers (see “Communications Physical Device Handler Model Overview” on page 100). The **net_attach** and **net_detach** services open and close the device handler. Once the device handler has been opened, the **net_xmit** service can be used to transmit packets. Asynchronous start done notifications are recorded by the **net_start_done** service. The **net_error** service handles error conditions.

The Communications Device Handler Interface services are:

add_netopt	This macro adds a network option structure to the list of network options.
del_netopt	This macro deletes a network option structure from the list of network options.
net_attach	Opens a communications I/O device handler.
net_detach	Closes a communications I/O device handler.
net_error	Handles errors for communication network interface drivers.
net_sleep	Sleeps on the specified wait channel.
net_start	Starts network IDs on a communications I/O device handler.
net_start_done	Starts the done notification handler for communications I/O device handlers.
net_wakeup	Wakes up all sleepers waiting on the specified wait channel.
net_xmit	Transmits data using a communications I/O device handler.
net_xmit_trace	Traces transmit packets. This kernel service was added for those network interfaces that do not use the net_xmit kernel service to trace transmit packets.

Process and Exception Management Kernel Services

The process and exception management kernel services provided by the base kernel provide the capability to:

- “Introduction to Kernel Processes” on page 11
- “Understanding Exception Handling” on page 18
- Provide process serialization
- Generate and handle signals
- Support event waiting and notification

Creating Kernel Processes

Kernel extensions use the **creatp** and **initp** kernel services to create and initialize a kernel process (see “Introduction to Kernel Processes” on page 11). The **setpinit** kernel service allow a kernel process to change its parent process from the one that created it to the **init** process, so that the creating process does not receive the death-of-child process signal upon kernel process termination. “Using Kernel Processes” on page 11 supplies additional information concerning use of these services.

Creating Kernel Threads

Kernel extensions use the **thread_create** and **kthread_start** services to create and initialize kernel-only threads. “Understanding Kernel Threads” on page 9 provides more information about threads.

The `thread_setsched` service is used to control the scheduling parameters, priority and scheduling policy, of a thread.

Kernel Structures Encapsulation

The `getpid` kernel service is used by a kernel extension in either the process or interrupt environment to determine the current execution environment (see “Understanding Execution Environments” on page 7) and obtain the process ID of the current process if in the process environment. The `rusage_incr` service provides an access to the `rusage` structure.

The thread-specific `uthread` structure is also encapsulated. The `getuerror` and `setuerror` kernel services should be used to access the `ut_error` field. The `thread_self` kernel service should be used to get the current thread’s ID.

Registering Exception Handlers

The `setjmpx`, `clrjmpx`, and `longjmpx` kernel services allow a kernel extension to register an exception handler by:

- Saving the exception handler’s context with the `setjmpx` kernel service
- Removing its saved context with the `clrjmpx` kernel service if no exception occurred
- Starting the next registered exception handler with the `longjmpx` kernel service if it was unable to handle the exception

Refer to “Handling Exceptions While in a System Call” on page 29 for additional information concerning use of these services.

Signal Management

Signals can be posted either to a kernel process or to a kernel thread. The `pidsig` service posts a signal to a specified kernel process; the `kthread_kill` service posts a signal to a specified kernel thread. A thread uses the `sig_chk` service to poll for signals delivered to the kernel process or thread in the kernel mode.

“Kernel Process Signal and Exception Handling” on page 14 provides more information about signal management.

Events Management

The event notification services provide support for two types of interprocess communications:

Primitive	Allows only one process thread waiting on the event.
Shared	Allows multiple processes threads waiting on the event.

The `et_wait` and `et_post` kernel services support single waiter event notification by using mutually agreed upon event control bits for the kernel thread being posted. There are a limited number of control bits available for use by kernel extensions. If the `kernel_lock` is owned by the caller of the `et_wait` service, it is released and acquired again upon wakeup.

The following kernel services support a shared event notification mechanism that allows for multiple threads to be waiting on the shared event.

<code>e_assert_wait</code>	<code>e_wakeup</code>
<code>e_block_thread</code>	<code>e_wakeup_one</code>

e_clear_wait	e_wakeup_w_result
e_sleep_thread	e_wakeup_w_sig

These services support an unlimited number of shared events (by using caller-supplied event words). The following list indicates methods to wait for an event to occur:

- Calling **e_assert_wait** and **e_block_thread** successively; the first call puts the thread on the event queue, the second blocks the thread. Between the two calls, the thread can do any job, like releasing several locks. If only one lock, or no lock at all, needs to be released, one of the two other methods should be preferred.
- Calling **e_sleep_thread**; this service releases a simple or a complex lock, and blocks the thread. The lock can be automatically reacquired at wakeup.

The **e_clear_wait** service can be used by a thread or an interrupt handler to wake up a specified thread, or by a thread that called **e_assert_wait** to remove itself from the event queue without blocking when calling **e_block_thread**. The other wakeup services are event-based. The **e_wakeup** and **e_wakeup_w_result** services wake up every thread sleeping on an event queue; while the **e_wakeup_one** service wakes up only the most favored thread. The **e_wakeup_w_sig** service posts a signal to every thread sleeping on an event queue, waking up all the threads whose sleep is interruptible.

The **e_sleep** and **e_sleepl** kernel services are provided for code that was written for previous releases of the operating system. Threads which have called one of these services are woken up by the **e_wakeup**, **e_wakeup_one**, **e_wakeup_w_result**, **e_wakeup_w_sig**, or **e_clear_wait** kernel services. If the caller of the **e_sleep** service owns the **kernel lock**, it is released before waiting and is acquired again upon wakeup. The **e_sleepl** service provides the same function as the **e_sleep** service except that a caller-specified lock is released and acquired again instead of the **kernel_lock**.

List of Process, Thread, and Exception Management Kernel Services

The Process, Thread, and Exception Management kernel services are listed below.

clrjmpx	Removes a saved context by popping the most recently saved jump buffer from the list of saved contexts.
creatp	Creates a new kernel process.
e_assert_wait	Asserts that the calling kernel thread is going to sleep.
e_block_thread	Blocks the calling kernel thread.
e_clear_wait	Clears the wait condition for a kernel thread.
e_sleep , e_sleep_thread , or e_sleepl	Forces the calling kernel thread to wait for the occurrence of a shared event.
e_sleep_thread	Forces the calling kernel thread to wait the occurrence of a shared event.
e_wakeup , e_wakeup_one , or e_wakeup_w_result	Notifies kernel threads waiting on a shared event of the event's occurrence.
e_wakeup_w_sig	Posts a signal to sleeping kernel threads.
et_post	Notifies a kernel thread of the occurrence of one or more events.
et_wait	Forces the calling kernel thread to wait for the occurrence of an event.
getpid	Gets the process ID of the current process.

getppid	Gets the parent process ID of the specified process.
initp	Changes the state of a kernel process from idle to ready.
kthread_kill	Posts a signal to a specified kernel-only thread.
kthread_start	Starts a previously created kernel-only thread.
limit_sigs	Changes the signal mask for the calling kernel thread.
longjmpx	Allows exception handling by causing execution to resume at the most recently saved context.
NLuprintf	Submits a request to print an internationalized message to the controlling terminal of a process.
pgsignal	Sends a signal to a process group.
pidsig	Sends a signal to a process.
rusage_incr	Increments a field of the rusage structure.
setjmpx	Allows saving the current execution state or context.
setpinit	Sets the parent of the current kernel process to the init process.
sig_chk	Provides the calling kernel thread with the ability to poll for receipt of signals.
sigsetmask	Changes the signal mask for the calling kernel thread.
sleep	Forces the calling kernel thread to wait on a specified channel.
thread_create	Creates a new kernel-only thread in the calling process.
thread_self	Returns the caller's kernel thread ID.
thread_setsched	Sets kernel thread scheduling parameters.
thread_terminate	Terminates the calling kernel thread.
uprintf	Submits a request to print a message to the controlling terminal of a process.

RAS Kernel Services

The Reliability, Availability, and Serviceability (RAS) kernel services are used to record the occurrence of hardware or software failures and to capture data about these failures. The recorded information can be examined using the **errpt**, **trcrpt**, or **crash** commands.

The **panic** kernel service is called when a catastrophic failure occurs and the system can no longer operate. The **panic** service performs a system dump. The system dump captures data areas that are registered in the Master Dump Table. The kernel and kernel extensions use the **dmp_add** kernel service to add an entry to the Master Dump Table and the **dmp_del** kernel service to remove an entry.

The **errsave** and **errlast** kernel service is called to record an entry in the system error log when a hardware or software failure is detected.

The **trcgenk** and **trcgenkt** kernel services are used along with the **trchhook** subroutine to record selected system events in the event-tracing facility.

List of RAS Kernel Services

The RAS kernel services are:

dmp_add	Specifies data to be included in a system dump by adding an entry to the master dump table.
dmp_del	Deletes an entry from the master dump table.
dmp_pprint	Initializes the remote dump protocol.
errsave and errlast	Allows the kernel and kernel extensions to write to the error log.
panic	Crashes the system.

trcgenk	Records a trace event for a generic trace channel.
trcgenkt	Records a trace event, including a time stamp, for a generic trace channel.

Security Kernel Services

The Security kernel services provide methods for controlling the auditing system and for determining the access rights to objects for the invoking process.

The following services are Security kernel services:

suser	Determines the privilege state of a process.
audit_svcstart	Initiates an audit record for a kernel service.
audit_svcbcopy	Appends event information to the current audit event buffer.
audit_svcfinis	Writes an audit record for a kernel service.

Timer and Time-of-Day Kernel Services

The Timer and Time-of-Day kernel services provide kernel extensions with the ability to be notified when a period of time has passed. The **tstart** service supports a very fine granularity of time. The **timeout** service is built on the **tstart** service and is provided for compatibility with earlier versions of the operating system. The **w_start** service provides a timer with less granularity, but much cheaper path-length overhead when starting a timer.

Time-Of-Day Kernel Services

The Time-Of-Day kernel services are:

curtime	Reads the current time into a time structure.
kgettickd	Retrieves the current status of the systemwide time-of-day timer-adjustment values.
ksettimer	Sets the systemwide time-of-day timer.
ksettickd	Sets the current status of the systemwide timer-adjustment values.

Fine Granularity Timer Kernel Services

The Fine Granularity Timer kernel services are:

delay	Suspends the calling process for the specified number of timer ticks.
talloc	Allocates a timer request block before starting a timer request.
tfree	Deallocates a timer request block.
tstart	Submits a timer request.
tstop	Cancels a pending timer request.

You can find additional information about using the Fine Granularity Timer services in “Using Fine Granularity Timer Services and Structures” on page 68.

Timer Kernel Services for Compatibility

The following Timer kernel services are provided for compatibility:

timeout	Schedules a function to be called after a specified interval.
timeoutcf	Allocates or deallocates callout table entries for use with the timeout kernel service.
untimeout	Cancels a pending timer request.

Watchdog Timer Kernel Services

The Watchdog timer kernel services are:

w_clear	Removes a watchdog timer from the list of watchdog timers known to the kernel.
w_init	Registers a watchdog timer with the kernel.
w_start	Starts a watchdog timer.
w_stop	Stops a watchdog timer.

Using Fine Granularity Timer Services and Structures

The **tstart**, **tfree**, **talloc**, and **tstop** services provide fine-resolution timing functions. These timer services should be used when the following conditions are required:

- Timing requests for less than one second
- Critical timing
- Absolute timing

The Watchdog timer services can be used for noncritical times having a one-second resolution. The **timeout** service can be used for noncritical times having a clock-tick resolution.

Timer Services Data Structures

The **trb** (timer request) structure is found in the `/sys/timer.h` file. The **itimerstruc_t** structure contains the second/nanosecond structure for time operations and is found in the `sys/time.h` file.

The **itimerstruc_t** **it** value substructure should be used to store time information for both absolute and incremental timers. The **T_ABSOLUTE** absolute request flag is defined in the `sys/timer.h` file. It should be ORed into the `t->flag` field if an absolute timer request is desired.

The **T_LOWRES** flag causes the system to round the `t->timeout` value to the next timer timeout. It should be ORed into the `t->flags` field. The timeout is always rounded to a larger value. Since the system maintains 10ms interval timer, **T_LOWRES** will never cause more than 10ms to be added to a timeout. The advantage of using **T_LOWRES** is that it prevents an extra interrupt from being generated.

The `t->timeout` and `t->flags` fields must be set or reset before each call to the **tstart** kernel service.

Coding the Timer Function

The `t->func` timer function should be declared as follows:

```
void func (t)
struct trb *t;
```

The argument to the `func` completion handler routine is the address of the `trb` structure, not the contents of the `t_union` field.

The `t->func` timer function is called on an interrupt level. Therefore, code for this routine must follow conventions for interrupt handlers.

Using Multiprocessor-Safe Timer Services

On a multiprocessor system, timer request blocks and watchdog timer structures could be accessed simultaneously by several processors. The kernel services shown below potentially alter critical information in these blocks and structures, and therefore check whether it is safe to perform the requested service before proceeding:

tstop	Cancels a pending timer request.
w_clear	Removes a watchdog timer from the list of watchdog timers known to the kernel.
w_init	Registers a watchdog timer with the kernel.

If the requested service cannot be performed, the kernel service returns an error value.

In order to be multiprocessor safe, the caller must check the value returned by these kernel services. If the service was not successful, the caller must take an appropriate action, for example, retrying in a loop. If the caller holds a device driver lock, it should release and then reacquire the lock within this loop in order to avoid deadlock.

Drivers which were written for uniprocessor systems do not check the return values of these kernel services and are not multiprocessor-safe. Such drivers can still run as funnelled device drivers.

Virtual File System (VFS) Kernel Services

The Virtual File System (VFS) kernel services are provided as fundamental building blocks for use when writing a virtual file system. These services present a standard interface for such functions as configuring file systems, creating and freeing v-nodes, and looking up path names.

Most functions involved in the writing of a file system are specific to that file system type. But a limited number of functions must be performed in a consistent manner across the various file system types to enable the logical file system to operate independently of the file system type.

The VFS kernel services are:

common_relock	Implements a generic interface to the byte range locking functions.
fidtopv	Maps a file system structure to a file ID.

gfsadd	Adds a file system type to the gfs table.
gfsdel	Removes a file system type from the gfs table.
vfs_hold	Holds a vfs structure and increments the structure's use count.
vfs_unhold	Releases a vfs structure and decrements the structure's use count.
vfsrele	Releases all resources associated with a virtual file system.
vfs_search	Searches the vfs list.
vn_free	Frees a v -node previously allocated by the vn_get kernel service.
vn_get	Allocates a virtual node and associates it with the designated virtual file system.
lookupvp	Retrieves the v -node that corresponds to the named path.

Chapter 5. Asynchronous I/O Subsystem

The following topics pertain to Asynchronous I/O:

- “Asynchronous I/O Overview”
- “Prerequisites” on page 73
- “Functions of Asynchronous I/O” on page 73
- “Asynchronous I/O Subroutines” on page 75
- “Subroutines Affected by Asynchronous I/O” on page 76
- “Changing Attributes for Asynchronous I/O” on page 76
- “64-bit Enhancements” on page 77

Asynchronous I/O Overview

Synchronous I/O occurs while you wait. Applications processing cannot continue until the I/O operation is complete.

In contrast, asynchronous I/O operations run in the background and do not block user applications. This improves performance, because I/O operations and applications processing can run simultaneously.

Using asynchronous I/O will usually improve your I/O throughput, especially when you are storing data in raw logical volumes (as opposed to Journaled file systems). The actual performance, however, depends on how many server processes are running that will handle the I/O requests.

Many applications, such as databases and file servers, take advantage of the ability to overlap processing and I/O. These asynchronous I/O operations use various kinds of devices and files. Additionally, multiple asynchronous I/O operations may run at the same time on one or more devices or files.

Each asynchronous I/O request has a corresponding control block in the application's address space. When an asynchronous I/O request is made, a handle is established in the control block. This handle is used to retrieve the status and the return values of the request.

Applications use the **aio_read** and **aio_write** subroutines to perform the I/O. Control returns to the application from the subroutine, as soon as the request has been queued. The application can then continue processing while the disk operation is being performed.

A kernel process (KPROC), called a server, is in charge of each request from the time it is taken off the queue until it completes. The number of servers limits the number of disk I/O operations that can be in progress in the system simultaneously.

The default values are `minservers=1` and `maxservers=10`. In systems that seldom run applications that use asynchronous I/O, this is usually adequate. For environments with many disk drives and key applications that use asynchronous I/O, the default is far too low. The result of a deficiency of servers is that disk I/O seems much slower than it should be. Not only do requests spend inordinate

lengths of time in the queue, but the low ratio of servers to disk drives means that the seek-optimization algorithms have too few requests to work with for each drive.

How do I know if I need to use AIO?

Using the `vmstat` command with an interval and count value, you can determine if the CPU is idle waiting for disk I/O. The `wa` column details the percentage of time the CPU was idle with pending local disk I/O.

If there is at least one outstanding I/O to a local disk when the wait process is running, the time is classified as waiting for I/O. Unless asynchronous I/O is being used by the process, an I/O request to disk causes the calling process to block (or sleep) until the request has been completed. Once a process's I/O request completes, it is placed on the run queue.

A `wa` value consistently over 25 percent may indicate that the disk subsystem is not balanced properly, or it may be the result of a disk-intensive workload.

Note: AIO will not relieve an overly busy disk drive. Using the `iostat` command with an interval and count value, you can determine if any disks are overly busy. Monitor the `%tm_act` column for each disk drive on the system. On some systems, a `%tm_act` of 35.0 or higher for one disk can cause noticeably slower performance. The relief for this case could be to move data from more busy to less busy disks, but simply having AIO will not relieve an overly busy disk problem.

Important for SMP

For SMP systems, the `us`, `sy`, `id` and `wa` columns are only averages over all processors. But keep in mind that the I/O wait statistic per processor is not really a processor-specific statistic; it is a global statistic. An I/O wait is distinguished from idle time only by the state of a pending I/O. If there is any pending disk I/O, and the processor is not busy, then it is an I/O wait time. Disk I/O is not tracked by processors, so when there is any I/O wait, all processors get charged (assuming they are all equally idle).

How many AIO Servers am I currently using?

The following command will tell you how many AIO Servers are currently running (you must run this command as the "root" user):

```
pstat -a | grep aios | wc -l
```

If the disk drives that are being accessed asynchronously are using the AIX Journaled File System (JFS), all I/O will be routed through the `aios` KPROCs.

If the disk drives that are being accessed asynchronously are using a form of RAW logical volume management, then the disk I/O is not routed through the `aios` KPROCs. In that case the number of servers running is not relevant.

However, if you want to confirm that an application that uses RAW logic volumes is taking advantage of AIO, and you are at AIX 4.3.2 or AIX 4.3.x with APAR IX79690 installed, you can disable the "Fastpath" option via SMIT. When this option has been disabled, even RAW I/O will be forced through the `aios` KPROCs. At that point, the `pstat` command listed in preceding discussion will work. You would not want to run the system with this option disabled for any length of time. This is simply a suggestion to confirm that the application is working with AIO and RAW logical volumes.

At AIX levels before AIX 4.3, the "Fastpath" is enabled by default and cannot be disabled.

How many AIO servers do I need?

Here are some suggested rules of thumb for determining what value to set MAXIMUM number of servers to:

1. The first rule of thumb suggests that you limit the MAXIMUM number of servers to a number equal to ten times the number of disks that are to be used concurrently, but not more than 80. The MINIMUM number of servers should be set to half of this maximum number.
2. Another rule of thumb is to set the MAXIMUM number of servers to 80 and leave the MINIMUM number of servers set to the default of 1 and reboot. Monitor the number of additional servers started throughout the course of normal workload. After a 24-hour period of normal activity, set the MAXIMUM number of servers to:
(The number of currently running aios + 10),
and set the MINIMUM number of servers to:
(The number of currently running aios - 10).
In some environments you may see more than 80 aios KPROCs running. If so, consider this rule of thumb:
3. A third suggestion is to take statistics using `vmstat -s` before any high I/O activity begins, and again at the end. Check the field `iodone`. From this you can determine how many physical I/Os are being handled in a given wall clock period. Then increase the MAXIMUM number of servers and see if you can get more `iodones` in the same time period.

Prerequisites

To make use of asynchronous I/O the following fileset must be installed:

```
bos.rte.aio
```

To determine if this fileset is installed, use:

```
lspp -l bos.rte.aio
```

You must also make the `aio0` device "Available" via SMIT.

```
smit chgaio
```

```
STATE to be configured at system restart available
```

Functions of Asynchronous I/O

Functions provided by the asynchronous I/O facilities are:

- "Large File-Enabled Asynchronous I/O (AIX Version 4.2.1 or later)"
- "Nonblocking I/O" on page 74
- "Notification of I/O Completion" on page 74
- "Cancellation of I/O Requests" on page 75

Large File-Enabled Asynchronous I/O (AIX Version 4.2.1 or later)

The fundamental data structure associated with all asynchronous I/O operations is `struct aiocb`. Within this structure is the `aio_offset` field which is used to specify the offset for an I/O operation.

The default asynchronous I/O interfaces are limited to an offset of 2G minus 1 due to the signed 32-bit definition of `aio_offset`. To overcome this limitation, a new `aio` control block with a signed 64-bit offset field and a new set of asynchronous I/O interfaces have been defined beginning with AIX Version 4.2.1.

The large offset-enabled asynchronous I/O interfaces are available under the `_LARGE_FILES` compilation environment and under the `_LARGE_FILE_API` programming environment. For further information, see *Writing Programs That Access Large Files in AIX Version 4.3 General Programming Concepts: Writing and Debugging Programs*.

Under the `_LARGE_FILES` compilation environment in AIX Version 4.2.1 or later, asynchronous I/O applications written to the default interfaces see the following redefinitions:

Item	Redefined To Be	Header File
<code>struct aiocb</code>	<code>struct aiocb64</code>	<code>sys/aio.h</code>
<code>aio_read()</code>	<code>aio_read64()</code>	<code>sys/aio.h</code>
<code>aio_write()</code>	<code>aio_write64()</code>	<code>sys/aio.h</code>
<code>aio_cancel()</code>	<code>aio_cancel64()</code>	<code>sys/aio.h</code>
<code>aio_suspend()</code>	<code>aio_suspend64()</code>	<code>sys/aio.h</code>
<code>aio_listio()</code>	<code>aio_listio()</code>	<code>sys/aio.h</code>
<code>aio_return()</code>	<code>aio_return64()</code>	<code>sys/aio.h</code>
<code>aio_error()</code>	<code>aio_error64()</code>	<code>sys/aio.h</code>

For information on using the `_LARGE_FILES` environment, see *Porting Applications to the Large File Environment in AIX Version 4.3 General Programming Concepts: Writing and Debugging Programs*

In the `_LARGE_FILE_API` environment, the 64-bit API interfaces are visible. This environment requires recoding of applications to the new 64-bit API name. For further information on using the `_LARGE_FILE_API` environment, see *Using the 64-Bit File System Subroutines in AIX Version 4.3 General Programming Concepts: Writing and Debugging Programs*

Nonblocking I/O

After issuing an I/O request, the user application can proceed without being blocked while the I/O operation is in progress. The I/O operation occurs while the application is running. Specifically, when the application issues an I/O request, the request is queued. The application can then resume running before the I/O operation is initiated.

To manage asynchronous I/O, each asynchronous I/O request has a corresponding control block in the application's address space. This control block contains the control and status information for the request. It can be used again when the I/O operation is completed.

Notification of I/O Completion

After issuing an asynchronous I/O request, the user application can determine when and how the I/O operation is completed. This information is provided in three ways:

- The application can poll the status of the I/O operation (see “Polling the Status of the I/O Operation”).
- The system can asynchronously notify the application when the I/O operation is done (see “Asynchronously Notifying the Application When the I/O Operation Completes”).
- The application can block until the I/O operation is complete (see “Blocking the Application until the I/O Operation Is Complete”).

Polling the Status of the I/O Operation

The application can periodically poll the status of the I/O operation. The status of each I/O operation is provided in the application’s address space in the control block associated with each request. Portable applications can retrieve the status by using the `aio_error` subroutine.

Asynchronously Notifying the Application When the I/O Operation Completes

Asynchronously notifying the I/O completion is done by signals. Specifically, an application may request that a `SIGIO` signal be delivered when the I/O operation is complete. To do this, the application sets a flag in the control block at the time it issues the I/O request. If several requests have been issued, the application can poll the status of the requests to determine which have actually completed.

Blocking the Application until the I/O Operation Is Complete

The third way to determine whether an I/O operation is complete is to let the calling process become blocked and wait until at least one of the I/O requests it is waiting for is complete. This is similar to synchronous style I/O. It is useful for applications that, after performing some processing, need to wait for I/O completion before proceeding.

Cancellation of I/O Requests

I/O requests can be canceled if they are cancelable. Cancellation is not guaranteed and may succeed or not depending upon the state of the individual request. If a request is in the queue and the I/O operations have not yet started, the request is cancellable. Typically, a request is no longer cancelable when the actual I/O operation has begun.

Asynchronous I/O Subroutines

Note: The 64-bit APIs are available beginning with AIX Version 4.2.1.

The following subroutines are provided for performing asynchronous I/O:

Subroutine	Purpose
<code>aio_cancel</code> or <code>aio_cancel64</code>	Cancels one or more asynchronous I/O requests.
<code>aio_error</code> or <code>aio_error64</code>	Retrieves the error status of an I/O request.
<code>lio_listio</code> or <code>lio_listio64</code>	Initiates multiple asynchronous read and write operations.
<code>aio_read</code> or <code>aio_read64</code>	Reads asynchronously from a file.
<code>aio_return</code> or <code>aio_return64</code>	Retrieves the return value of an I/O request.
<code>aio_suspend</code> or <code>aio_suspend64</code>	Blocks until an asynchronous I/O is completed.
<code>aio_write</code> or <code>aio_write64</code>	Writes asynchronously to a file.

Note: These subroutines may change to conform with the IEEE POSIX 1003.4 interface specification.

Order and Priority of Asynchronous I/O Calls

An application may issue several asynchronous I/O requests on the same file or device. However, since the I/O operations are performed asynchronously, the order in which they are handled may not be the order in which the I/O calls were made. The application must enforce ordering of its own I/O requests if ordering is required.

Priority among the I/O requests is not currently implemented. The `aio_reqprio` field in the control block is currently ignored.

For files that support `seek` operations, seeking is allowed as part of the asynchronous read or write operations. The `whence` and `offset` fields are provided in the control block of the request to set the `seek` parameters. The seek pointer is updated when the asynchronous read or write call returns.

Subroutines Affected by Asynchronous I/O

The following existing subroutines are affected by asynchronous I/O:

- The `close` subroutine
- The `exit` subroutine
- The `exec` subroutine
- The `fork` subroutine

If the application closes a file, or calls the `_exit` or `exec` subroutines while it has some outstanding I/O requests, the requests are canceled. If they cannot be canceled, the application is blocked until the requests have completed. When a process calls the `fork` subroutine, its asynchronous I/O is not inherited by the child process.

One fundamental limitation in asynchronous I/O is page hiding. When an unbuffered (raw) asynchronous I/O is issued, the page that contains the user buffer is hidden during the actual I/O operation. This ensures cache consistency. However, the application may access the memory locations that fall within the same page as the user buffer. This may cause the application to block as a result of a page fault. To alleviate this, allocate page aligned buffers and do not touch the buffers until the I/O request using it has completed.

Changing Attributes for Asynchronous I/O

You can change attributes relating to asynchronous I/O using the `chdev` command or SMIT. Likewise, you can use SMIT to configure and remove (unconfigure) asynchronous I/O. (Alternatively, you can use the `mkdev` and `rmdev` commands to configure and remove asynchronous I/O). To start SMIT at the main menu for asynchronous I/O, enter `smit aio`.

MINIMUM number of servers

indicates the minimum number of kernel processes dedicated to asynchronous I/O processing. Since each kernel process uses memory, this number should not be large when the amount of asynchronous I/O expected is small.

MAXIMUM number of servers

indicates the maximum number of kernel processes dedicated to

asynchronous I/O processing. There can never be more than this many asynchronous I/O requests in progress at one time, so this number limits the possible I/O concurrency.

Maximum number of REQUESTS

indicates the maximum number of asynchronous I/O requests that can be outstanding at one time. This includes requests that are in progress as well as those that are waiting to be started. The maximum number of asynchronous I/O requests cannot be less than the value of AIO_MAX, as defined in the `/usr/include/sys/limits.h` file, but it can be greater. It would be appropriate for a system with a high volume of asynchronous I/O to have a maximum number of asynchronous I/O requests larger than AIO_MAX.

Server PRIORITY

indicates the priority level of kernel processes dedicated to asynchronous I/O. The lower the priority number is, the more favored the process is in scheduling. Concurrency is enhanced by making this number slightly less than the value of PUSER, the priority of a normal user process. It cannot be made lower than the values of PRI_SCHED.

Since the default priority is (40+nice), these daemons will be slightly favored with this value of (39+nice). If you want to favor them more, `m` changes slowly. A very low priority can interfere with the system process that require low priority.

Attention: Raising the server PRIORITY (decreasing this numeric value) is not recommended. The system can hang or crash.

PUSER and PRI_SCHED are defined in the `/usr/include/sys/pri.h` file.

STATE to be configured at system restart

indicates the state to which asynchronous I/O is to be configured during system initialization. The possible values are 1.) `defined`, which indicates that the asynchronous I/O will be left in the defined state and not available for use, and 2.) `available`, indicating that asynchronous I/O will be configured and available for use.

STATE of FastPath

You will only see this option if you are at AIX 4.3.2 or any level of AIX 4.3.x with APAR IX79690 installed. Disabling this option forces ALL I/O activity through the `aio` KPROCs, even I/O activity involving RAW logical volumes. At AIX levels before AIX 4.3 the "Fastpath" is enabled by default and cannot be disabled.

64-bit Enhancements

Asynchronous I/O (AIO) has been enhanced to support 64-bit enabled applications. On 64-bit platforms, both 32-bit and 64-bit AIO can occur simultaneously.

The struct `aio_cb`, the fundamental data structure associated with all asynchronous I/O operation, has changed. The element of this struct, `aio_return`, is now defined as `ssize_t`. Previously, it was defined as an `int`. AIO supports large files by default. An application compiled in 64-bit mode can do AIO to a large file without any additional `#defines` or special opening of those files.

Chapter 6. Device Configuration Subsystem

Devices are usually pieces of equipment that attach to a computer. Devices include printers, adapters, and disk drives. Additionally, devices are special files that can handle device-related tasks.

System users cannot operate devices until device configuration occurs. To configure devices, the Device Configuration Subsystem is available.

- “Scope of Device Configuration Support”
- “Device Configuration Subsystem Overview”
- “General Structure of the Device Configuration Subsystem” on page 80

Scope of Device Configuration Support

The term *device* has a wider range of meaning in this operating system than in traditional operating systems. Traditionally, *devices* refers to hardware components such as disk drives, tape drives, printers, and keyboards. Pseudo-devices, such as the console, **error** special file, and **null** special file, are also included in this category. However, in this operating system, all of these devices are referred to as *kernel devices*, which have device drivers and are known to the system by major and minor numbers.

Also, in this operating system, hardware components such as buses, adapters, and enclosures (including racks, drawers, and expansion boxes) are considered devices.

Device Configuration Subsystem Overview

Devices are organized hierarchically within the system. This organization requires lower-level device dependence on upper-level devices in child-parent relationships. The system device (`sys0`) is the highest-level device in the system node, which consists of all physical devices in the system.

Each device is classified into functional classes, functional subclasses and device types (for example, printer *class*, parallel *subclass*, 4201 Proprinter *type*). These classifications are maintained in the device configuration databases with all other device information.

A *DDS* (device dependent structure) is a structure provided to communicate a device's characteristics from a *Configure* method to a device driver. The device's DDS is built each time the device is configured (*Configure method*).

The Device Configuration Subsystem consists of:

High-level Commands	Maintain (add, delete, view, change) configured devices within the system. These commands manage all of the configuration functions and are performed by invoking the appropriate device methods for the device being configured. These commands call device methods and low-level commands. The system uses the high-level Configuration Manager (cfgmgr) command used to invoke automatic device configurations through system boot phases and the user can invoke the command during system run time. <i>Configuration rules</i> govern the cfgmgr command.
Device Methods	Define and configure, start and stop devices. The device methods are used to identify or change the device <i>states</i> (operational modes). Device methods can call low-level commands.
Low-level Commands	Perform routines and functions common to all devices (e.g., to update device attribute information).
Database	Maintains data through the <i>ODM</i> (Object Data Manager) by object classes. Predefined Device Objects contain configuration data for all devices that can possibly be used by the system. Customized Device Objects contain data for <i>device instances</i> that are actually in use by the system.

General Structure of the Device Configuration Subsystem

The Device Configuration Subsystem can be viewed from three different levels:

- High-level perspective
- Device method level
- Low-level perspective

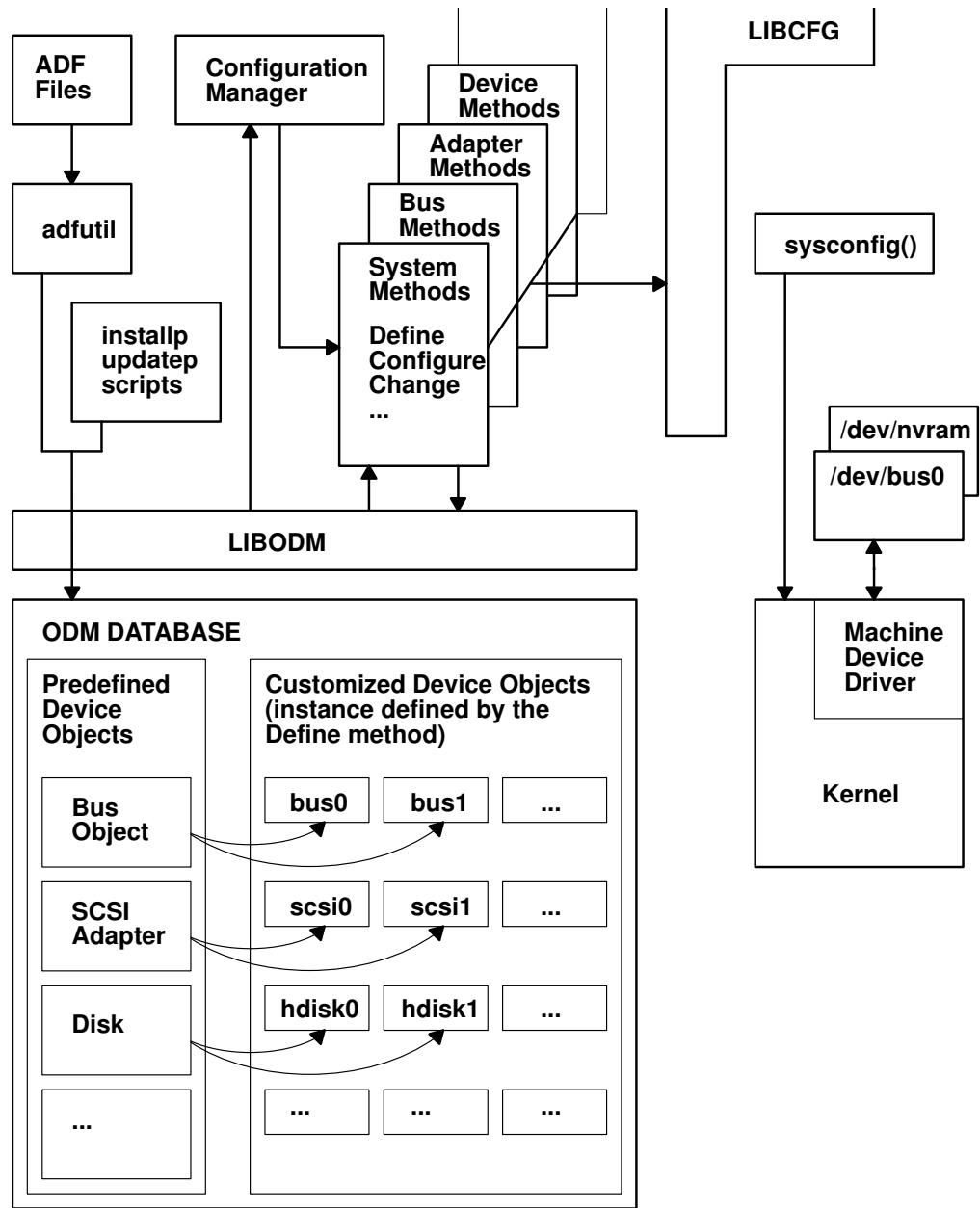
Data that is used by the three levels is maintained in the *configuration database* (see “Device Configuration Database Overview” on page 84). The database is managed as object classes by the Object Data Manager (ODM). All information relevant to support the device configuration process is stored in the configuration database.

The system cannot use any device unless it is configured.

The database has two components: the Predefined database and the Customized database. The *Predefined database* contains configuration data for all devices that could possibly be supported by the system. The *Customized database* contains configuration data for the devices actually defined and configured in that particular system.

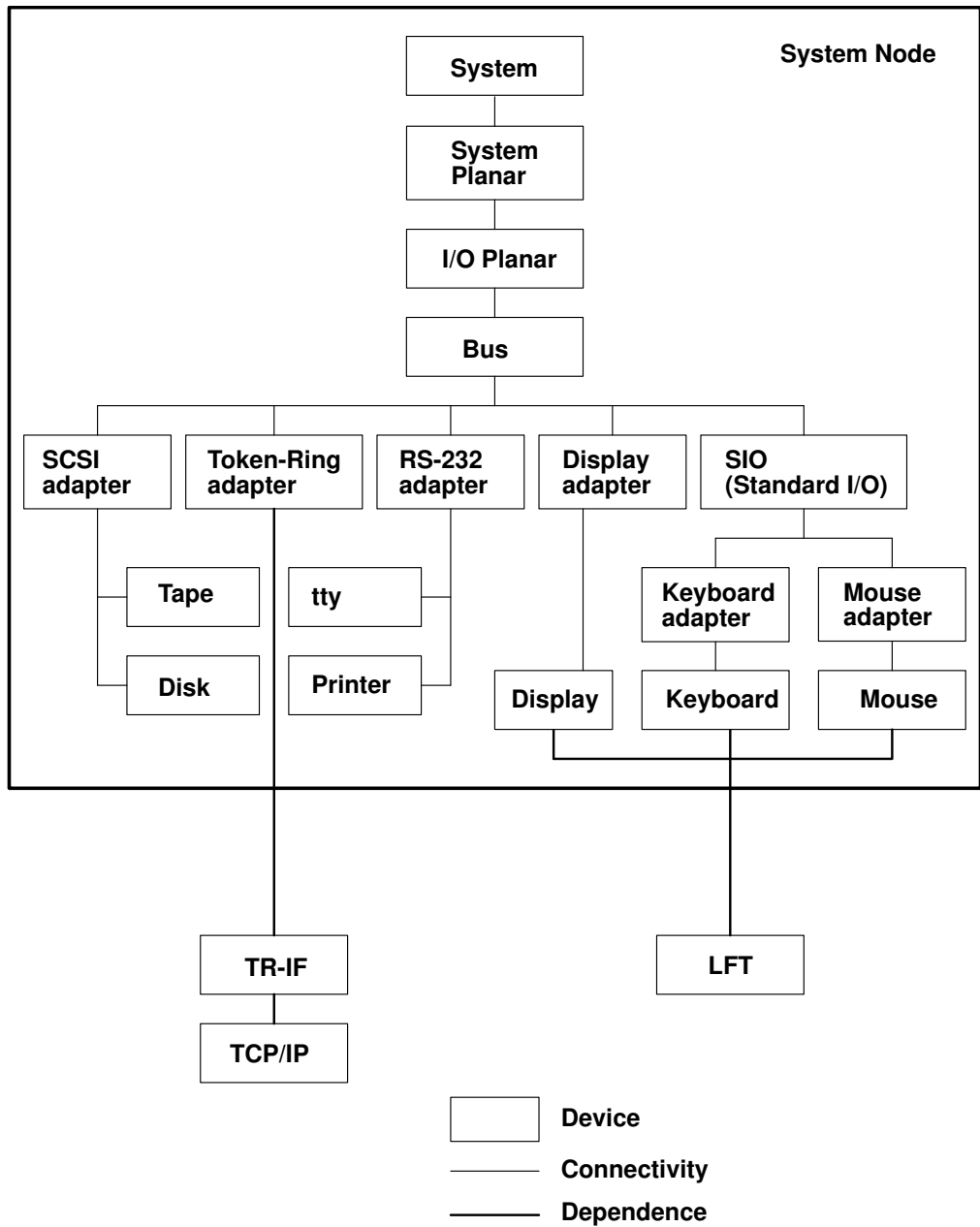
The *configuration manager* (**cfgmgr** command) performs the configuration of a system’s devices automatically when the system is booted (see “Device Configuration Manager Overview” on page 85). This high-level program can also be invoked through the system keyboard to perform automatic device configuration. The configuration manager command configures devices as specified by *Configuration rules*.

These components are illustrated in the Components Involved in Device Configuration Support diagram on 81.



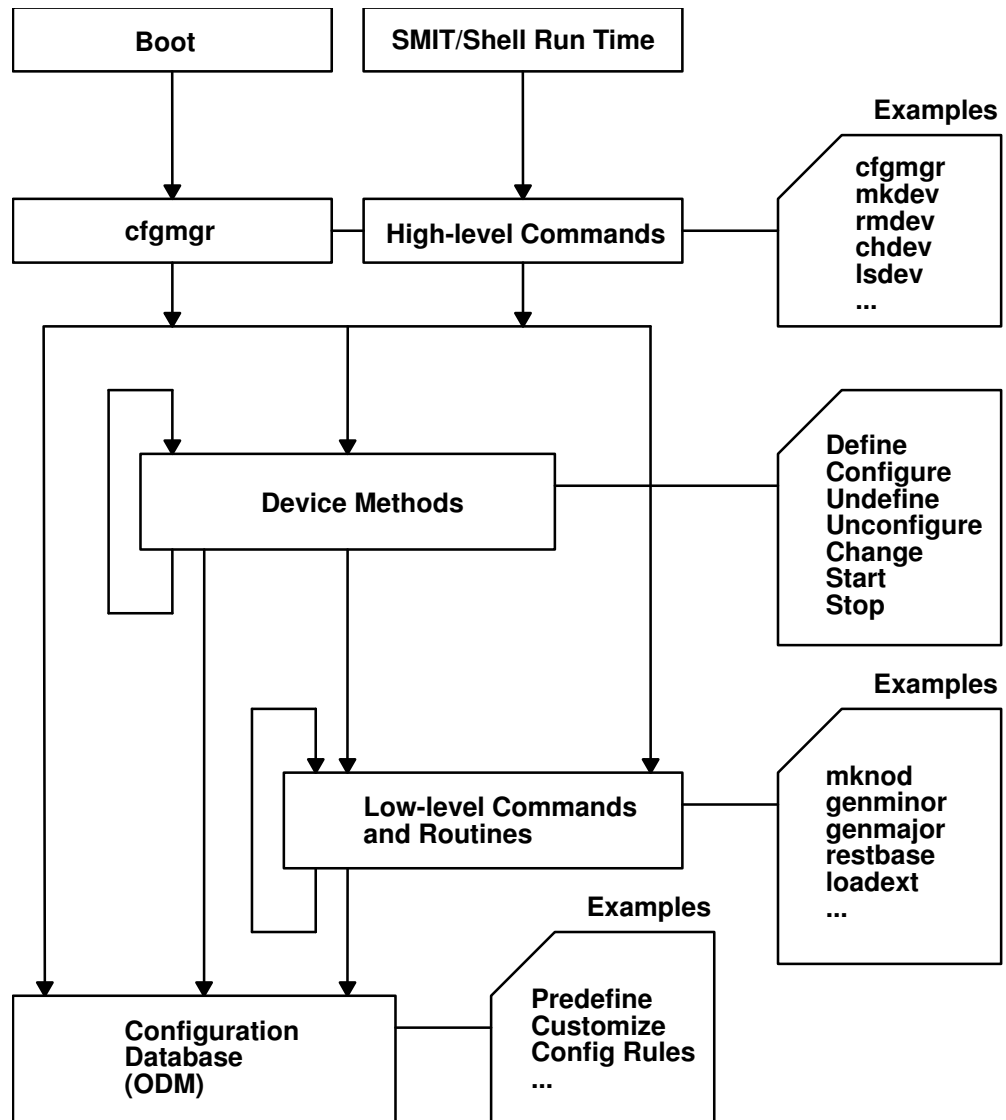
Components Involved in Device Configuration Support

The Devices Graph: Examples of Connectivity and Dependence diagram on 82 provides more information about the connections and dependencies between these components.



Devices Graph: Examples of Connectivity and Dependence

The Overview of System Management of Devices diagram on 83 illustrates the general structure of the Device Configuration Subsystem.



Overview of System Management of Devices

High-Level Perspective

From a high-level, user-oriented perspective, device configuration comprises the following basic tasks:

- Adding a device to the system
- Deleting a device from the system
- Changing the attributes of a device
- Showing information about a device

From a high-level, system-oriented perspective, device configuration provides the basic task of automatic device configuration: running the configuration manager program.

A set of high-level commands accomplish all of these tasks during run time: **chdev**, **mkdev**, **lsattr**, **lsconn**, **lsdev**, **lsparent**, **rmdev**, and **cfgmgr**. High-level commands can invoke device methods and low-level commands.

Device Method Level

Beneath the high-level commands (including the **cfgmgr** Configuration Manager program) is a set of functions called *device methods*. These methods perform well-defined configuration steps, including these five functions:

- Defining a device in the configuration database
- Configuring a device to make it available
- Changing a device to make a change in its characteristics
- Unconfiguring a device to make it unavailable
- Undefined a device from the configuration database

Device methods also provide two optional functions for devices that need them:

- Starting a device to take it from the Stopped state to the Available state
- Stopping a device to take it to the Stopped state

The Device States diagram (see “Understanding Device States” on page 90) illustrates all possible device states and how the various methods affect device state changes.

The high-level device commands (including **cfgmgr**) can use the device methods. These methods insulate high-level configuration programs from kernel-specific, hardware-specific, and device-specific configuration steps. Device methods can invoke low-level commands.

Low-Level Perspective

Beneath the device methods is a set of low-level device configuration commands and library routines that can be directly called by device methods as well as by high-level configuration programs.

Device Configuration Database Overview

The Configuration database is an object-oriented database. The Object Data Manager (ODM) provides facilities for accessing and manipulating it through object classes.

There are actually two databases used in the configuration process:

Predefined database	Contains information about all possible types of devices that can be defined for the system.
Customized database	Describes all devices currently defined for use in the system. Items are referred to as <i>device instances</i> .

“ODM Device Configuration Object Classes” in *AIX Technical Reference: Kernel and Subsystems Volume 1* provides access to the object classes that make up the Predefined and Customized databases.

Devices must be defined in the database for the system to make use of them. For a device to be in the Defined state, the Configuration database must contain a complete description of it. This information includes items such as the device driver name, the device major and minor numbers, the device method names, the device attributes, connection information, and location information.

Basic Device Configuration Procedures Overview

At system boot time, the Configuration Manager (**cfgmgr** high-level command) is automatically invoked to configure all devices detected as well as any device whose device information is stored in the Configuration database. At run time, you can configure a specific device by directly invoking (or indirectly invoking through a usability interface layer) high-level device commands. The "Overview of System Management of Devices" diagram on 83 illustrates this interface.

High-level device commands invoke methods and allow the user to add, delete, show, and change devices and their associated attributes.

When a specific device is defined through its Define method, the information from the Predefined database for that type of device is used to create the information describing the specific device instance. This specific device instance information is then stored in the Customized database.

The process of configuring a device is often highly device-specific. The Configure method for a kernel device must:

- Load the device's driver into the kernel.
- Pass the device dependent structure (DDS) describing the device instance to the driver (see "Device Dependent Structure (DDS) Overview" on page 95).
- Create a special file for the device in the **/dev** directory.

Of course, many devices do not have device drivers. For this type of device the configured state is not as meaningful. However, it still has a Configure method that simply marks the device as configured or performs more complex operations to determine if there are any devices attached to it.

The configuration process requires that a device be defined or configured before a device attached to it can be defined or configured. At system boot time, the Configuration Manager first configures the system device as shown in the "Devices Graph: Examples of Connectivity and Dependence" diagram on 82. The remaining devices are configured by traversing down the parent-child connections layer by layer. The Configuration Manager then configures any pseudo-devices that need to be configured.

Device Configuration Manager Overview

The Configuration Manager is a rule-driven program that automatically configures devices in the system during system boot and run time. When the Configuration Manager is invoked, it reads rules from the Configuration Rules object class and performs the indicated actions.

Devices in the system are organized in clusters of tree structures known as *nodes*. Each tree is a logical subsystem by itself. For example, the system node consists of all the physical devices in the system. The top of the node is the system device. Below the bus and connected to it are the adapters. The bottom of the hierarchy contains devices to which no other devices are connected. Most pseudo-devices, including low-function terminal (HFT LFT) and pseudo-terminal (pty) devices, are organized as separate tree structures or nodes.

Devices Graph

The Devices Graph: Examples of Connectivity and Dependence diagram on 82 provides an example of the connections and dependencies of devices in the system. "Understanding Device Dependencies and Child Devices" on page 93 provides more information.

Configuration Rules

Each rule in the Configuration Rules (Config_Rules) object class specifies a program name that the Configuration Manager must execute. These programs are typically the configuration programs for the devices at the top of the nodes. When these programs are invoked, the names of the next lower-level devices that need to be configured are returned.

If the **-m** (mask) flag is not used, the **cfgmgr** command executes all of the rules for the specified phase. When a mask is specified, the **cfgmgr** command applies the mask to each rule for the phase. If the mask specified with the **-m** flag matches the `boot_mask` field from the configuration rules, the rule is executed. Otherwise, the **cfgmgr** command does not execute the rule. In this way, phase 1 of the boot process can be tailored for a particular type of boot (for example, **DISK_BOOT**).

The Configuration Manager configures the next lower-level devices by invoking the configuration methods for those devices. In turn, those configuration methods return a list of to-be-configured device names. The process is repeated until no more device names are returned. As a result, all devices in the same node are configured in transverse order. There are three different types of rules:

- Phase 1
- Phase 2
- Service

The system boot process is divided into two phases. In each phase, the Configuration Manager is invoked. During phase 1, the Configuration Manager is called with a **-f** flag, which specifies that *phase = 1* rules are to be executed. This results in the configuration of base devices into the system, so that the root file system can be used. During phase 2, the Configuration Manager is called with a **-s** flag, which specifies that *phase = 2* rules are to be executed. This results in the configuration of the rest of the devices into the system.

"Understanding System Boot Processing" in *AIX Version 4.3 System Management Guide: Operating System and Devices* contains diagrams that illustrate the separate step of system boot processing.

The Configuration Manager invokes the programs in the order specified by the sequence value in the rule. In general, the lower the sequence number within a given phase, the higher the priority. Thus, a rule with a 2 sequence number is executed before a rule with a sequence number of 5. An exception is made for 0 sequence numbers, which indicate a don't-care condition. Any rule with a sequence number of 0 is executed last. The Configuration Rules (Config_Rules) object class provides an example of this process.

If device names are returned from the program invoked, the Configuration Manager finishes traversing the node tree before it invokes the next program. Note that some program names may not be associated with any devices, but they must be included to configure the system.

Invoking the Configuration Manager

During system boot time, the Configuration Manager is run in two phases. In phase 1, it configures the base devices needed to successfully start the system. These devices include the root volume group, which permits the configuration database to be read in from the root file system.

In phase 2, the Configuration Manager configures the remaining devices using the configuration database from the root file system. During this phase, different rules are used, depending on the key switch position on the front panel. If the key is in service position, the rules for service mode are used. Otherwise, the phase 2 rules are used.

The Configuration Manager can also be invoked during run time to configure all the detectable devices that may have been turned off at system boot or added after the system boot. In this case, the Configuration Manager uses the phase 2 rules.

Device Classes, Subclasses, and Types Overview

To manage the wide variety of devices it supports more easily, the operating system classifies them hierarchically. One advantage of this arrangement is that device methods and high-level commands can operate against a whole set of similar devices.

Devices are categorized into three main groups:

- Functional classes
- Functional subclasses
- Device types

Devices are organized into a set of *functional classes* at the highest level. From a user's point of view, all devices belonging to the same class perform the same functions. For example, all printer devices basically perform the same function of generating printed output.

However, devices within a class can have different interfaces. A class can therefore be partitioned into a set of *functional subclasses* in which devices belonging to the same subclass have similar interfaces. For example, serial printers and parallel printers form two subclasses of printer devices.

Finally, a device subclass is a collection of *device types*. All devices belonging to the same device type share the same manufacturer's model name and number. For example, 3812-2 (model 2 Pageprinter) and 4201 (Proprinter II) printers represent two types of printers.

Devices of the same device type can be managed by different drivers if the type belongs to more than one subclass. For example, the 4201 printer belongs to both the serial interface and parallel interface subclasses of the printer class, although there are different drivers for the two interfaces. However, a device of a particular class, subclass, and type can be managed by only one device driver.

Devices in the system are organized in clusters of tree structures known as *nodes*. For example, the system node consists of all the physical devices in the system. At the top of the node is the system device. Below the bus and connected to it are the

adapters. The bottom of the hierarchy contains the devices to which no other devices are connected. Most pseudo-devices, including LFT and PTY, are organized as separate nodes.

The Devices Graph: Examples of Connectivity and Dependence diagram on 82 illustrates this structure.

Writing a Device Method

Device methods are programs associated with a device that perform basic device configuration operations. These operations consist of defining, undefining, configuring, unconfiguring, and reconfiguring a device. Some devices also use optional start and stop operations.

There are five basic device methods:

Define	Creates a device instance in the Customized database.
Configure	Configures a device instance already represented in the Customized database. This method is responsible for making a device available for use in the system.
Change	Reconfigures a device by allowing device characteristics or attributes to be changed.
Unconfigure	Makes a configured device unavailable for use in the system. The device instance remains in the Customized database but must be reconfigured before it can be used.
Undefine	Deletes a device instance from the Customized database.

Some devices also require these two optional methods:

Stop	Provides the ability to stop a device without actually unconfiguring it. For example, a command can be issued to the device driver telling it to stop accepting normal I/O requests.
Start	Starts a device that has been stopped with the Stop method. For example, a command can be issued to the device driver informing it that it can now accept normal I/O requests.

Invoking Methods

One device method can invoke another device method. For instance, a Configure method for a device may need to invoke the Define method for child devices. The Change method can invoke the Unconfigure and Configure methods. To ensure proper operation, a method that invokes another method must always use the `odm_run_method` subroutine.

Example Methods

See the `/usr/samples` directory for example device method source code. These source code excerpts are provided for example purposes only. The examples do not function as written.

Understanding Device Methods Interfaces

Device methods are not executed directly from the command line. They are only invoked by the Configuration Manager at boot time or by the **cfgmgr**, **mkdev**, **chdev**, and **rmdev** configuration commands at run time. As a result, any device method you write should meet well-defined interfaces.

The parameters that are passed into the methods as well as the exit codes returned must both satisfy the requirements for each type of method. Additionally, some methods must write information to the **stdout** and **stderr** files.

These interfaces are defined for each of the device methods in the individual articles on writing each method.

To better understand how these interfaces work, one needs to understand, at least superficially, the flow of operations through the Configuration Manager and the run-time configuration commands.

Configuration Manager

The Configuration Manager begins by invoking a Node Configuration program listed in one of the rules in the Configuration Rules (Config_Rules) object class. A node is a group of devices organized into a tree structure representing the various interconnections of the devices. The Node Configuration program is responsible for starting the configuration process for a node. It does this by querying the Customized database to see if the device at the top of the node is represented in the database. If so, the program writes the logical name of the device to the **stdout** file and then returns to the Configuration Manager.

The Configuration Manager intercepts the Node Configuration program's **stdout** file to obtain the name of the device that was written. It then invokes the Configure method for that device. The device's Configure method performs the steps necessary to make the device available. If the device is not an intermediate one, the Configure method simply returns to the Configuration Manager. However, if the device is an intermediate device that has child devices (see "Understanding Device Dependencies and Child Devices" on page 93), the Configure method must determine whether any of the child devices need to be configured. If so, the Configure method writes the names of all the child devices to be configured to the **stdout** file and then returns to the Configuration Manager.

The Configuration Manager intercepts the Configure method's **stdout** file to retrieve the names of the children. It then invokes, one at a time, the Configure methods for each child device. Each of these Configure methods operates as described for the parent device. For example, it might simply exit when complete, or write to its **stdout** file a list of additional device names to be configured and then exit. The Configuration Manager will continue to intercept the device names written to the **stdout** file and to invoke the Configure methods for those devices until the Configure methods for all the devices have been run and no more names are written to the **stdout** file.

Run-Time Configuration Commands

User configuration commands invoke device methods during run time.

mkdev The **mkdev** command is invoked to define or configure, or define and configure, devices at run time. If just defining a device, the **mkdev** command invokes the Define method for the device. The Define method creates the customized device instance in the Customized Devices (CuDv) object class and writes the name assigned to the device to the **stdout** file. The **mkdev** command intercepts the device name written to the **stdout** file by the Define method to learn the name of the device. If user-specified attributes are supplied with the **-a** flag, the **mkdev** command then invokes the Change method for the device.

If defining and configuring a device, the **mkdev** command invokes the Define method, gets the name written to the **stdout** file with the Define method, invokes the Change method for the device if user-specified attributes were supplied, and finally invokes the device's Configure method.

If only configuring a device, the device must already exist in the CuDv object class and its name must be specified to the **mkdev** command. In this case, the **mkdev** command simply invokes the Configure method for the device.

chdev The **chdev** command is used to change the characteristics, or attributes, of a device. The device must already exist in the CuDv object class, and the name of the device must be supplied to the **chdev** command. The **chdev** command simply invokes the Change method for the device.

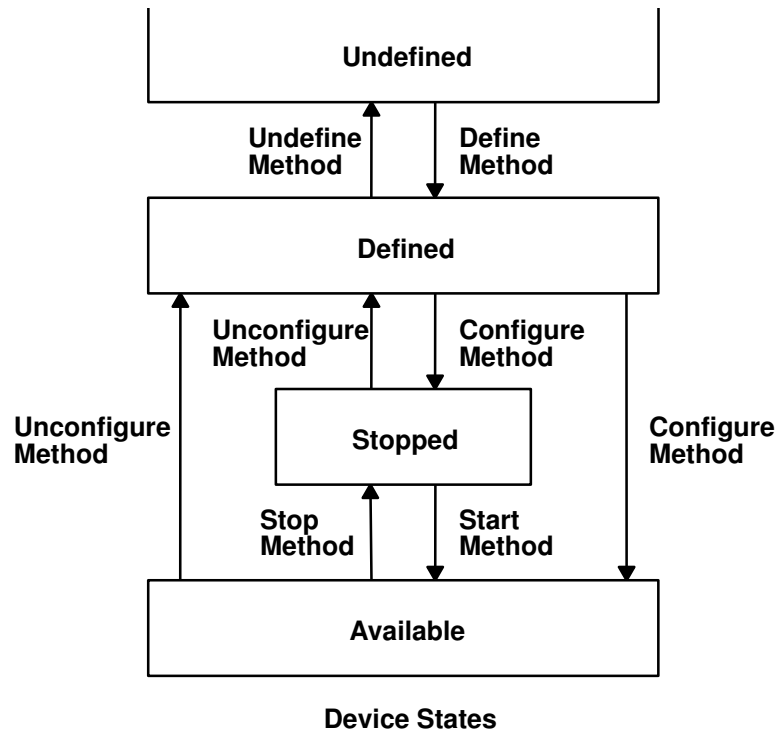
rmdev The **rmdev** command can be used to undefine or unconfigure, or unconfigure and undefine, a device. In all cases, the device must already exist in the CuDv object class and the name of the device must be supplied to the **rmdev** command. The **rmdev** command then invokes the Undefine method, the Unconfigure method, or the Unconfigure method followed by the Undefine method, depending on the function requested by the user.

cfgmgr The **cfgmgr** command can be used to configure all detectable devices that did not get configured at boot time. This might occur if the devices had been powered off at boot time. The **cfgmgr** command is the Configuration Manager and operates in the same way at run time as it does at boot time. The boot time operation is described in "Device Configuration Manager Overview" on page 85.

Understanding Device States

Device methods are responsible for changing the state of a device in the system. A device can be in one of four states as represented by the Device Status Flag descriptor in the device's object in the Customized Devices (CuDv) object class.

The Device States diagram on 91 illustrates both the possible states and the device methods that affect them.



- Defined** Represented in the Customized database, but neither configured nor available for use in the system.
- Available** Configured and available for use.
- Undefined** Not represented in the Customized database.
- Stopped** Configured, but not available for use by applications. (Optional state)

The Define method is responsible for creating a device instance in the Customized database and setting the state to Defined. The Configure method performs all operations necessary to make the device usable and then sets the state to Available.

The Change method usually does not change the state of the device. If the device is in the Defined state, the Change method applies all changes to the database and leaves the device defined. If the device is in the Available state, the Change method attempts to apply the changes to both the database and the actual device, while leaving the device available. However, if an error occurs when applying the changes to the actual device, the Change method may need to unconfigure the device, thus changing the state to Defined.

Any Unconfigure method you write must perform the operations necessary to make a device unusable. Basically, this method undoes the operations performed by the Configure method and sets the device state to Defined. Finally, the Undefine method actually deletes all information for a device instance from the Customized database, thus reverting the instance to the Undefined state.

The Stopped state is an optional state that some devices require. A device that supports this state needs Start and Stop methods. The Stop method changes the state from Available to Stopped. The Start method changes it from Stopped back to Available.

Adding an Unsupported Device to the System

The operating system provides support for a wide variety of devices. However, some devices are not currently supported. You can add a currently unsupported device only if you also add the necessary software to support it.

To add a currently unsupported device to your system, you may need to:

- Modify the Predefined database (see “Modifying the Predefined Database”)
- Add appropriate device methods (see “Adding Device Methods”)
- Add a device driver (see “Adding a Device Driver” on page 93)
- Use procedures (see “Using installp Procedures” on page 93)

Modifying the Predefined Database

To add a currently unsupported device to your system, you must modify the Predefined database. To do this, you must add information about your device to three predefined object classes:

- Predefined Devices (PdDv) object class
- Predefined Attribute (PdAt) object class
- Predefined Connection (PdCn) object class

To describe the device, you must add one object to the PdDv object class to indicate the class, subclass, and device type (see “Device Classes, Subclasses, and Types Overview” on page 87). You must also add one object to the PdAt object class for each device attribute, such as interrupt level or block size. Finally, you must add objects to the PdCn object class if the device is an intermediate device. If the device is an intermediate device, you must add an object for each different connection location on the intermediate device.

You can use the **odmadd** Object Data Manager (ODM) command from the command line or in a shell script to populate the necessary Predefined object classes from stanza files.

The Predefined database is shipped populated with supported devices. For some supported devices, such as serial and parallel printers and SCSI disks, the database also contains generic device objects. These generic device objects can be used to configure other similar devices that are not explicitly supported in the Predefined database.

For example, if you have a serial printer that closely resembles a printer supported by the system, and the system’s device driver for serial printers works on your printer, you can add the device driver as a printer of type **osp** (other serial printer). If these generic devices successfully add your device, you do not need to provide additional system software.

Adding Device Methods

You must add device methods when adding system support for a new device. Primary methods needed to support a device are:

- Define
- Configure

- Change
- Undefine
- Unconfigure

When adding a device that closely resembles devices already supported, you might be able to use one of the methods of the already supported device. For example, if you are adding a new type of SCSI disk whose interfaces are identical to supported SCSI disks, the existing methods for SCSI disks may work. If so, all you need to do is populate the Predefined database with information describing the new SCSI disk, which will be similar to information describing a supported SCSI disk.

If you need instructions on how to write a device method, see “Writing a Device Method” on page 88.

Adding a Device Driver

If you add a new device, you will probably need to add a device driver. However, if you are adding a new device that closely resembles an already supported device, you might be able to use the existing device driver. For example, when you are adding a new type of SCSI disk whose interfaces are identical to supported SCSI disks, the existing SCSI disk device driver may work.

Using installp Procedures

The **installp** procedures provide a method for adding the software and Predefined information needed to support your new device. You may need to write shell scripts to perform tasks such as populating the Predefined database.

Understanding Device Dependencies and Child Devices

The dependencies that one device has on another can be represented in the configuration database in two ways. One way usually represents physical connections such as a keyboard device connected to a particular keyboard adapter. The keyboard device has a dependency on the keyboard adapter in that it cannot be configured until after the adapter is configured. This relationship is usually referred to as a parent-child relationship, with the adapter as parent and the keyboard device as child. These relationships are represented with the Parent Device Logical Name and Location Where Device Is Connected descriptors in the Customized Devices (CuDv) object class.

The second method represents a logical connection. A device method can add an object identifying both a dependent device and the device upon which it depends to the Customized Dependency (CuDep) object class (see “Writing a Device Method” on page 88). The dependent device is considered to *have* a dependency, and the depended-upon device is considered to *be* a dependency. CuDep objects are usually added to the database to represent a situation in which one device requires access to another device. For example, the hft0 lft0 device depends upon a particular keyboard or display device.

These two types of dependencies differ significantly. The configuration process uses parent-child dependencies at boot time to configure all devices that make up a node. The CuDep dependency is usually only used by a device’s Configure method to retrieve the names of the devices on which it depends. The Configure method can then check to see if those devices exist.

For device methods, the parent-child relationship is the more important. Parent-child relationships affect device-method activities in these ways:

- A parent device cannot be unconfigured if it has a configured child.
- A parent device cannot be undefined if it has a defined or configured child.
- A child device cannot be defined if the parent is not defined or configured.
- A child device cannot be configured if the parent is not configured.
- A parent device's configuration cannot be changed if it has a configured child. This guarantees that the information about the parent that the child's device driver may be using remains valid.

However, when a device is listed as a dependency of another device in the CuDep object class, the only effect is to prevent the depended-upon device from being undefined. The name of the dependency is important to the dependent device. If the depended-upon device were allowed to be undefined, a third device could be defined and assigned the same name.

Writers of Unconfigure and Change methods for a depended-upon device should not worry about whether the device is listed as a dependency. If the depended-upon device is actually open by the other device, the Unconfigure and Change operations will fail because their device is busy. But if the depended-upon device is *not* currently open, the Unconfigure or Change operations can be performed without affecting the dependent device.

The possible parent-child connections are defined in the Predefined Connection (PdCn) object class. Each predefined device type that can be a parent device is represented in this object class. There is an object for each connection location (such as slots or ports) describing the subclass of devices that can be connected at that location. The subclass is used to identify each device since it indicates the devices' connection type (for example, SCSI or rs232).

There is no corresponding predefined object class describing the possible CuDep dependencies. A device method can be written so that it already knows what the dependencies are. If predefined data is required, it can be added as predefined attributes for the dependent device in the Predefined Attribute (PdAt) object class.

The "Devices Graph" diagram on 82 provides an example of device dependencies and connections in the system.

Accessing Device Attributes

The predefined device attributes for each type of predefined device are stored in the Predefined Attribute (PdAt) object class. The objects in the PdAt object class identify the default values as well as other possible values for each attribute. The Customized Attribute (CuAt) object class contains only attributes for customized device instances that have been changed from their default values.

When a customized device instance is created by a Define method, its attributes assume the default values. As a result, no objects are added to the CuAt object class for the device. If an attribute for the device is changed from the default value by the Change method, an object to describe the attribute's current value is added to the CuAt object class for the attribute. If the attribute is subsequently changed back to the default value, the Change method deletes the CuAt object for the attribute.

Any device methods that need the current attribute values for a device must access both the PdAt and CuAt object classes. If an attribute appears in the CuAt object class, then the associated object identifies the current value. Otherwise, the default value from the PdAt attribute object identifies the current value.

Modifying an Attribute Value

When modifying an attribute value, methods you write must obtain the objects for that attribute from both the PdAt and CuAt object classes.

Any method you write must be able to handle the following four scenarios:

- If the new value differs from the default value and no object currently exists in the CuAt object class, any method you write must add an object into the CuAt object class to identify the new value.
- If the new value differs from the default value and an object already exists in the CuAt object class, any method you write must update the CuAt object with the new value.
- If the new value is the same as the default value and an object exists in the CuAt object class, any method you write must delete the CuAt object for the attribute.
- If the new value is the same as the default value and no object exists in the CuAt object class, any method you write does not need to do anything.

Your methods can use the **getattr** and **putattr** subroutines to get and modify attributes. The **getattr** subroutine checks both the PdAt and CuAt object classes before returning an attribute to you. It always returns the information in the form of a CuAt object even if returning the default value from the PdAt object class.

Use the **putattr** subroutine to modify these attributes.

Device Dependent Structure (DDS) Overview

A *device dependent structure* (DDS) contains information that describes a device instance to the device driver. It typically contains information about device-dependent attributes as well as other information the driver needs to communicate with the device. In many cases, information about a device's parent is included. (For instance, a driver needs information about the adapter and the bus the adapter is plugged into to communicate with a device connected to an adapter.)

A device's DDS is built each time the device is configured. The Configure method can fill in the DDS with fixed values, computed values, and information from the Configuration database. Most of the information from the Configuration database usually comes from the attributes for the device in the Customized Attribute (CuAt) object class, but can come from any of the object classes. Information from the database for the device's parent device or parent's parent device can also be included. The DDS is passed to the device driver with the **SYS_CFGDD** flag of the **sysconfig** subroutine, which calls the device driver's **ddconfig** subroutine with the **CFG_INIT** command.

How the Change Method Updates the DDS

The Change method is invoked when changing the configuration of a device. The Change method must ensure consistency between the Configuration database and the view that any device driver may have of the device. This is accomplished by:

1. Not allowing the configuration to be changed if the device has configured children; that is, children in either the Available or Stopped states. This ensures that a DDS built using information in the database about a parent device remains valid because the parent cannot be changed.
2. If a device has a device driver and the device is in either the Available or Stopped state, the Change method must communicate to the device driver any changes that would affect the DDS. This may be accomplished with `ioctl` operations, if the device driver provides the support to do so. It can also be accomplished by taking the following steps:
 - a. Terminating the device instance by calling the `sysconfig` subroutine with the `SYS_CFGDD` operation. This operation calls the device driver's `ddconfig` subroutine with the `CFG_TERM` command.
 - b. Rebuilding the DDS using the changed information.
 - c. Passing the new DDS to the device driver by calling the `sysconfig` subroutine `SYS_CFGDD` operation. This operation then calls the `ddconfig` subroutine with the `CFG_INIT` command.

Many Change methods simply invoke the device's Unconfigure method, apply changes to the database, and then invoke the device's Configure method. This process ensures the two stipulated conditions since the Unconfigure method, and thus the change, will fail, if the device has Available or Stopped children. Also, if the device has a device driver, its Unconfigure method terminates the device instance. Its Configure method also rebuilds the DDS and passes it to the driver.

Guidelines for DDS Structure

There is no single defined DDS format. Writers of device drivers and device methods must agree upon a particular device's DDS format. When obtaining information about a parent device, you may want to group that information together in the DDS.

When building a DDS for a device connected to an adapter card, you will typically need the following adapter information:

slot number	Obtained from the <code>connwhere</code> descriptor of the adapter's Customized Device (CuDv) object.
bus resources	Obtained from attributes for the adapter in the Customized Attribute (CuAt) or Predefined Attribute (PdAt) object classes. These include attributes for bus interrupt levels, interrupt priorities, bus memory addresses, bus I/O addresses, and DMA arbitration levels.

These two attributes must be obtained for the adapter's parent bus device:

bus_id	Identifies the I/O bus. This field is needed by the device driver to access the I/O bus.
bus_type	Identifies the type of bus such as a Micro Channel bus or a PC AT bus.

Note: The `getattr` device configuration subroutine should be used whenever attributes are obtained from the Configuration database. This subroutine returns the Customized attribute value if the attribute is represented in the Customized Attribute object class. Otherwise, it returns the default value from the Predefined Attribute object class.

Finally, a DDS generally includes the device's logical name. This is used by the device driver to identify the device when logging an error for the device.

Example of DDS

The following example provides a guide for using DDS format.

```
/* Device DDS */
struct device_dds {
    /* Bus information */
    ulong bus_id;          /* I/O bus id */
    ushort us_type;       /* Bus type, i.e. BUS_MICRO_CHANNEL*/
    /* Adapter information */
    int slot_num;         /* Slot number */
    ulong io_addr_base;   /* Base bus i/o address */
    int bus_intr_lvl;     /* bus interrupt level */
    int intr_priority;    /* System interrupt priority */
    int dma_lvl;          /* DMA arbitration level */
    /* Device specific information */
    int block_size;       /* Size of block in bytes */
    int abc_attr;         /* The abc attribute */
    int xyz_attr;         /* The xyz attribute */
    char resource_name[16]; /* Device logical name */
};
```

List of Device Configuration Commands

The high-level device configuration commands are:

chdev	Changes a device's characteristics.
lsdev	Displays devices in the system and their characteristics.
mkdev	Adds a device to the system.
rmdev	Removes a device from the system.
lsattr	Displays attribute characteristics and possible values of attributes for devices in the system.
lsconn	Displays the connections a given device, or kind of device, can accept.
lsparent	Displays the possible parent devices that accept a specified connection type or device.
cfgmgr	Configures devices by running the programs specified in the Configuration Rules (Config_Rules) object class.

The low-level device configuration commands are:

bootlist	Alters the list of boot devices seen by ROS when the machine boots.
restbase	Reads the base customized information from the boot image and restores it into the Device Configuration database used during system boot phase 1.
savebase	Saves information about base customized devices in the Device Configuration Database onto the boot device.

Associated commands are:

devnm Names a device.

mknod Creates a special file (directory entry and i-node).
lscfg Displays diagnostic information about a device.

List of Device Configuration Subroutines

Following are the preexisting conditions for using the device configuration library subroutines:

- The caller has initialized the Object Data Manager (ODM) before invoking any of these library subroutines. This is done using the **initialize_odm** subroutine. Similarly, the caller must terminate the ODM (using the **terminate_odm** subroutine) after these library subroutines have completed. Only the **attrval** subroutine does not require initialization and termination.
- Since all of these library subroutines (except the **attrval**, **getattr**, and **putattr** subroutines) access the Customized Device Driver (CuDvDr) object class, this class must be exclusively locked and unlocked at the proper times. The application does this by using the **odm_lock** and **odm_unlock** subroutines. In addition, those library subroutines that access the CuDvDr object class exclusively lock this class with their own internal locks.

Following are the device configuration library subroutines:

attrval	Verifies that attributes are within range.
busresolve	Allocates bus resources for Micro channel adapters.
genmajor	Generates the next available major number for a device.
genminor	Generates the smallest unused minor number or a requested minor number for a device.
genseq	Generates a sequence number.
getattr	Returns attribute objects from either the Predefined Attribute (PdAt) or Customized Attribute (CuAt) object class, or both.
getminor	Gets from the CuDvDr object class the minor numbers for a given major number.
loadext	Loads or unloads and binds or unbinds device drivers to or from the kernel.
putattr	Updates attribute information in the CuAt object class or creates a new object for the attribute information.
reldevno	Releases the minor number or major number, or both, for a device instance.
relmajor	Releases the major number associated with a specific device driver instance.

Chapter 7. Communications I/O Subsystem

The Communication I/O Subsystem design introduces a more efficient, streamlined approach to attaching data link control (DLC) processes to communication and LAN adapters.

The Communication I/O Subsystem consists of one or more physical device handlers (PDHs) that control various communication adapters. The interface to the physical device handlers can support any number of processes, the limit being device-dependent.

Note: A PDH, as used for the Communications I/O, provides both the device head role for interfacing to users, and the device handler role for performing I/O to the device.

A communications PDH is a special type of multiplexed character device driver (see “Communications Physical Device Handler Model Overview” on page 100). Information common to all communications device handlers is discussed here. Additionally, individual communications PDHs have their own adapter-specific sets of information. Refer to the following to learn more about the adapter types:

- “MPQP Device Handler Interface Overview” on page 104
- “Serial Optical Link Device Handler Overview” on page 108

Each adapter type requires a device driver. Each PDH can support one or more adapters of the same type.

There are two interfaces a user can use to access a PDH. One is from a user-mode process (application space), and the other is from a kernel-mode process (within the kernel).

User-Mode Interface to a Communications PDH

The user-mode process uses system calls (**open**, **close**, **select**, **poll**, **ioctl**, **read**, **write**) to interface to the PDH to send or receive data. The **poll** or **select** subroutine notifies a user-mode process of available receive data, available transmit, and status and exception conditions.

Kernel-Mode Interface to a Communications PDH

The kernel-mode interface to a communications PDH differs from the interface supported for a user-mode process in the following ways:

- Kernel services are used instead of system calls. This means that, for example, the **fp_open** kernel service is used instead of the **open** subroutine. The same holds true for the **fp_close**, **fp_ioctl**, and **fp_write** kernel services.
- The **ddread** entry point, **ddselect** entry point, and **CIO_GET_STAT** (Get Status) **ddioctl** operation are not supported in kernel mode. Instead, kernel-mode processes specify at open time the addresses of their own procedures for handling receive data available, transmit available and status or exception conditions. The PDH directly calls the appropriate procedure, whenever that

condition arises. These kernel procedures must execute and return quickly since they are executing within the priority of the PDH.

- The **ddwrite** operation for a kernel-mode process differs from a user-mode process in that there are two ways to issue a **ddwrite** operation to transmit data:
 - Transmit each buffer of data with the **fp_write** kernel service.
 - Use the fast write operation, which allows the user to directly call the **ddwrite** operation (no context switching) for each buffer of data to be transmitted. This operation helps increase the performance of transmitted data. A **fp_ioctl** (**CIO_GET_FASTWRT**) kernel service call obtains the functional address of the write function. This address is used on all subsequent write function calls. Support of the fast write operation is optional for each device.

CDLI Device Drivers

Some device drivers have a different design and use the services known as Common Data Link Interface (CDLI). The following are device drivers that use CDLI:

- Forum-Compliant ATM LAN Emulation Device Driver
- Fiber Distributed Data Interface (FDDI) Device Driver
- High-Performance (8fc8) Token-Ring Device Driver
- High-Performance (8fa2) Token-Ring Device Driver
- Ethernet Device Drivers

Communications Physical Device Handler Model Overview

A physical device handler (PDH) must provide eight common entry points. An individual PDH names its entry points by placing a unique identifier in front of the supported command type. The following are the required eight communications PDH entry points:

ddconfig	Performs configuration functions for a device handler. Supported the same way that the common ddconfig entry point is.
ddmpx	Allocates or deallocates a channel for a multiplexed device handler. Supported the same way as the common ddmpx device handler entry point.
ddopen	Performs data structure allocation and initialization for a communications PDH. Supported the same way as the common ddopen entry point. Time-consuming tasks, such as port initialization and connection establishment, are deferred until the (CIO_START) ddioctl call is issued. A PDH can support multiple users of a single port.
ddclose	Frees up system resources used by the specified communications device until they are needed again. Supported the same way as the common ddclose entry point.
ddwrite	Queues a message for transmission or blocks until the message can be queued. The ddwrite entry point can attempt to queue a transmit request (nonblocking) or wait for it to be queued (blocking), depending on the setting of the DNDELAY flag. The caller has the additional option of requesting an asynchronous acknowledgment when the transmission actually completes.
ddread	Returns a message of data to a user-mode process. Supports blocking or nonblocking reads depending on the setting of the DNDELAY flag. A blocking read request does not return to the caller until data is available. A nonblocking read returns with a message of data if it is immediately available. Otherwise, it returns a length of 0 (zero).

ddselect	Checks to see if a specified event or events has occurred on the device for a user-mode process. Supported the same way as the common ddselect entry point.
ddioctl	Performs the special I/O operations requested in an ioctl subroutine. Supported the same way as the common ddioctl entry point. In addition, a communications PDH must support the following four options: <ul style="list-style-type: none"> • CIO_START • CIO_HALT • CIO_QUERY • CIO_GET_STAT

Individual PDHs can add additional commands. Hardware initialization and other time-consuming activities, such as call establishment, are performed during the **CIO_START** operation.

Use of mbuf Structures in the Communications PDH

PDHs use **mbuf** structures to buffer send and receive data. These structures allow the PDH to gather data when transmitting frames and scatter for receive operations. The **mbuf** structures are internal to the kernel and are used only by kernel-mode processes and PDHs.

PDHs and kernel-mode processes require a set of utilities for obtaining and returning **mbuf** structures from a buffer pool.

Kernel-mode processes use the Berkeley **mbuf** scheme for transmit and receive buffers. The structure for an **mbuf** is defined in the `/usr/include/sys/mbuf.h` file.

Common Communications Status and Exception Codes

In general, communication device handlers return codes from a group of common exception codes. However, device handlers for specific communication devices can return device-specific exception codes. Common exception codes are defined in the `/usr/include/sys/comio.h` file and include the following:

CIO_OK	Indicates that the operation was successful.
CIO_BUF_OVFLW	Indicates that the data was lost due to buffer overflow.
CIO_HARD_FAIL	Indicates that a hardware failure was detected.
CIO_NOMBUF	Indicates that the operation was unable to allocate mbuf structures.
CIO_TIMEOUT	Indicates that a time-out error occurred.
CIO_TX_FULL	Indicates that the transmit queue is full.
CIO_NET_RCVRY_ENTER	Enters network recovery.
CIO_NET_RCVRY_EXIT	Indicates the device handler is exiting network recovery.
CIO_NET_RCVRY_MODE	Indicates the device handler is in Recovery mode.
CIO_INV_CMD	Indicates that an invalid command was issued.
CIO_BAD_MICROCODE	Indicates that the microcode download failed.
CIO_NOT_DIAG_MODE	Indicates that the command could not be accepted because the adapter is not open in Diagnostic mode.
CIO_BAD_RANGE	Indicates that the parameter values have failed a range check.
CIO_NOT_STARTED	Indicates that the command could not be accepted because the device has not yet been started by the first call to CIO_START operation.
CIO_LOST_DATA	Indicates that the receive packet was lost.

CIO_LOST_STATUS	Indicates that a status block was lost.
CIO_NETID_INV	Indicates that the network ID was not valid.
CIO_NETID_DUP	Indicates that the network ID was a duplicate of an existing ID already in use on the network.
CIO_NETID_FULL	Indicates that the network ID table is full.

Status Blocks for Communications Device Handlers Overview

Status blocks are used to communicate status and exception information.

User-mode processes receive a status block whenever they request a **CIO_GET_STAT** operation. A user-mode process can wait for the next available status block by issuing a **ddselect** entry point with the specified **POLLPRI** event.

A kernel-mode process receives a status block through the **stat_fn** procedure. This procedure is specified when the device is opened with the **ddopen** entry point.

Status blocks contain a code field and possible options. The code field indicates the type of status block code (for example, **CIO_START_DONE**). A status block's options depend on the block code. The C structure of a status block is defined in the **/usr/include/sys/comio.h** file.

The following are the six common status codes:

- **CIO_START_DONE** (see “**CIO_START_DONE**”)
- **CIO_HALT_DONE** (see “**CIO_HALT_DONE**”)
- **CIO_TX_DONE** (see “**CIO_TX_DONE**” on page 103)
- **CIO_NULL_BLK** (see “**CIO_NULL_BLK**” on page 103)
- **CIO_LOST_STATUS** (see “**CIO_LOST_STATUS**” on page 103)
- **CIO_ASYNC_STATUS** (see “**CIO_ASYNC_STATUS**” on page 103)

Additional device-dependent status block codes may be defined.

CIO_START_DONE

This block is provided by the device handler when the **CIO_START** operation completes:

<code>option[0]</code>	The CIO_OK or CIO_HARD_FAIL status/exception code from the common or device-dependent list. See 101 or 101.
<code>option[1]</code>	The low-order two bytes are filled in with the <code>netid</code> field. This field is passed when the CIO_START operation is invoked.
<code>option[2]</code>	Device-dependent.
<code>option[3]</code>	Device-dependent.

CIO_HALT_DONE

This block is provided by the device handler when the **CIO_HALT** operation completes:

<code>option[0]</code>	The CIO_OK status/exception code from the common or device-dependent list (see 101).
------------------------	---

- option[1] The low-order two bytes are filled in with the `netid` field. This field is passed when the `CIO_START` operation is invoked.
- option[2] Device-dependent.
- option[3] Device-dependent.

CIO_TX_DONE

The following block is provided when the physical device handler (PDH) is finished with a transmit request for which acknowledgment was requested:

- option[0] The `CIO_OK` or `CIO_TIMEOUT` status/exception code from the common or device-dependent list. See 101 or 101.
- option[1] The `write_id` field specified in the `write_extension` structure passed in the `ext` parameter to the `ddwrite` entry point.
- option[2] For a kernel-mode process, indicates the `mbuf` pointer for the transmitted frame.
- option[3] Device-dependent.

CIO_NULL_BLK

This block is returned whenever a status block is requested but there are none available:

- option[0] Not used
- option[1] Not used
- option[2] Not used
- option[3] Not used

CIO_LOST_STATUS

This block is returned once after one or more status blocks is lost due to status queue overflow. The `CIO_LOST_STATUS` block provides the following:

- option[0] Not used
- option[1] Not used
- option[2] Not used
- option[3] Not used

CIO_ASYNC_STATUS

This status block is used to return status and exception codes that occur unexpectedly:

- option[0] The `CIO_HARD_FAIL` or `CIO_LOST_DATA` status/exception code from the common or device-dependent list. See 101 or 101.
- option[1] Device-dependent
- option[2] Device-dependent
- option[3] Device-dependent

MPQP Device Handler Interface Overview

The Multiprotocol Quad Port (MPQP) device handler is a component of the communication I/O subsystem (see “Chapter 7. Communications I/O Subsystem” on page 99). The MPQP device handler interface is made up of the following eight entry points:

mpclose	Resets the MPQP device to a known state and returns system resources back to the system on the last close for that adapter. The port no longer transmits or receives data.
mpconfig	Provides functions for initializing and terminating the MPQP device handler and adapter.
mpioctl	Provides the following functions for controlling the MPQP device: CIO_START Initiates a session with the MPQP device handler. CIO_STOP Ends a session with the MPQP device handler. CIO_COUNTER Reads the counter values accumulated by the MPQP device handler. CIO_GET_STATUS Gets the status of the current MPQP adapter and device handler. MP_START_AR Puts the MPQP port into Autoresponse mode. MP_STOP_AR Permits the MPQP port to exit Autoresponse mode. MP_CHG_PARMS Permits the data link control (DLC) to change certain profile parameters after the MPQP device has been started.
mpopen	Opens a channel on the MPQP device for transmitting and receiving data.
mpmpx	Provides allocation and deallocation of a channel.
mpread	Provides the means for receiving data to the MPQP device.
mpselect	Provides the means for determining which specified events have occurred on the MPQP device.
mpwrite	Provides the means for transmitting data to the MPQP device.

Binary Synchronous Communication (BSC) with the MPQP Adapter

The MPQP adapter software performs low-level BSC frame-type determination to facilitate character parsing at the kernel-mode process level. Frames received without errors are parsed. A message type is returned in the status field of the extension block along with a pointer to the receive buffer. The message type indicates the type of frame that was received.

For control frames that only contain control characters, the message type is returned and no data is transferred from the board. For example, if an ACK0 was received, the message type MP_ACK0 is returned in the status field of the extension block. In addition, a NULL pointer for the receive buffer is returned. If an error occurs, the error status is logged by the device driver. Unlogged buffer overrun errors are an exception.

Note: In BSC communications, the caller receives either a message type or an error status.

Read operations must be performed using the `readx` subroutine since the `read_extension` structure is needed to return BSC function results.

BSC Message Types Detected by the MPQP Adapter

BSC message types are defined in the `/usr/include/sys/mpqp.h` file. The MPQP adapter can detect the following message types:

MP_ACK0	MP_DISC	MP_STX_ETX
MP_ACK1	MP_SOH_ITB	MP_STX_ENQ
MP_WACK	MP_SOH_ETB	MP_DATA_ACK0
MP_NAK	MP_SOH_ETX	MP_DATA_ACK1
MP_ENQ	MP_SOH_ENQ	MP_DATA_NAK
MP_EOT	MP_STX_ITB	MP_DATA_ENQ
MP_RVI	MP_STX_ETB	

BSC Receive Errors Logged by the MPQP Adapter

The MPQP adapter detects many types of receive errors. As errors occur they are logged and the appropriate statistical counter is incremented. The kernel-mode process is not notified of the error. The following are the possible BSC receive errors logged by the MPQP adapter:

- Receive overrun because the card did not keep up with line data.
- Driver did not supply buffer in time for data.
- A cyclical redundancy check (CRC) or longitudinal redundancy check (LRC) framing error.
- Parity error.
- Clear to Send (CTS) timeout while the adapter is in Autoreponse mode.
- Data synchronization lost.
- ID field greater than 15 bytes (BSC).
- Invalid pad at end of frame (BSC).
- Unexpected or invalid data (BSC).

If status and data information are available, but no extension block is provided, the `read` operation returns the data, but not the status information.

Note: Errors, such as buffer overflow errors, can occur during the read data operation. In these cases, the return value is the byte count. Therefore, status should be checked even if no `errno` global value is returned.

Description of the MPQP Card

The MPQP card is a 4-port multiprotocol adapter that supports BSC and SDLC on the EIA232-D, EIA422-A, X.21, and V.35 physical interfaces. When using the X.21 physical interface, X.21 centralized multipoint operation on a leased-circuit public data network is not supported. The MPQP card uses the microchannel bus to communicate with the adapter programmed I/O (PIO) and first party DMA (bus master).

The adapter has 512K bytes of RAM and an Intel 80C186 processor. There are 16 dedicated DMA channels between the RAM and the physical ports. The drivers and receivers for each of the electrical interfaces reside on a daughter board that is joined to the base card with two 60-pin connectors.

A shielded cable attaches to the 78-pin D-shell connector on the daughter board and routes all signals to a fan-out box (FOB). The FOB has nine standard connectors that support each possible configuration on each port. Standard 15-pin or 25-pin cables are used between the FOB and the modem for each electrical interface.

The following are the interfaces available on each port:

Port Configurations				
Number	Port-0	Port-1	Port-2	Port-3
1	EIA-232D	EIA-232D	EIA-232D	EIA-232D
2	EIA-422A	EIA-232D	EIA-232D	EIA-232D
3	V.35 EIA-232D	EIA-232D V.35	EIA-232D EIA-232D	EIA-232D EIA-232D
4	X.21	EIA-232D	EIA-232D	EIA-232D
5	EIA-422A	V.35	EIA-232D	EIA-232D
6	V.35	V.35	EIA-232D	EIA-232D
7	X.21	V.35	EIA-232D	EIA-232D
8	EIA-232D	EIA-232D	EIA-422A	EIA-232D
9	EIA-422A	EIA-232D	EIA-422A	EIA-232D
10	V.35 EIA-232D	EIA-232D V.35	EIA-422A EIA-422A	EIA-232D EIA-232D
11	X.21	EIA-232D	EIA-422A	EIA-232D
12	EIA-422A	V.35	EIA-422A	EIA-232D
13	V.35	V.35	EIA-422A	EIA-232D
14	X.21	V.35	EIA-422A	EIA-232D

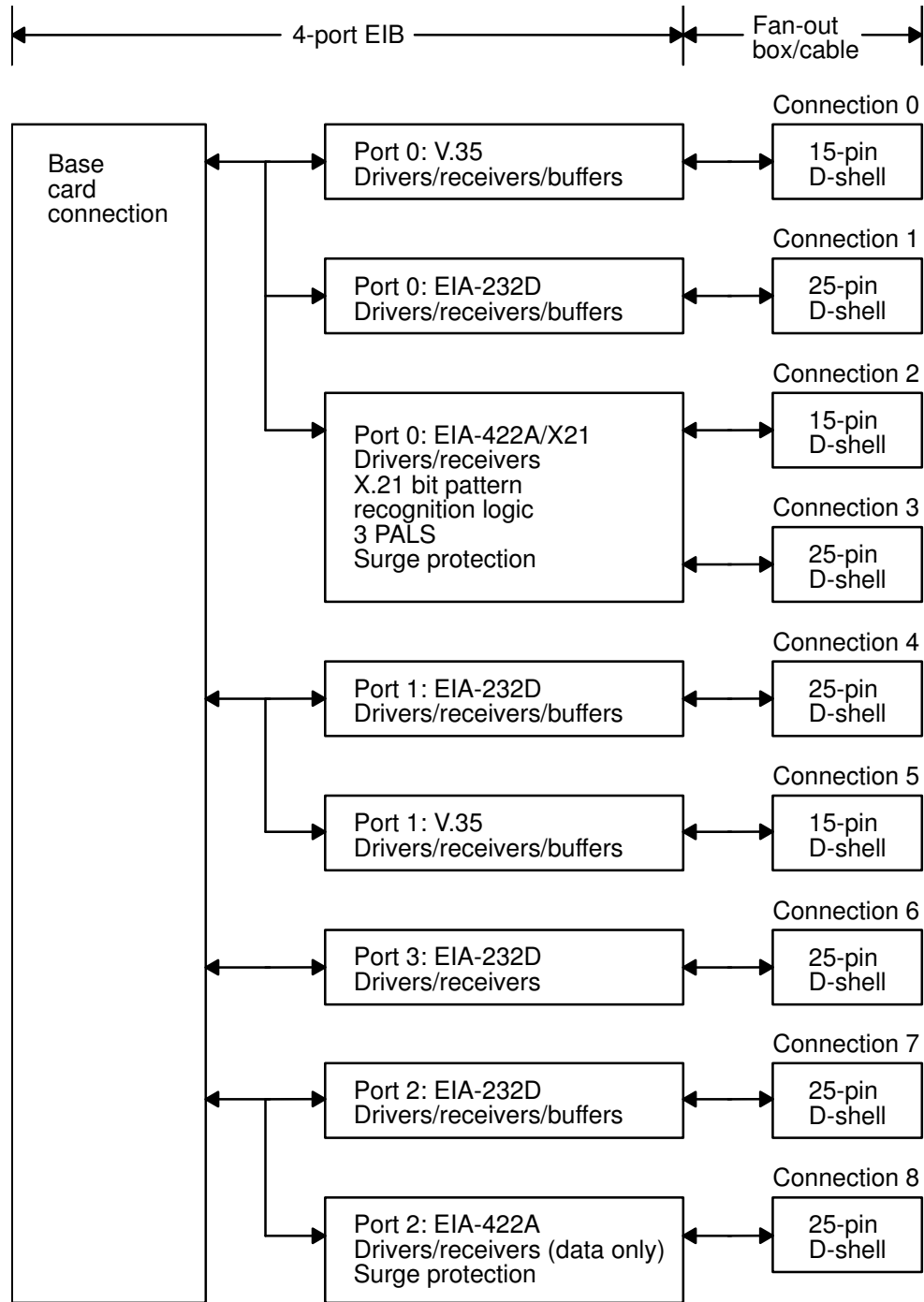
Port 0 EIA232-D, EIA422-A, X.21, and V.35. This port has the highest DMA priority. The EIA-422A interface on this port has data and clock signals.

Port 1 EIA232-D and V.35.

Port 2 EIA232-D and EIA422-A (data only). The EIA-422A interface on Port 2 only has data signals.

Port 3 EIA232-D. This port has the lowest priority.

The modem interfaces are supported by each physical interface shown in the following diagram.



Block Diagram

Call Establishment Protocol			
Physical Interface	Leased	Manual Switched	Autodial
EIA232-D	X	X	X
EIA422-A	X		
V.35	X		
X.21	X		X*

* Adheres to CCITT X.21 dial specifications.

The following diagram depicts the mapping of physical interfaces to the FOB connectors.

Serial Optical Link Device Handler Overview

The serial optical link (SOL) device handler is a component of the communication I/O subsystem. The device handler can support one to four serial optical ports. An optical port consists of two separate pieces. The serial link adapter is on the system planar and is packaged with two to four adapters in a single chip. The serial optical channel converter plugs into a slot on the system planar and provides two separate optical ports.

Special Files

There are two separate interfaces to the serial optical link device handler. The special file `/dev/ops0` provides access to the optical port subsystem. An application that opens this special file has access to all the ports, but it does not need to be aware of the number of ports available. Each write operation includes a destination processor ID. The device handler sends the data out the correct port to reach that processor. In case of a link failure, the device handler uses any link that is available.

The `/dev/op0`, `/dev/op1`, ..., `/dev/opn` special files provide a diagnostic interface to the serial link adapters and the serial optical channel converters. Each special file corresponds to a single optical port that can only be opened in Diagnostic mode. A diagnostic open allows the diagnostic ioctls to be used, but normal reads and writes are not allowed. A port that is open in this manner cannot be opened with the `/dev/ops0` special file. In addition, if the port has already been opened with the `/dev/ops0` special file, attempting to open a `/dev/opx` special file will fail unless a forced diagnostic open is used.

Entry Points

The SOL device handler interface consists of the following entry points:

<code>sol_close</code>	Resets the device to a known state and frees system resources.
<code>sol_config</code>	Provides functions to initialize and terminate the device handler, and query the vital product data (VPD).
<code>sol_fastwrt</code>	Provides the means for kernel-mode users to transmit data to the SOL device driver.

sol_ioctl	Provides various functions for controlling the device. The valid sol_ioctl operations are: CIO_GET_FASTWRT Gets attributes needed for the sol_fastwrt entry point. CIO_GET_STAT Gets the device status. CIO_HALT Halts the device. CIO_QUERY Queries device statistics. CIO_START Starts the device. IOCINFO Provides I/O character information. SOL_CHECK_PRID Checks whether a processor ID is connected. SOL_GET_PRIDS Gets connected processor IDs.
sol_mpx	Provides allocation and deallocation of a channel.
sol_open	Initializes the device handler and allocates the required system resources.
sol_read	Provides the means for receiving data.
sol_select	Determines if a specified event has occurred on the device.
sol_write	Provides the means for transmitting data.

Configuring the Serial Optical Link Device Driver

When configuring the serial optical link (SOL) device driver, consider the physical and logical devices, and changeable attributes of the SOL subsystem (see “Changeable Attributes of the Serial Optical Link Subsystem” on page 110).

Physical and Logical Devices

The SOL subsystem consists of several physical and logical devices in the ODM configuration database:

Device	Description
slc (serial link chip)	There are two serial link adapters in each COMBO chip. The slc device is automatically detected and configured by the system.
otp (optic two-port card)	Also known as the serial optical channel converter. There is one SOCC possible for each slc . The otp device is automatically detected and configured by the system.
op (optic port)	There are two optic ports per otp . The op device is automatically detected and configured by the system.

Device	Description
ops (optic port subsystem)	This is a logical device. There is only one created at any time. The ops device requires some additional configuration initially, and is then automatically configured from that point on. The /dev/ops0 special file is created when the ops device is configured. The ops device cannot be configured when the processor ID is set to -1.

Changeable Attributes of the Serial Optical Link Subsystem

The system administrator can change the following attributes of the serial optical link subsystem:

Note: If your system uses serial optical link to make a direct, point-to-point connection to another system or systems, special conditions apply. You must start interfaces on two systems at approximately the same time, or a method error occurs. If you wish to connect to at least one machine on which the interface has already been started, this is not necessary.

Processor ID	This is the address by which other machines connected by means of the optical link address this machine. The processor ID can be any value in the range of 1 to 254. To avoid a conflict on the network, this value is initially set to -1, which is not valid, and the ops device cannot be configured. Note: If you are using TCP/IP over the serial optical link, the processor ID must be the same as the low-order octet of the IP address. It is not possible to successfully configure TCP/IP if the processor ID does not match.
Receive Queue Size	This is the maximum number of packets that is queued for a user-mode caller. The default value is 30 packets. Any integer in the range from 30 to 150 is valid.
Status Queue Size	This is the maximum number of status blocks that will be queued for a user-mode caller. The default value is 10. Any integer in the range from 3 to 20 is valid.

The standard SMIT interface is available for setting these attributes, listing the serial optical channel converters, handling the initial configuration of the **ops** device, generating a trace report, generating an error report, and configuring TCP/IP.

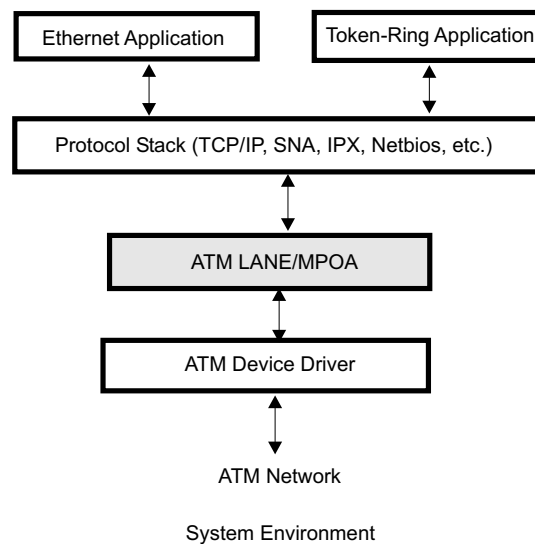
Forum-Compliant ATM LAN Emulation Device Driver

Note: The ATM LAN Emulation device driver is available for systems running AIX Version 4.1.5 (or later).

The Forum-Compliant ATM LAN Emulation (LANE) device driver allows communications applications and access methods that would normally operate over local area network (LAN) attachments to operate over high-speed ATM networks. This ATM LANE function supports LAN Emulation Client (LEC) as specified in *The ATM Forum Technical Committee LAN Emulation Over ATM Version 1.0*, as well as MPOA Client (MPC) via a subset of *ATM Forum LAN Emulation Over ATM Version 2 - LUNI Specification*, and *ATM Forum Multi-Protocol Over ATM Version 1.0*.

The Forum-Compliant ATM LAN Emulation (LANE) device driver allows communications applications and access methods that would normally operate over local area network (LAN) attachments to operate over high-speed ATM networks. This ATM LANE function supports LAN Emulation Client (LEC) as specified in *The ATM Forum Technical Committee LAN Emulation Over ATM Version 1.0*.

The ATM LANE device driver emulates the operation of Standard Ethernet, IEEE 802.3 Ethernet, and IEEE 802.5 Token Ring LANs. It encapsulates each LAN packet and transfers its LAN data over an ATM network at up to 155 megabits per second. This data can also be bridged transparently to a traditional LAN with ATM/LAN bridges such as the IBM 2216. (See the "System Environment" illustration on 111.)

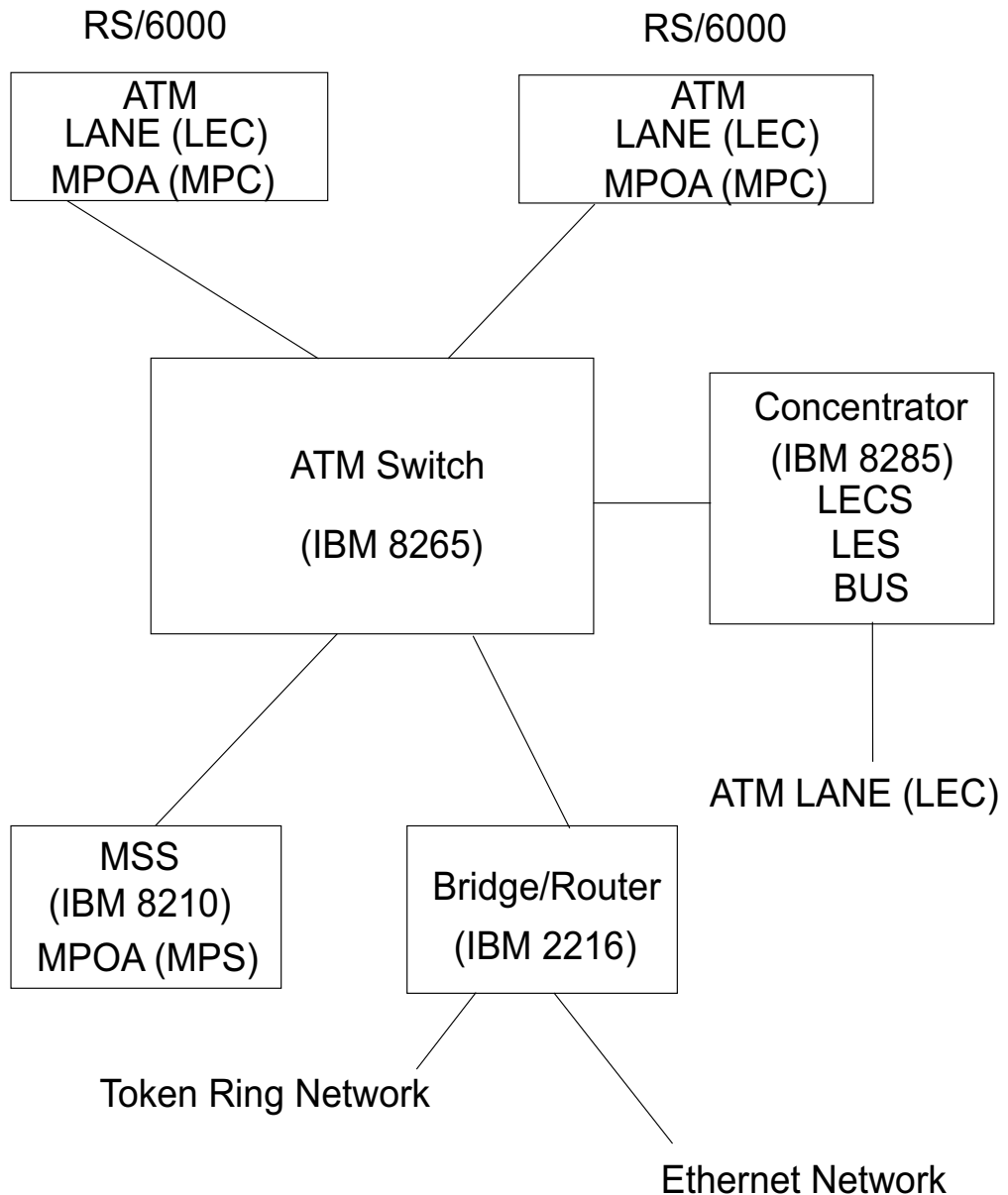


The ATM LANE device driver emulates the operation of Standard Ethernet, IEEE 802.3 Ethernet, and IEEE 802.5 Token Ring LANs. It encapsulates each LAN packet and transfers its LAN data over an ATM network at up to 155 megabits per second. This data can also be bridged transparently to a traditional LAN with ATM/LAN bridges such as the IBM 8281. (See the System Environment illustration.)

Each LEC participates in an emulated LAN containing additional functions such as:

- A LAN Emulation Configuration Server (LECS) that provides automated configuration of the LEC's operational attributes.
- A LAN Emulation Server (LES) that provides address resolution
- A Broadcast and Unknown Server (BUS) that distributes packets sent to a broadcast address or packets sent without knowing the ATM address of the remote station (for example, whenever an ARP response has not been received yet).

There is always at least one ATM switch and a possibility of additional switches, bridges, or concentrators. An example of a typical network topology is shown in the following illustration.



Typical Network Topology

The ATM LANE device driver is a dynamically loadable AIX device driver. Each LE Client or MPOA Client is configurable by the operator, and the LANE driver is loaded into the system as part of that configuration process. If an LE Client or MPOA Client has already been configured, the LANE driver is automatically reloaded at reboot time as part of the system configuration process.

The ATM LANE device driver is a dynamically loadable device driver that operates on a system running AIX Version 4.1.5 (or later). Each LE Client is configurable by the operator, and the LANE driver is loaded into the system as part of that configuration process. If an LE Client has already been configured, the LANE driver is automatically reloaded at reboot time as part of the system configuration process.

The interface to the ATM LANE device driver is through kernel services known as Network Services.

Interfacing to the ATM LANE device driver is achieved by calling the device driver's entry points for opening the device, closing the device, transmitting data, and issuing device control commands, just as you would interface to any of the AIX Common Data Link Interface (CDLI) LAN device drivers.

The ATM LANE device driver interfaces with all hardware-level ATM device drivers that support AIX CDLI, AIX ATM Call Management, and AIX ATM Signaling.

Adding ATM LANE Clients

At least one ATM LAN Emulation client must be added to the system to communicate over an ATM network using the ATM Forum LANE protocol. A user with root authority can add Ethernet or Token-Ring clients using the **smit atmle_panel** fast path.

Entries are required for the Local LE Client's LAN MAC Address field and possibly the LES ATM Address or LECS ATM Address fields, depending on the support provided at the server. If the server accepts the "well-known ATM address" for LECS, the value of the Automatic Configuration via LECS field can be set to **Yes**, and the LES and LECS ATM Address fields can be left blank. If the server does not support the "well-known ATM address" for LECS, an ATM address must be entered for either LES (manual configuration) or LECS (automatic configuration). All other configuration attribute values are optional. If used, you can accept the defaults for ease-of-use.

Configuration help text is also available within the SMIT LE Client add and change menus.

Configuration Parameters for the ATM LANE Device Driver

The ATM LANE device driver supports the following configuration parameters for each LE Client:

addl_drvr	Specifies the CDLI demuxer being used by the LE Client. The value set by the ATM LANE device driver is /usr/lib/methods/cfgdmxtok for Token Ring emulation and /usr/lib/methods/cfgdmxeth for Ethernet. This is not an operator-configurable attribute.
addl_stat	Specifies the routine being used by the LE client to generate device-specific statistics for the entstat and tokstat commands. The values set by the ATM LANE device driver are: <ul style="list-style-type: none">• /usr/sbin/atmle_ent_stat• /usr/sbin/atmle_tok_stat
arp_aging_time	The addl_stat attribute is not operator-configurable. Specifies the maximum timeout period (in seconds) that the LE Client will maintain an LE_ARP cache entry without verification (ATM Forum LE Client parameter C17). The default value is 300 seconds.

arp_cache_size	Specifies the maximum number of LE_ARP cache entries that will be held by the LE Client before removing the least recently used entry. The default value is 32 entries.
arp_response_timeout	Specifies the maximum timeout period (in seconds) for LE_ARP request/response exchanges (ATM Forum LE Client parameter C20). The default value is 1 second.
atm_device	Specifies the logical name of the physical ATM device driver that this LE Client is to operate with, as specified in the CuDv database (for example, atm0 , atm1 , atm2 , ...). The default is atm0 .
auto_cfg	Specifies whether the LE Client is to be automatically configured. Select Yes if the LAN Emulation Configuration Server (LECS) will be used by the LE Client to obtain the ATM address of the LE ARP Server, as well as any additional configuration parameters provided by the LECS. The default value is No (manual configuration). The attribute values are: Yes auto configuration No manual configuration Note: Configuration parameters provided by LECS override configuration values provided by the operator.
control_timeout	Specifies the maximum timeout period (in seconds) for most request/response control frame interactions (ATM Forum LE Client parameter C7). The default value is 120 seconds (2 minutes).

elan_name

Specifies the name of the Emulated LAN this LE Client wishes to join (ATM Forum LE Client parameter C5). This is an SNMPv2 DisplayString of 1-32 characters, or may be left blank (unused). See RFC1213 for a definition of an SNMPv2 DisplayString.

NOTES:

1. Any operator configured **elan_name** should match exactly what is expected at the LECS/LES server when attempting to join an ELAN. Some servers can alias the ELAN name and allow the operator to specify a logical name that correlates to the actual name. Other servers may require the exact name to be specified.

Previous versions of AIX LANE would accept any **elan_name** from the server, even when configured differently by the operator. However, with multiple LECS/LES now possible, it is desirable that only the ELAN identified by the network administrator is joined. Use the **force_elan_name** attribute below to insure that the name you have specified will be the only ELAN joined.

If no **elan_name** attribute is configured at the AIX LEC, or the **force_elan_name** attribute is disabled, the server can stipulate whatever **elan_name** is available.

Failure to use an ELAN name that is identical to the server's when specifying the **elan_name** and **force_elan_name** attributes will cause the LEC to fail the join process, with **entstat/tokstat** status indicating Driver Flag Limbo.

2. Blanks may be inserted within an **elan_name** by typing a tilde (` `) character whenever a blank character is desired. This allows a network administrator to specify an ELAN name with imbedded blanks as in the default of some servers. Any tilde character that occupies the first character position of the **elan_name** is left as is (i.e., the resulting name may start with a tilde but all remaining tilde characters are converted to blanks).

failsafe_time

Specifies the maximum timeout period (in seconds) that the LE Client will attempt to recover from a network outage. A value of zero indicates that attempts to recover should not stop unless a nonrecoverable error is encountered. The default value is 0 (unlimited).

flush_timeout

Specifies the maximum timeout period (in seconds) for FLUSH request/response exchanges (ATM Forum LE Client parameter C21). The default value is 4 seconds.

force_elan_name

Specifies that the Emulated LAN Name returned from the LECS or LES servers must exactly match the name entered in the **elan_name** attribute above. Select Yes if the **elan_name** field must match the server configuration and join parameters. This allows a specific ELAN to be joined when multiple LECS and LES servers are available on the network. The default value is No, which allows the server to specify the ELAN Name.

fwd_delay_time	Specifies the maximum timeout period (in seconds) that the LE Client will maintain an entry for a non-local MAC address in its LE_ARP cache without verification, when the Topology Change flag is true (ATM Forum LE Client parameter C18). The default value is 15 seconds.
lan_type	Identifies the type of local area network being emulated (ATM Forum LE Client parameter C2). Both Ethernet/IEEE 802.3 and Token Ring LANs can be emulated using ATM Forum LANE. The attribute values are: <ul style="list-style-type: none"> • Ethernet/IEEE802.3 • TokenRing
lecs_atm_addr	If you are doing auto configuration using the LE Configuration Server (LECS), this field specifies the ATM address of LECS. It can remain blank if the address of LECS is not known and the LECS is connected via PVC (VPI=0, VCI=17) or the well-known address, or is registered via ILMI. If the 20-byte address of the LECS is known, it must be entered as hexadecimal numbers using a . (period) as the delimiter between bytes. Leading zeros of each byte may be omitted, for example: 47.0.79.0.0.0.0.0.0.0.0.0.0.0.a0.3.0.0.1 (the LECS well-known address)
les_atm_addr	If you are doing manual configuration (without the aid of an LECS), this field specifies the ATM address of the LE ARP Server (LES) (ATM Forum LE Client parameter C9). This 20-byte address must be entered as hexadecimal numbers using a . (period) as the delimiter between bytes. Leading zeros of each byte may be omitted, for example: 39.11.ff.22.99.99.99.0.0.0.0.1.49.10.0.5a.68.0.a.1
local_lan_addr	Specifies the local unicast LAN MAC address that will be represented by this LE Client and registered with the LE Server (ATM Forum LE Client parameter C6). This 6-byte address must be entered as hexadecimal numbers using a . (period) as the delimiter between bytes. Leading zeros of each byte may be omitted. Ethernet Example: 2.60.8C.2C.D2.DC Token Ring Example: 10.0.5A.4F.4B.C4
max_arp_retries	Specifies the maximum number of times an LE_ARP request can be retried (ATM Forum LE Client parameter C13). The default value is 1.
max_config_retries	Specifies the number of times a configuration control frame such as LE_JOIN_REQUEST should be retried, using a duration of control_timeout seconds between retries. The default is 1.
max_frame_size	Specifies the maximum AAL-5 send data-unit size of data frames for this LE Client. In general, this value should coincide with the LAN type and speed as follows: <p>Unspecified for auto LECS configuration</p> <p>1516 bytes for Ethernet and IEEE 802.3 networks</p> <p>4544 bytes for 4 Mbps Token Rings</p> <p>18190 bytes for 16 Mbps Token Rings</p>

max_queued_frames	Specifies the maximum number of outbound packets that will be held for transmission per LE_ARP cache entry. This queueing occurs when the Maximum Unknown Frame Count (max_unknown_fct) has been reached, or when flushing previously transmitted packets while switching to a new virtual channel. The default value is 60 packets.
max_rdy_retries	Specifies the maximum number of READY_QUERY packets sent in response to an incoming call that has not yet received data or a READY_IND packet. The default value is 2 retries.
max_unknown_fct	Specifies the maximum number of frames for a given unicast LAN MAC address that may be sent to the Broadcast and Unknown Server (BUS) within time period Maximum Unknown Frame Time (max_unknown_ftm) (ATM Forum LE Client parameter C10). The default value is 1.
max_unknown_ftm	Specifies the maximum timeout period (in seconds) that a given unicast LAN address may be sent to the Broadcast and Unknown Server (BUS). The LE Client will send no more than Maximum Unknown Frame Count (max_unknown_fct) packets to a given unicast LAN destination within this timeout period (ATM Forum LE Client parameter C11). The default value is 1 second.
mpoa_enabled	Specifies whether Forum MPOA and LANE-2 functions should be enabled for this LE Client. Select Yes if MPOA will be operational on the LE Client. Select No when traditional LANE-1 functionality is required. The default is No (LANE-1).
mpoa_primary	Specifies whether this LE Client is to be the primary configurator for MPOA via LAN Emulation Configuration Server (LECS). Select Yes if this LE Client will be obtaining configuration information from the LECS for the MPOA Client. This attribute is only meaningful if running auto config with an LECS, and indicates that the MPOA configuration TLVs from this LEC will be made available to the MPC. Only one LE Client can be active as the MPOA primary configurator. The default is No.
path_sw_delay	Specifies the maximum timeout period (in seconds) that frames sent on any path in the network will take to be delivered (ATM Forum LE Client parameter C22). The default value is 6 seconds.
peak_rate	Specifies the forward and backward peak bit rate in K-bits per second that will be used by this LE Client to set up virtual channels. It is generally best to specify a value that is compatible with the lowest speed remote device that you expect this LE Client to be communicating with. Higher values may cause congestion in the network. The default value is 155000 K-bits per second, and is adjusted to the actual speed of the adapter for known adapters.
ready_timeout	Specifies the maximum timeout period (in seconds) in which data or a READY_IND message is expected from a calling party (ATM Forum LE Client parameter C28). The default value is 4 seconds.
ring_speed	Specifies the Token Ring speed as viewed by the ifnet layer. The value set by the ATM LANE device driver is 16 Mbps for Token Ring emulation and ignored for Ethernet. This is not an operator-configurable attribute.

soft_restart	Specifies whether active data VC's are to be maintained during connection loss of ELAN services such as the LE ARP Server (LES) or Broadcast and Unknown Server (BUS). Normal ATM Forum operation forces a disconnect of data VC's when LES/BUS connections are lost. This option to maintain active data VC's may be advantageous when server backup capabilities are available. The default value is No.
vcc_activity_timeout	Specifies the maximum timeout period (in seconds) for inactive Data Direct VCCs. Any switched Data Direct VCC that does not transmit or receive data frames in this timeout period is terminated (ATM Forum LE Client parameter C12). The default value is 1200 seconds (20 minutes).

Device Driver Configuration and Unconfiguration

The **atmle_config** entry point performs configuration functions for the ATM LANE device driver.

Device Driver Open

The **atmle_open** function is called to open the specified network device.

The LANE device driver does an asynchronous open. It starts the process of attaching the device to the network, sets the **NDD_UP** flag in the **ndd_flags** field, and returns 0. The network attachment will continue in the background where it is driven by network activity and system timers.

Note: The Network Services **ns_alloc** routine which calls this open routine causes the open to be synchronous. It waits until the **NDD_RUNNING** or the **NDD_LIMBO** flag is set in the **ndd_flags** field or 15 seconds have passed.

If the connection is successful, the **NDD_RUNNING** flag will be set in the **ndd_flags** field, and an **NDD_CONNECTED** status block will be sent. The **ns_alloc** routine will return at this time.

If the device connection fails, the **NDD_LIMBO** flag will be set in the **ndd_flags** field, and an **NDD_LIMBO_ENTRY** status block will be sent.

If the device is eventually connected, the **NDD_LIMBO** flag will be turned off, and the **NDD_RUNNING** flag will be set in the **ndd_flags** field. Both **NDD_CONNECTED** and **NDD_LIMBO_EXIT** status blocks will be sent.

Device Driver Close

The **atmle_close** function is called by the Network Services **ns_free** routine to close the specified network device. This function resets the device to a known state and frees system resources associated with the device.

The device will not be detached from the network until the device's transmit queue is allowed to drain.

Data Transmission

The **atmle_output** function transmits data using the network device.

If the destination address in the packet is a broadcast address, the **M_BCAST** flag in the **p_mbuf->m_flags** field should be set prior to entering this routine. A

broadcast address is defined as FF.FF.FF.FF.FF.FF (hex) for both Ethernet and Token Ring and C0.00.FF.FF.FF.FF (hex) for Token Ring.

If the destination address in the packet is a multicast or group address, the **M_MCAST** flag in the `p_mbuf->m_flags` field should be set prior to entering this routine. A multicast or group address is defined as any nonindividual address other than a broadcast address.

The device driver will keep statistics based on the **M_BCAST** and **M_MCAST** flags.

AIX Token Ring LANE emulates a duplex device. If a Token Ring packet is transmitted with a destination address that matches the LAN MAC address of the local LE Client, the packet is received. This is also true for Token Ring packets transmitted to a broadcast address, enabled functional address, or an enabled group address. AIX Ethernet LANE, on the other hand, emulates a simplex device and does not receive its own broadcast or multicast transmit packets.

Data Reception

When the LANE device driver receives a valid packet from a network ATM device driver, the LANE device driver calls the **nd_receive** function that is specified in the **ndd_t** structure of the network device. The **nd_receive** function is part of a CDLI network demuxer. The packet is passed to the **nd_receive** function in mbufs.

The LANE device driver passes one packet to the **nd_receive** function at a time.

The device driver sets the **M_BCAST** flag in the `p_mbuf->m_flags` field when a packet is received which has an all-stations broadcast destination address. This address value is defined as FF.FF.FF.FF.FF.FF (hex) for both Token Ring and Ethernet and is defined as C0.00.FF.FF.FF.FF (hex) for Token Ring.

The device driver sets the **M_MCAST** flag in the `p_mbuf->m_flags` field when a packet is received which has a nonindividual address that is different than an all-stations broadcast address.

Any packets received from the network are discarded if they do not fit the currently emulated LAN protocol and frame format are discarded.

Asynchronous Status

When a status event occurs on the device, the LANE device driver builds the appropriate status block and calls the **nd_status** function that is specified in the **ndd_t** structure of the network device. The **nd_status** function is part of a CDLI network demuxer.

The following Status Blocks are defined for the LANE device driver:

Hard Failure

When an error occurs within the internal operation of the ATM LANE device driver, it is considered unrecoverable. If the device was operational at the time of the error, the **NDD_LIMBO** and **NDD_RUNNING** flags are turned off, and the **NDD_DEAD** flag is set in the `ndd_flags` field, and a hard failure status block is generated.

code	Set to NDD_HARD_FAIL
option[0]	Set to NDD_UCODE_FAIL

Enter Network Recovery Mode

When the device driver detects an error which requires initiating recovery logic to make the device temporarily unavailable, the following status block is returned by the device driver:

code	Set to NDD_LIMBO_ENTER
option[0]	Set to NDD_UCODE_FAIL

Note: While the device driver is in this recovery logic, the network connections may not be fully functional. The device driver will notify users when the device is fully functional by way of an NDD_LIMBO_EXIT asynchronous status block.

When a general error occurs during operation of the device, this status block is generated.

Exit Network Recovery Mode

When the device driver has successfully completed recovery logic from the error that made the device temporarily unavailable, the following status block is returned by the device driver. This status block means the device is now fully functional.

code	Set to NDD_LIMBO_EXIT
option[0]	The option field is not used.

Device Control Operations

The `atmle_ctl` function is used to provide device control functions.

ATMLE_MIB_GET

This control requests the LANE device driver's current ATM LAN Emulation MIB statistics.

The user should pass in the address of an `atmle_mibs_t` structure as defined in `usr/include/sys/atmle_mibs.h`. The driver will return `EINVAL` if the buffer area is smaller than the required structure.

The `ndd_flags` field can be checked to determine the current state of the LANE device.

ATMLE_MIB_QUERY

This control requests the LANE device driver's ATM LAN Emulation MIB support structure.

The user should pass in the address of an `atmle_mibs_t` structure as defined in `usr/include/sys/atmle_mibs.h`. The driver will return `EINVAL` if the buffer area is smaller than the required structure.

The device driver does *not* support any variables for `read_write` or `write` only. If the syntax of a member of the structure is some integer type, the level of support flag will be stored in the whole field, regardless of the size of the field. For those fields which are defined as character arrays, the value will be returned only in the first byte in the field.

NDD_CLEAR_STATS

This control requests all the statistics counters kept by the LANE device driver to be zeroed.

NDD_DISABLE_ADDRESS

This command disables the receipt of packets destined for a multicast/group address; and for Token Ring, it disables the receipt of packets destined for a functional address. For Token Ring, the functional address indicator (bit 0 "the MSB" of byte 2) indicates whether the address is a functional address (the bit is a 0) or a group address (the bit is a 1).

In all cases, the length field value is required to be 6. Any other value will cause the LANE device driver to return EINVAL.

Functional Address: The reference counts are decremented for those bits in the functional address that are enabled (set to 1). If the reference count for a bit goes to zero, the bit will be "turned off" in the functional address mask for this LE Client.

If no functional addresses are active after receipt of this command, the **TOK_RECEIVE_FUNC** flag in the `ndd_flags` field is reset. If no functional or multicast/group addresses are active after receipt of this command, the **NDD_ALTADDRES** flag in the `ndd_flags` field is reset.

Multicast/Group Address: If a multicast/group address which is currently enabled is specified, receipt of packets destined for that group address is disabled. If an address is specified that is not currently enabled, EINVAL is returned.

If no functional or multicast/group addresses are active after receipt of this command, the **NDD_ALTADDRES** flag in the `ndd_flags` field is reset. Additionally for Token Ring, if no multicast/group address is active after receipt of this command, the **TOK_RECEIVE_GROUP** flag in the `ndd_flags` field is reset.

NDD_DISABLE_MULTICAST

The **NDD_DISABLE_MULTICAST** command disables the receipt of *all* packets with unregistered multicast addresses, and only receives those packets whose multicast addresses were registered using the **NDD_ENABLE_ADDRESS** command. The *arg* and *length* parameters are not used. The **NDD_MULTICAST** flag in the `ndd_flags` field is reset only after the reference count for multicast addresses has reached zero.

NDD_ENABLE_ADDRESS

This command enables the receipt of packets destined for a multicast/group address; and additionally for Token Ring, it enables the receipt of packets destined for a functional address. For Ethernet, the address is entered in canonical format which is left-to-right byte order with the I/G (Individual/Group) indicator as the least significant bit of the first byte. For Token Ring, the address format is entered in noncanonical format which is left-to-right bit and byte order and has a functional address indicator. The functional address indicator (the most significant bit of byte 2) indicates whether the address is a functional address (the bit value is 0) or a group address (the bit value is 1).

In all cases, the length field value is required to be 6. Any other length value will cause the LANE device driver to return EINVAL.

Functional Address: The Token-Ring network architecture provides bit-specific functional addresses for widely used functions, such as Ring Parameter Server or Configuration Report Server. Ring stations use functional address "masks" to

identify these functions. The specified address is "or'ed" with the currently specified functional addresses, and the resultant address is set as the functional address for the device. Functional addresses are encoded in a bit-significant format, thereby allowing multiple individual groups to be designated by a single address.

For example, if function G is assigned a functional address of C0.00.00.08.00.00 (hex), and function M is assigned a functional address of C0.00.00.00.00.40 (hex), then ring station Y, whose node contains function G and M, would have a mask of C0.00.00.08.00.40 (hex). Ring station Y would receive packets addressed to either function G or M or to an address like C0.00.00.08.00.48 (hex) since that address contains bits specified in the "mask."

Note: The LANE device driver forces the first 2 bytes of the functional address to be C0.00 (hex). In addition, bits 6 and 7 of byte 5 of the functional address are forced to 0.

The **NDD_ALTADDRS** and **TOK_RECEIVE_FUNC** flags in the `ndd_flags` field are set.

Since functional addresses are encoded in a bit-significant format, reference counts are kept on each of the 31 least significant bits of the address. Reference counts are not kept on the 17 most significant bits (the C0.00 (hex) of the functional address and the functional address indicator bit).

Multicast/Group Address: A multicast/group address table is used by the LANE device driver to store address filters for incoming multicast/group packets. If the LANE device driver is unable to allocate kernel memory when attempting to add a multicast/group address to the table, the address is not added and ENOMEM is returned.

If the LANE device driver is successful in adding a multicast/group address, the **NDD_ALTADDRS** flag in the `ndd_flags` field is set. Additionally for Token Ring, the **TOK_RECEIVE_GROUP** flag is set, and the first 2 bytes of the group address are forced to be C0.00 (hex).

NDD_ENABLE_MULTICAST

The **NDD_ENABLE_MULTICAST** command enables the receipt of packets with any multicast (or group) address. The *arg* and *length* parameters are not used. The **NDD_MULTICAST** flag in the `ndd_flags` field is set.

NDD_GET_ALL_STATS

This control requests all current LANE statistics, based on both the generic LAN statistics and the ATM LANE protocol in progress.

For Ethernet, you should pass in the address of an `ent_ndd_stats_t` structure as defined in file `/usr/include/sys/cdli_entuser.h`.

For Token Ring, you should pass in the address of a `tok_ndd_stats_t` structure as defined in file `/usr/include/sys/cdli_tokuser.h`.

The driver will return EINVAL if the buffer area is smaller than the required structure.

The `ndd_flags` field can be checked to determine the current state of the LANE device.

NDD_GET_STATS

This control requests the current generic LAN statistics based on the LAN protocol being emulated.

For Ethernet, you should pass in the address of an **ent_ndd_stats_t** structure as defined in file `/usr/include/sys/cdli_entuser.h`.

For Token Ring, you should pass in the address of a **tok_ndd_stats_t** structure as defined in file `/usr/include/sys/cdli_tokuser.h`.

The `ndd_flags` field can be checked to determine the current state of the LANE device.

NDD_MIB_ADDR

This control requests the current receive addresses that are enabled on the LANE device driver. The following address types are returned, up to the amount of memory specified to accept the address list:

- Local LAN MAC Address
- Broadcast Address FF.FF.FF.FF.FF.FF (hex)
- Broadcast Address C0.00.FF.FF.FF.FF (hex)
- (returned for Token Ring only)
- Functional Address Mask
- (returned for Token Ring only, and only if at least one functional address has been enabled)
- Multicast/Group Address 1 through n
- (returned only if at least one multicast/group address has been enabled)

Each address is 6-bytes in length.

NDD_MIB_GET

This control requests the current MIB statistics based on whether the LAN being emulated is Ethernet or Token Ring.

If Ethernet, you should pass in the address of an **ethernet_all_mib_t** structure as defined in file `/usr/include/sys/ethernet_mibs.h`.

If Token Ring, you should pass in the address of a **token_ring_all_mib_t** structure as defined in file `/usr/include/sys/tokenring_mibs.h`.

The driver will return EINVAL if the buffer area is smaller than the required structure.

The `ndd_flags` field can be checked to determine the current state of the LANE device.

NDD_MIB_QUERY

This control requests LANE device driver's MIB support structure based on whether the LAN being emulated is Ethernet or Token Ring.

If Ethernet, you should pass in the address of an **ethernet_all_mib_t** structure as defined in file `/usr/include/sys/ethernet_mibs.h`.

If Token Ring, you should pass in the address of a **token_ring_all_mib_t** structure as defined in file `/usr/include/sys/tokenring_mibs.h`.

The driver will return EINVAL if the buffer area is smaller than the required structure.

The device driver does *not* support any variables for read_write or write only. If the syntax of a member of the structure is some integer type, the level of support flag will be stored in the whole field, regardless of the size of the field. For those fields which are defined as character arrays, the value will be returned only in the first byte in the field.

Tracing and Error Logging in the ATM LANE Device Driver

The LANE device driver has two trace points:

- 3A1 - Normal Code Paths
- 3A2 - Error Conditions

Tracing can be enabled through SMIT or with the **trace** command.

```
trace -a -j 3a1,3a2
```

Tracing can be disabled through SMIT or with the **trcstop** command. Once trace is stopped, the results can be formatted into readable text with the **trcrpt** command.

```
trcrpt > /tmp/trc.out
```

LANE error log templates:

ERRID_ATMLE_MEM_ERR

An error occurred while attempting to allocate memory or pin the code. This error log entry accompanies return code ENOMEM on an open or control operation.

ERRID_ATMLE_LOST_SW

The LANE device driver lost contact with the ATM switch. The device driver will enter Network Recovery Mode in an attempt to recover from the error and will be temporarily unavailable during the recovery procedure. This generally occurs when the cable is unplugged from the switch or ATM adapter.

ERRID_ATMLE_REGAIN_SW

Contact with the ATM switch has been re-established (for example, the cable has been plugged back in).

ERRID_ATMLE_NET_FAIL

The device driver has gone into Network Recovery Mode in an attempt to recover from a network error and is temporarily unavailable during the recovery procedure. User intervention is not required for this error unless the problem persists.

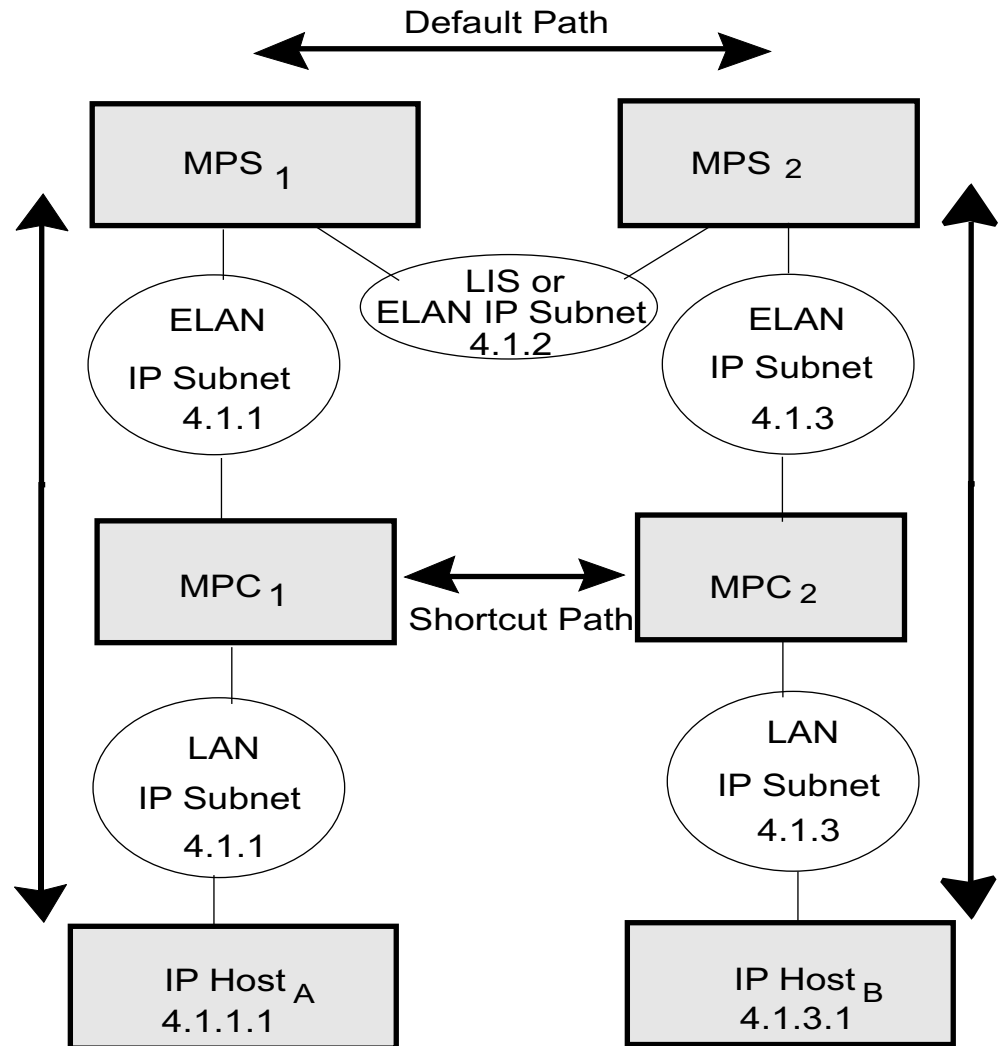
ERRID_ATMLE_RCVRY_COMPLETE

The network error which caused the LANE device driver to go into error recovery mode has been corrected.

Adding an ATM MPOA Client

An MPOA (Multi-Protocol Over ATM) Client (MPC) can be added to the system to allow ATM LANE packets that would normally be routed through various LANE IP Subnets or Logical IP Subnets (LIS's) within an ATM network, to be sent and received over shortcut paths that do not contain routers. MPOA can provide significant savings on end-to-end throughput performance for large data transfers, and can free up resources in routers that might otherwise be used up handling

packets that could have bypassed routers altogether. See the following mpoa environment figure.



Only one MPOA Client is established per node. This MPC can support multiple ATM ports, containing LE Clients/Servers and MPOA Servers. The key requirement being, that for this MPC to create shortcut paths, each remote target node must also support MPOA Client, and must be directly accessible via the matrix of switches representing the ATM network.

A user with root authority can add this MPOA Client using the `smit mpoa_panel` fast path, or by navigating through Devices - Communication - ATM Adapter - Services - Multi-Protocol Over ATM (MPOA).

No configuration entries are required for the MPOA Client. Ease-of-use default values are provided for each of the attributes which are derived from ATM Forum recommendations.

Configuration help text is also available within MPOA Client SMIT to aid in making any modifications to attribute default values.

Configuration Parameters for ATM MPOA Client

The ATM LANE device driver supports the following configuration parameters for the MPOA Client:

<i>auto_cfg</i>	Auto Configuration with LEC/LECS. Specifies whether the MPOA Client is to be automatically configured via LANE Configuration Server (LECS). Select Yes if a primary LE Client will be used to obtain the MPOA configuration attributes, which will override any manual or default values. The default value is No (manual configuration). The attribute values are: Yes - auto configuration No - manual configuration
<i>sc_setup_count</i>	Shortcut Setup Frame Count. This attribute is used in conjunction with <i>sc_setup_time</i> to determine when to establish a shortcut path. Once the MPC has forwarded at least <i>sc_setup_count</i> packets to the same target within a period of <i>sc_setup_time</i> , the MPC attempts to create a shortcut VCC. This attribute correlates to ATM Forum MPC Configuration parameter <i>MPC-p1</i> . The default value is 10 packets.
<i>sc_setup_time</i>	Shortcut Setup Frame Time (in seconds). This attribute is used in conjunction with <i>sc_setup_count</i> above to determine when to establish a shortcut path. Once the MPC has forwarded at least <i>sc_setup_count</i> packets to the same target within a period of <i>sc_setup_time</i> , the MPC attempts to create a shortcut VCC. This attribute correlates to ATM Forum MPC Configuration parameter <i>MPC-p2</i> . The default value is 1 second.
<i>init_retry_time</i>	Initial Request Retry Time (in seconds). Specifies the length of time to wait before sending the first retry of a request that does not receive a response. This attribute correlates to ATM Forum MPC Configuration parameter <i>MPC-p4</i> . The default value is 5 seconds.
<i>retry_time_max</i>	Maximum Request Retry Time (in seconds). Specifies the maximum length of time to wait when retrying requests that have not received a response. Each retry duration after the initial retry are doubled (2x) until the retry duration reaches this Maximum Request Retry Time. All subsequent retries will wait this maximum value. This attribute correlates to ATM Forum MPC Configuration parameter <i>MPC-p5</i> . The default value is 40 seconds.
<i>hold_down_time</i>	Failed resolution request retry Hold Down Time (in seconds). Specifies the length of time to wait before reinitiating a failed address resolution attempt. This value is normally set to a value greater than <i>retry_time_max</i> . This attribute correlates to ATM Forum MPC Configuration parameter <i>MPC-p6</i> . The default value is 160 seconds.
<i>vcc_inact_time</i>	VCC Inactivity Timeout value (in minutes). Specifies the maximum length of time to keep a shortcut VCC enabled when there is no send or receive activity on that VCC. The default value is 20 minutes.

Tracing and Error Logging in the ATM MPOA Client

The ATM MPOA Client has two trace points:

- 3A3 - Normal Code Paths
- 3A4 - Error Conditions

Tracing can be enabled through SMIT or with the "trace" command.


```
trace -a -j 3a3,3a4
```

racing can be disabled through SMIT or with the `trcstop` command. Once trace is stopped, the results can be formatted into readable text with the `trcrpt` command.

```
trcrpt > /tmp/trc.out
```

MPOA Client error log templates:

Each of the MPOA Client error log templates are prefixed with `ERRID_MPOA`. An example of an MPOA error entry is as follows:

ERRID_MPOA_MEM_ERR

An error occurred while attempting to allocate kernel memory.

Fiber Distributed Data Interface (FDDI) Device Driver

The FDDI device driver is a dynamically loadable device driver that runs on systems using AIX Version 4.1 (or later). The device driver is automatically loaded into the system at device configuration time as part of the configuration process.

The interface to the device is through the kernel services known as Network Services.

Interfacing to the device driver is achieved by calling the device driver's entry points for opening the device, closing the device, transmitting data, doing a remote dump, and issuing device control commands.

The FDDI device driver supports the SMT 7.2 standard.

Configuration Parameters for FDDI Device Driver

Software Transmit Queue	The driver provides a software transmit queue to supplement the hardware queue. The queue is configurable and contains between 3 and 250 mbufs. The default is 30 mbufs.
Alternate Address	The driver supports specifying a configurable alternate address to be used instead of the address burned in on the card. This address must have the local bit set. Addresses between 0x400000000000 and 0x7FFFFFFFFFFFFF are supported. The default is 0x400000000000.
Enable Alternate Address	The driver supports enabling the alternate address set with the Alternate Address parameter. Values are YES and NO, with NO as the default.
PMF Password	The driver provides the ability to configure a PMF password. The password default is 0, meaning no password.
Max T-Req	The driver enables the user to configure the card's maximum T-Req.
TVX Lower Bound	The driver enables the user to configure the card's TVX Lower Bound.
User Data	The driver enables the user to set the user data field on the adapter. This data can be any string up to 32 bytes of data. The default is a zero length string.

FDDI Device Driver Configuration and Unconfiguration

The `fddi_config` entry point performs configuration functions for the FDDI device driver.

Device Driver Open

The `fddi_open` function is called to open the specified network device.

The device is initialized. When the resources have been successfully allocated, the device is attached to the network.

If the station is not connected to another running station, the device driver opens, but is unable to transmit Logical Link Control (LLC) packets. When in this mode, the device driver sets the `CFDDI_NDD_LLC_DOWN` flag (defined in `/usr/include/sys/cdli_fddiuser.h`). When the adapter is able to make a connection with at least one other station this flag is cleared and LLC packets can be transmitted.

Device Driver Close

The `fddi_close` function is called to close the specified network device. This function resets the device to a known state and frees system resources used by the device.

The device is not detached from the network until the device's transmit queue is allowed to drain.

Data Transmission

The `fddi_output` function transmits data using the network device.

The FDDI device driver supports up to three mbuf's for each packet. It cannot gather from more than three locations to a packet.

The FDDI device driver does *not* accept user-memory mbufs. It uses `bcopy` on small frames which does not work on user memory.

The driver supports up to the entire mtu in a single mbuf.

The driver requires that the entire mac header be in a single mbuf.

The driver will not accept chained frames of different types. The user should not send Logical Link Control (LLC) and station management (SMT) frames in the same call to output.

The user needs to fill the frame out completely before calling the output routine. The mac header for a FDDI packet is defined by the `cfddi_hdr_t` structure defined in `/usr/include/sys/cdli_fddiuser.h`. The first byte of a packet is used as a flag for routing the packet on the adapter. For most driver users the value of the packet should be set to `FDDI_TX_NORM`. The possible flags are:

<code>CFDDI_TX_NORM</code>	Transmits the frame onto the ring. This is the normal flag value.
<code>CFDDI_TX_LOOPBACK</code>	Moves the frame from the adapter's transmit queue to its receive queue as if it were received from the media. The frame is not transmitted onto the media.

CFDDI_TX_PROC_ONLY	Processes the status information frame (SIF) or parameter management frame (PMF) request frame and sends a SIF or PMF response to the host. The frame is not transmitted onto the media. This flag is <i>not</i> valid for LLC packets.
CFDDI_TX_PROC_XMIT	Processes the SIF or PMF request frames and sends a SIF or PMF response to the host. The frame is also transmitted onto the media. This flag is <i>not</i> valid for LLC packets.

Data Reception

When the FDDI device driver receives a valid packet from the network device, the FDDI device driver calls the **nd_receive** function that is specified in the **ndd_t** structure of the network device. The **nd_receive** function is part of a CDLI network demuxer. The packet is passed to the **nd_receive** function in mbufs.

Reliability, Availability, and Serviceability for FDDI Device Driver

The FDDI device driver has three trace points. The IDs are defined in the **/usr/include/sys/cdli_fddiuser.h** file.

For FDDI the type of data in an error log is the same for every error log. Only the specifics and the title of the error log change. Information that follows includes an example of an error log and a list of error log entries.

Example FDDI Error Log

```

Detail Data
FILE NAME
line: 332 file: fddiintr_b.c
POS REGISTERS
F48E D317 3CC7 0008
SOURCE ADDRESS
4000 0000 0000
ATTACHMENT CLASS
0000 0001
MICRO CHANNEL AND PIO EXCEPTION CODES
0000 0000 0000 0000 0000 0000
FDDI LINK STATISTICS
0080 0000 04A0 0000 0000 0000 0001 0000 0000 0000
0001 0008 0008 0005 0005 0012 0003 0002 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
SELF TESTS
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000
DEVICE DRIVER INTERNAL STATE
0fdd 0fdd 0000 0000 0000 0000 0000 0000

```

Error Log Entries

The FDDI device driver returns the following are the error log entries:

ERRID_CFDDI_RMV_ADAP	<p>This error indicates that the adapter has received a disconnect command from a remote station. The FDDI device driver will initiate shutdown of the device. The device is no longer functional due to this error. User intervention is required to bring the device back online.</p> <p>If there is no local LAN administrator, user action is required to make the device available.</p> <p>For the device to be brought back online, the device needs to be reset. This can be accomplished by having all users of the FDDI device driver close the device.</p> <p>When all users have closed the device and the device is reset, the device can be brought back online.</p>
ERRID_CFDDI_ADAP_CHECK	<p>This error indicates that an FDDI adapter check has occurred. If the device was connected to the network when this error occurred, the FDDI device goes into Network Recovery Mode in an attempt to recover from the error. The device is temporarily unavailable during the recovery procedure. User intervention is not required to bring the device back online.</p>
ERRID_CFDDI_DWNLD	<p>Indicates that the microcode download to the FDDI adapter has failed. If this error occurs during the configuration of the device, the configuration of the device fails. User intervention is required to make the device available.</p>
ERRID_CFDDI_RCVRY_ENTER	<p>Indicates that the FDDI device driver has entered Network Recovery Mode in an attempt to recover from an error. The error which caused the device to enter this mode, is error logged before this error log entry. The device is not fully functional until the device has left this mode. User intervention is not required to bring the device back online.</p>
ERRID_CFDDI_RCVRY_EXIT	<p>Indicates that the FDDI device driver has successfully recovered from the error which caused the device to go into Network Recovery Mode. The device is now fully functional.</p>
ERRID_CFDDI_RCVRY_TERM	<p>Indicates that the FDDI device driver was unable to recover from the error which caused the device to go into Network Recovery Mode and has terminated recovery logic. The termination of recovery logic may be due to an irrecoverable error being detected or the device being closed. If termination is due to an irrecoverable error, that error will be error logged before this error log entry. User intervention is required to bring the device back online.</p>
ERRID_CFDDI_MC_ERR	<p>Indicates that the FDDI device driver has detected a Micro Channel error. The device driver initiates recovery logic in an attempt to recover from the error. User intervention is not required for this error unless the problem persists.</p>
ERRID_CFDDI_TX_ERR	<p>Indicates that the FDDI device driver has detected a transmission error. User intervention is not required unless the problem persists.</p>

ERRID_CFDDI_PIO	Indicates the FDDI device driver has detected a program IO error. The device driver initiates recovery logic in an attempt to recover from the error. User intervention is not required for this error unless the problem persists.
ERRID_CFDDI_DOWN	Indicates that the FDDI device has been shutdown due to an irrecoverable error. The FDDI device is no longer functional due to the error. The irrecoverable error which caused the device to be shutdown is error logged before this error log entry. User intervention is required to bring the device back online.
ERRID_CFDDI_SELF_TEST	Indicates that the FDDI adapter has received a run self-test command from a remote station. The device is unavailable while the adapter's self-tests are being run. If the tests are successful, the FDDI device driver initiates logic to reconnect the device to the network. Otherwise, the device will be shutdown.
ERRID_CFDDI_SELF_ERR	Indicates that an error occurred during the FDDI self-tests. User intervention is required to bring the device back online.
ERRID_CFDDI_PATH_ERR	Indicates that an error occurred during the FDDI adapter's path tests. The FDDI device driver will initiate recovery logic in an attempt to recover from the error. The FDDI device will temporarily be unavailable during the recovery procedure. User intervention is not required to bring the device back online.
ERRID_CFDDI_PORT	Indicates that a port on the FDDI device is in a stuck condition. User intervention is not required for this error. This error typically occurs when a cable is not correctly connected.
ERRID_CFDDI_BYPASS	Indicates that the optical bypass switch is in a stuck condition. User intervention is not required for this error.
ERRID_CFDDI_CMD_FAIL	Indicates that a command to the adapter has failed.

High-Performance (8fc8) Token-Ring Device Driver

The 8fc8 Token-Ring device driver is a dynamically loadable device driver that will run on a system running AIX Version 4.1 (or later). The device driver will be automatically loaded into the system at device configuration time as part of the configuration process.

The interface to the device is through the kernel services known as Network Services.

Interfacing to the device driver is achieved by calling the device driver's entry points for opening the device, closing the device, transmitting data, doing a remote dump, and issuing device control commands.

The Token-Ring device driver interfaces with the Token-Ring High-Performance Network Adapter (8fc8). It provides a Micro Channel-based connection to a Token-Ring network. The adapter is IEEE 802.5 compatible and supports both 4 and 16 megabit per second networks. The adapter supports only a Shielded Twisted-Pair (STP) Token-Ring connection.

Configuration Parameters for Token-Ring Device Driver

Ring Speed	The device driver will support a user configurable parameter which indicates if the Token-Ring is to be run at 4 or 16 megabits per second.
Software Transmit Queue	The device driver will support a user configurable transmit queue, that can be set to store between 32 and 160 transmit request pointers. Each transmit request pointer corresponds to a transmit request which may be for several buffers of data.
Attention MAC frames	The device driver will support a user configurable parameter that indicates if attention MAC frames should be received.
Beacon MAC frames	The device driver will support a user configurable parameter that indicates if beacon MAC frames should be received.
Network Address	The driver supports the use of the device's hardware address as the network address or an alternate network address configured through software. When an alternate address is used, any valid individual address can be used. The most significant bit of the address must be set to zero (definition of an individual address).

Device Driver Configuration and Unconfiguration

The `tok_config` entry point performs configuration functions Token-Ring device driver.

Device Driver Open

The `tok_open` function is called to open the specified network device.

The Token Ring device driver does an asynchronous open. It starts the process of attaching the device to the network, sets the `NDD_UP` flag in the `ndd_flags` field, and returns 0. The network attachment will continue in the background where it is driven by device activity and system timers.

Note: The Network Services `ns_alloc` routine which calls this open routine causes the open to be synchronous. It waits until the `NDD_RUNNING` flag is set in the `ndd_flags` field or 60 seconds have passed.

If the connection is successful, the `NDD_RUNNING` flag will be set in the `ndd_flags` field and a `NDD_CONNECTED` status block will be sent. The `ns_alloc` routine will return at this time.

If the device connection fails, the `NDD_LIMBO` flag will be set in the `ndd_flags` field and a `NDD_LIMBO_ENTRY` status block will be sent.

If the device is eventually connected, the `NDD_LIMBO` flag will be turned off and the `NDD_RUNNING` flag will be set in the `ndd_flags` field. Both `NDD_CONNECTED` and `NDD_LIMBO_EXIT` status blocks will be set.

Device Driver Close

The `tok_close` function is called to close the specified network device. This function resets the device to a known state and frees system resources associated with the device.

The device will not be detached from the network until the device's transmit queue is allowed to drain.

Data Transmission

The `tok_output` function transmits data using the network device.

The device driver does *not* support mbufs from user memory (which have the `M_EXT` flag set).

If the destination address in the packet is a broadcast address, the `M_BCAST` flag in the `p_mbuf->m_flags` field should be set prior to entering this routine. A broadcast address is defined as `0xFFFF FFFF FFFF` or `0xC000 FFFF FFFF`. If the destination address in the packet is a multicast address the `M_MCAST` flag in the `p_mbuf->m_flags` field should be set prior to entering this routine. A multicast address is defined as a non-individual address other than a broadcast address. The device driver will keep statistics based upon the `M_BCAST` and `M_MCAST` flags.

If a packet is transmitted with a destination address which matches the adapter's address, the packet will be received. This is true for the adapter's physical address, broadcast addresses (`0xC000 FFFF FFFF` or `0xFFFF FFFF FFFF`), enabled functional addresses, or an enabled group address.

Data Reception

When the Token-Ring device driver receives a valid packet from the network device, the Token-Ring device driver calls the `nd_receive` function that is specified in the `ndd_t` structure of the network device. The `nd_receive` function is part of a CDLI network demuxer. The packet is passed to the `nd_receive` function in mbufs.

The Token-Ring device driver passes one packet to the `nd_receive` function at a time.

The device driver sets the `M_BCAST` flag in the `p_mbuf->m_flags` field when a packet is received which has an all-stations broadcast address. This address is defined as `0xFFFF FFFF FFFF` or `0xC000 FFFF FFFF`.

The device driver sets the `M_MCAST` flag in the `p_mbuf->m_flags` field when a packet is received which has a non-individual address that is different than the all-stations broadcast address.

The adapter does not pass invalid packets to the device driver.

Asynchronous Status

When a status event occurs on the device, the Token-Ring device driver builds the appropriate status block and calls the `nd_status` function that is specified in the `ndd_t` structure of the network device. The `nd_status` function is part of a CDLI network demuxer.

The following Status Blocks are defined for the Token-Ring device driver.

Hard Failure

When a hard failure has occurred on the Token-Ring device, the following status blocks can be returned by the Token-Ring device driver. One of these status blocks indicates that a fatal error occurred.

NDD_PIO_FAIL: When a PIO error occurs, it is retried 3 times. If the error still occurs, it is considered unrecoverable and this status block is generated.

code Set to NDD_HARD_FAIL
option[0] Set to NDD_PIO_FAIL
option[] The remainder of the status block may be used to return additional status information.

TOK_RECOVERY_THRESH: When most network errors occur, they are retried. Some errors are retried with no limit and others have a recovery threshold. Errors that have a recovery threshold and fail all the retries specified by the recovery threshold are considered unrecoverable and generate the following status block:

code Set to NDD_HARD_FAIL
option[0] Set to TOK_RECOVERY_THRESH
option[1] The specific error which occurred. Possible values are:

- TOK_DUP_ADDR - duplicate node address
- TOK_PERM_HW_ERR - the device has an unrecoverable hardware error
- TOK_RING_SPEED - ring beaconing on physical insertion to the ring
- TOK_RMV_ADAP - remove ring station MAC frame received

Enter Network Recovery Mode

When the device driver has detected an error which requires initiating recovery logic that will make the device temporarily unavailable, the following status block is returned by the device driver:

Note: While the device driver is in this recovery logic, the device may not be fully functional. The device driver will notify users when the device is fully functional by way of an NDD_LIMBO_EXIT asynchronous status block.

NDD_ADAP_CHECK: When an adapter check has occurred, this status block is generated.

code Set to NDD_LIMBO_ENTER
option[0] Set to NDD_ADAP_CHECK
option[1] The adapter check interrupt information is stored in the 2 high-order bytes. The adapter also returns three two-byte parameters. Parameter 0 is stored in the 2 low-order bytes.
option[2] Parameter 1 is stored in the 2 high-order bytes. Parameter 2 is stored in the 2 low-order bytes.

NDD_AUTO_RMV: When an internal hardware error following the beacon automatic removal process has been detected, this status block is generated.

code Set to NDD_LIMBO_ENTER
option[0] Set to NDD_AUTO_RMV

NDD_BUS_ERR: The device has detected a I/O channel error.

code Set to NDD_LIMBO_ENTER
option[0] Set to NDD_BUS_ERR
option[1] Set to error information from the device.

NDD_CMD_FAIL: The device has detected an error in a command the device driver issued to it.

code Set to NDD_LIMBO_ENTER
option[0] Set to NDD_CMD_FAIL
option[1] Set to error information from the device.

NDD_TX_ERROR: The device has detected an error in a packet given to the device.

code Set to NDD_LIMBO_ENTER
option[0] Set to NDD_TX_ERROR
option[1] Set to error information from the device.

NDD_TX_TIMEOUT: The device has detected an error in a packet given to the device.

code Set to NDD_LIMBO_ENTER
option[0] Set to NDD_TX_TIMEOUT

TOK_ADAP_INIT: When the initialization of the device fails, this status block is generated.

code Set to NDD_LIMBO_ENTER
option[0] Set to TOK_ADAP_INIT
option[1] Set to error information from the device.

TOK_ADAP_OPEN: When a general error occurs during open of the device, this status block is generated.

code Set to NDD_LIMBO_ENTER
option[0] Set to TOK_ADAP_OPEN
option[1] Set to the device open error code from the device.

TOK_DMA_FAIL: A d_complete has failed.

code Set to NDD_LIMBO_ENTER
option[0] Set to TOK_DMA_FAIL

TOK_RING_SPEED: When an error code of 0x27 (physical insertion, ring beaconing) occurs during open of the device, this status block is generated.

code Set to NDD_LIMBO_ENTER
option[0] Set to TOK_RING_SPEED

TOK_RMV_ADAP: The device has received a remove ring station MAC frame indicating that a network management function had directed this device to get off the ring.

code Set to NDD_LIMBO_ENTER
option[0] Set to TOK_RMV_ADAP

TOK_WIRE_FAULT: When an error code of 0x11 (lobe media test, function failure) occurs during open of the device, this status block is generated.

code Set to NDD_LIMBO_ENTER
option[0] Set to TOK_WIRE_FAULT

Exit Network Recovery Mode

When the device driver has successfully completed recovery logic from the error that made the device temporarily unavailable, the following status block is returned by the device driver. This status block means the device is now fully functional.

code Set to NDD_LIMBO_EXIT
option[] The option fields are not used.

Network Device Driver Status

When the device driver has status or event information to report, the following status block is returned by the device driver:

Ring Beaconsing: When the Token-Ring device has detected a beaconsing condition (or the ring has recovered from one), the following status block is generated by the Token-Ring device driver:

code Set to NDD_STATUS
option[0] Set to TOK_BEACONSING
option[1] Set to the ring status received from the device.

Device Connected

When the device is successfully connected to the network the following status block is returned by the device driver:

code Set to NDD_CONNECTED
option[] The option fields are not used.

Device Control Operations

The `tok_ctl` function is used to provide device control functions.

NDD_GET_STATS

The user should pass in the `tok_ddd_stats_t` structure as defined in `usr/include/sys/cdli_tokuser.h`. The driver will fail a call with a buffer smaller than the structure.

The statistics which are returned contain statistics obtained from the device. If the device is inoperable, the statistics which are returned will not contain the current device statistics. The copy of the `ddd_flags` field can be checked to determine the state of the device.

NDD_MIB_QUERY

The `arg` parameter specifies the address of the `token_ring_all_mib_t` structure. This structure is defined in the `/usr/include/sys/tokenring_mibs.h` file.

The device driver does *not* support any variables for `read_write` or `write only`. If the syntax of a member of the structure is some integer type, the level of support

flag will be stored in the whole field, regardless of the size of the field. For those fields which are defined as character arrays, the value will be returned only in the first byte in the field.

NDD_MIB_GET

The *arg* parameter specifies the address of the `token_ring_all_mib_t` structure. This structure is defined in the `/usr/include/sys/tokenring_mibs.h` file.

If the device is inoperable, the `upstream` field of the `Dot5Entry_t` structure will be zero instead of containing the nearest active upstream neighbor (NAUN). Also the statistics which are returned contain statistics obtained from the device. If the device is inoperable, the statistics which are returned will not contain the current device statistics. The copy of the `ndd_flags` field can be checked to determine the state of the device.

NDD_ENABLE_ADDRESS

This command enables the receipt of packets with a functional or a group address. The functional address indicator (bit 0 "the MSB" of byte 2) indicates whether the address is a functional address (the bit is a 0) or a group address (the bit is a 1). The length field is not used because the address must be 6 bytes in length.

Functional Address: The specified address is "or'ed" with the currently specified functional addresses and the resultant address is set as the functional address for the device. Functional addresses are encoded in a bit-significant format, thereby allowing multiple individual groups to be designated by a single address.

The Token-Ring network architecture provides bit-specific functional addresses for widely-used functions, such as configuration report server. Ring stations use functional address "masks" to identify these functions. For example, if function G is assigned a functional address of `0xC000 0008 0000`, and function M is assigned a function address of `0xC000 0000 0040`, then ring station Y, whose node contains function G and M, would have a mask of `0xC000 0008 0040`. Ring station Y would receive packets addressed to either function G or M or to an address like `0xC000 0008 0048` since that address contains bits specified in the "mask".

Note: The device forces the first 2 bytes of the functional address to be `0xC000`. In addition, bits 6 and 7 of byte 5 of the functional address are forced to a 0 by the device.

The `NDD_ALTADDRS` and `TOK_RECEIVE_FUNC` flags in the `ndd_flags` field are set.

Since functional addresses are encoded in a bit-significant format, reference counts are kept on each of the 31 least significant bits of the address. Reference counts are not kept on the 17 most significant bits (the `0xC000` of the functional address and the functional address indicator bit).

Group Address: If no group address is currently enabled, the specified address is set as the group address for the device. The group address will not be set and `EINVAL` will be returned if a group address is currently enabled.

The device forces the first 2 bytes of the group address to be `0xC000`.

The `NDD_ALTADDRS` and `TOK_RECEIVE_GROUP` flags in the `ndd_flags` field are set.

NDD_DISABLE_ADDRESS

This command disables the receipt of packets with a functional or a group address. The functional address indicator (bit 0 "the MSB" of byte 2) indicates whether the address is a functional address (the bit is a 0) or a group address (the bit is a 1). The length field is not used because the address must be 6 bytes in length.

Functional Address: The reference counts are decremented for those bits in the functional address that are a one (on). If the reference count for a bit goes to zero, the bit will be "turned off" in the functional address for the device.

If no functional addresses are active after receipt of this command, the TOK_RECEIVE_FUNC flag in the `ndd_flags` field is reset. If no functional or group addresses are active after receipt of this command, the NDD_ALTADDRS flag in the `ndd_flags` field is reset.

Group Address: If the group address which is currently enabled is specified, receipt of packets with a group address is disabled. If a different address is specified, EINVAL will be returned.

If no group address is active after receipt of this command, the TOK_RECEIVE_GROUP flag in the `ndd_flags` field is reset. If no functional or group addresses are active after receipt of this command, the NDD_ALTADDRS flag in the `ndd_flags` field is reset.

NDD_MIB_ADDR

The following addresses are returned:

- Device Physical Address (or alternate address specified by user)
- Broadcast Address 0xFFFF FFFF FFFF
- Broadcast Address 0xC000 FFFF FFFF
- Functional Address (only if a user specified a functional address)
- Group Address (only if a user specified a group address)

NDD_CLEAR_STATS

The counters kept by the device will be zeroed.

NDD_GET_ALL_STATS

The *arg* parameter specifies the address of the `mon_all_stats_t` structure. This structure is defined in the `/usr/include/sys/cdli_tokuser.h` file.

The statistics which are returned contain statistics obtained from the device. If the device is inoperable, the statistics which are returned will not contain the current device statistics. The copy of the `ndd_flags` field can be checked to determine the state of the device.

Trace Points and Error Log Templates for 8fc8 Token-Ring Device Driver

The Token-Ring device driver has three trace points. The IDs are defined in the `usr/include/sys/cdli_tokuser.h` file.

The Token-Ring error log templates are:

ERRID_CTOK_ADAP_CHECK	The microcode on the device performs a series of diagnostic checks when the device is idle. These checks can find errors and they are reported as adapter checks. If the device was connected to the network when this error occurred, the device driver will go into Network Recovery Mode in an attempt to recover from the error. The device is temporarily unavailable during the recovery procedure. User intervention is not required for this error unless the problem persists.
ERRID_CTOK_ADAP_OPEN	The device driver was unable to open the device. The device driver will go into Network Recovery Mode in an attempt to recover from the error. The device is temporarily unavailable during the recovery procedure. User intervention is not required for this error unless the problem persists.
ERRID_CTOK_AUTO_RMV	An internal hardware error following the beacon automatic removal process has been detected. The device driver will go into Network Recovery Mode in an attempt to recover from the error. The device is temporarily unavailable during the recovery procedure. User intervention is not required for this error unless the problem persists.
ERRID_CONFIG	The ring speed (or ring data rate) is probably wrong. Contact the network administrator to determine the speed of the ring. The device driver will only retry twice at 2 minute intervals after this error log entry has been generated.
ERRID_CTOK_DEVICE_ERR	The device detected an I/O channel error or an error in a command the device driver issued, an error occurred during a PIO operation, or the device has detected an error in a packet given to the device. The device driver will go into Network Recovery Mode in an attempt to recover from the error. The device is temporarily unavailable during the recovery procedure. User intervention is not required for this error unless the problem persists.
ERRID_CTOK_DOWNLOAD	The download of the microcode to the device failed. User intervention is required to make the device available.
ERRID_CTOK_DUP_ADDR	The device has detected that another station on the ring has an device address which is the same as the device address being tested. Contact network administrator to determine why.
ERRID_CTOK_MEM_ERR	An error occurred while allocating memory or timer control block structures.
ERRID_CTOK_PERM_HW	The device driver could not reset the card. For example, did not receive status from the adapter within the retry period.
ERRID_CTOK_RCVRY_EXIT	The error which caused the device driver to go into error recovery mode has been corrected.
ERRID_CTOK_RMV_ADAP	The device has received a remove ring station MAC frame indicating that a network management function has directed this device to get off the ring. Contact network administrator to determine why.
ERRID_CTOK_WIRE_FAULT	There is probably a loose (or bad) cable between the device and the MAU. There is some chance that it might be a bad device. The device driver will go into Network Recovery Mode in an attempt to recover from the error. The device is temporarily unavailable during the recovery procedure. User intervention is required for this error.

High-Performance (8fa2) Token-Ring Device Driver

The 8fa2 Token-Ring device driver is a dynamically loadable device driver that will run on a system running AIX Version 4.1 (or later). The device driver is automatically loaded into the system at device configuration time as part of the configuration process.

The interface to the device is through the kernel services known as Network Services.

Interfacing to the device driver is achieved by calling the device driver's entry points for opening the device, closing the device, transmitting data, doing a remote dump, and issuing device control commands.

The Token-Ring device driver interfaces with the Token-Ring High-Performance Network Adapter (8fa2). It provides a Micro Channel-based connection to a Token-Ring network. The adapter is IEEE 802.5 compatible and supports both 4 and 16 megabit per second networks. The adapter supports only a RJ-45 connection.

Configuration Parameters for 8fa2 Token-Ring Device Driver

The following lists the configuration parameters necessary to use the device driver.

- Ring Speed** Indicates the Token-Ring speed. The speed is set at 4 or 16 megabits per second or autosense.
- 4** Specifies that the device driver will open the adapter with 4 Mbits. It will return an error if ring speed does not match the network speed.
- 16** Specifies that the device driver will open the adapter with 16 Mbits. It will return an error if ring speed does not match the network speed.

autosense

Specifies that the adapter will open with the speed used determined as follows:

- If it is an open on an existing network, the speed will be the ring speed of the network.
- If it is an open on a new network:
 - If the adapter is a new adapter, 16 Mbits is used.
 - If the adapter had successfully opened, the ring speed will be the ring speed of the last successful open.

Software Transmit Queue

Specifies a transmit request pointer that can be set to store between 32 and 2048 transmit request pointers. Each transmit request pointer corresponds to a transmit request which may be for several buffers of data.

Attention MAC frames

Indicates if attention MAC frames should be received.

Beacon MAC frames

Indicates if beacon MAC frames should be received.

Priority Data Transmission

Specifies a request priority transmission of the data packets.

Network Address

Specifies the use of the device's hardware address as the network address or an alternate network address configured through software. When an alternate address is used, any valid Individual Address can be used. The most significant bit of the address must be set to zero (definition of an Individual Address).

Device Driver Configuration and Unconfiguration

The `tok_config` entry point performs configuration functions Token-Ring device driver.

Device Driver Open

The `tok_open` function is called to open the specified network device.

The Token Ring device driver does a synchronous open. The device will be initialized at this time. When the resources have been successfully allocated, the device will start the process of attaching the device to the network.

If the connection is successful, the `NDD_RUNNING` flag will be set in the `ndd_flags` field and a `NDD_CONNECTED` status block will be sent.

If the device connection fails, the `NDD_LIMBO` flag will be set in the `ndd_flags` field and a `NDD_LIMBO_ENTRY` status block will be sent.

If the device is eventually connected, the `NDD_LIMBO` flag will be turned off and the `NDD_RUNNING` flag will be set in the `ndd_flags` field. Both `NDD_CONNECTED` and `NDD_LIMBO_EXIT` status blocks will be set.

Device Driver Close

The `tok_close` function is called to close the specified network device. This function resets the device to a known state and frees system resources associated with the device.

The device will not be detached from the network until the device's transmit queue is allowed to drain.

Data Transmission

The `tok_output` function transmits data using the network device.

The device driver does *not* support mbufs from user memory (which have the `M_EXT` flag set).

If the destination address in the packet is a broadcast address the `M_BCAST` flag in the `p_mbuf->m_flags` field should be set prior to entering this routine. A broadcast address is defined as `0xFFFF FFFF FFFF` or `0xC000 FFFF FFFF`. If the destination address in the packet is a multicast address the `M_MCAST` flag in the `p_mbuf->m_flags` field should be set prior to entering this routine. A multicast address is defined as a non-individual address other than a broadcast address. The device driver will keep statistics based upon the `M_BCAST` and `M_MCAST` flags.

If a packet is transmitted with a destination address which matches the adapter's address, the packet will be received. This is true for the adapter's physical address, broadcast addresses (`0xC000 FFFF FFFF` or `0xFFFF FFFF FFFF`), enabled functional addresses, or an enabled group address.

Data Reception

When the Token-Ring device driver receives a valid packet from the network device, the Token-Ring device driver calls the `nd_receive` function that is specified in the `ndd_t` structure of the network device. The `nd_receive` function is part of a CDLI network demuxer. The packet is passed to the `nd_receive` function in mbufs.

The Token-Ring device driver will only pass one packet to the `nd_receive` function at a time.

The device driver will set the `M_BCAST` flag in the `p_mbuf->m_flags` field when a packet is received which has an all stations broadcast address. This address is defined as `0xFFFF FFFF FFFF` or `0xC000 FFFF FFFF`.

The device driver will set the `M_MCAST` flag in the `p_mbuf->m_flags` field when a packet is received which has a non-individual address which is different than the all-stations broadcast address.

The adapter will not pass invalid packets to the device driver.

Asynchronous Status

When a status event occurs on the device, the Token-Ring device driver builds the appropriate status block and calls the `nd_status` function that is specified in the `ndd_t` structure of the network device. The `nd_status` function is part of a CDLI network demuxer.

The following Status Blocks are defined for the Token-Ring device driver.

Hard Failure

When a hard failure has occurred on the Token-Ring device, the following status blocks can be returned by the Token-Ring device driver. One of these status blocks indicates that a fatal error occurred.

NDD_PIO_FAIL:

Indicates that when a PIO error occurs, it is retried 3 times. If the error persists, it is considered unrecoverable and the following status block is generated:

code Set to `NDD_HARD_FAIL`

option[0]
Set to `NDD_PIO_FAIL`

option[]
The remainder of the status block is used to return additional status information.

NDD_HARD_FAIL:

Indicates that when a transmit error occurs it is retried. If the error is unrecoverable, the following status block is generated:

code Set to `NDD_HARD_FAIL`

option[0]
Set to `NDD_HARD_FAIL`

option[]
The remainder of the status block is used to return additional status information.

NDD_ADAP_CHECK:

Indicates that when an adapter check has occurred, the following status block is generated:

code Set to `NDD_ADAP_CHECK`

option[]
The remainder of the status block is used to return additional status information.

NDD_DUP_ADDR:

Indicates that the device detected a duplicated address in the network and the following status block is generated:

code Set to NDD_DUP_ADDR

option[]

The remainder of the status block is used to return additional status information.

NDD_CMD_FAIL:

Indicates that the device detected an error in a command that the device driver issued. The following status block is generated:

code Set to NDD_CMD_FAIL

option[0]

Set to the command code

option[]

Set to error information from the command.

TOK_RING_SPEED:

Indicates that when a ring speed error occurs while the device is being open, the following status block is generated:

code Set to NDD_LIMBO_ENTER

option[]

Set to error information.

Enter Network Recovery Mode

Indicates that when the device driver has detected an error which requires initiating recovery logic that will make the device temporarily unavailable, the following status block is returned by the device driver.

Note: While the device driver is in this recovery logic, the device may not be fully functional. The device driver will notify users when the device is fully functional by way of an NDD_LIMBO_EXIT asynchronous status block.

code Set to NDD_LIMBO_ENTER

option[0]

Set to one of the following:

- NDD_CMD_FAIL
- TOK_WIRE_FAULT
- NDD_BUS_ERROR
- NDD_ADAP_CHECK
- NDD_TX_TIMEOUT
- TOK_BEACONING

option[]

The remainder of the status block is used to return additional status information by the device driver.

Exit Network Recovery Mode

Indicates that when the device driver has successfully completed recovery logic from the error that made the device temporarily unavailable, the following status block is returned by the device driver. This status block indicates the device is now fully functional.

code Set to NDD_LIMBO_EXIT

option[]

N/A

Device Connected

Indicates that when the device is successfully connected to the network the following status block is returned by the device driver:

code Set to NDD_CONNECTED

option[]

N/A

Device Control Operations

The `tok_ctl` function is used to provide device control functions.

NDD_GET_STATS

The user should pass in the `tok_ndd_stats_t` structure as defined in `<sys/cdli_tokuser.h>`. The driver will fail a call with a buffer smaller than the structure.

The structure must be in a kernel heap so that the device driver can copy the statistics into it; and it must be pinned.

NDD_PROMISCUOUS_ON

Setting promiscuous mode will *not* cause non-LLC frames to be received by the driver unless the user also enables those filters (Attention MAC frames, Beacon MAC frames).

The driver will maintain a counter of requests.

NDD_PROMISCUOUS_OFF

This command will release a request from a user to PROMISCUOUS_ON; it will not exit the mode on the adapter if more requests are outstanding.

NDD_MIB_QUERY

The `arg` parameter specifies the address of the `token_ring_all_mib_t` structure. This structure is defined in the `/usr/include/sys/tokenring_mibs.h` file.

The device driver does *not* support any variables for read_write or write only. If the syntax of a member of the structure is some integer type, the level of support flag will be stored in the whole field, regardless of the size of the field. For those fields which are defined as character arrays, the value will be returned only in the first byte in the field.

NDD_MIB_GET

The *arg* parameter specifies the address of the `token_ring_all_mib_t` structure. This structure is defined in the `/usr/include/sys/tokenring_mibs.h` file.

NDD_ENABLE_ADDRESS

This command enables the receipt of packets with a functional or a group address. The functional address indicator (bit 0 "the MSB" of byte 2) indicates whether the address is a functional address (the bit is a 0) or a group address (the bit is a 1). The length field is not used because the address must be 6 bytes in length.

Functional Address:

The specified address is ORed with the currently specified functional addresses and the resultant address is set as the functional address for the device. Functional addresses are encoded in a bit-significant format, thereby allowing multiple individual groups to be designated by a single address.

The Token-Ring network architecture provides bit-specific functional addresses for widely used functions, such as configuration report server. Ring stations use functional address "masks" to identify these functions. For example, if function G is assigned a functional address of 0xC000 0008 0000, and function M is assigned a function address of 0xC000 0000 0040, then ring station Y, whose node contains function G and M, would have a mask of 0xC000 0008 0040. Ring station Y would receive packets addressed to either function G or M or to an address like 0xC000 0008 0048 since that address contains bits specified in the "mask".

The `NDD_ALTADDRS` and `TOK_RECEIVE_FUNC` flags in the `ndd_flags` field are set.

Since functional addresses are encoded in a bit-significant format, reference counts are kept on each of the 31 least significant bits of the address.

Group Address:

The device support 256 general group addresses. The promiscuous mode will be turned on when the group address needed to be set are more than 256. The device driver will maintain a reference count on this operation.

The `NDD_ALTADDRS` and `TOK_RECEIVE_GROUP` flags in the `ndd_flags` field are set.

NDD_DISABLE_ADDRESS

This command disables the receipt of packets with a functional or a group address. The functional address indicator (bit 0 "the MSB" of byte 2) indicates whether the address is a functional address (the bit is a 0) or a group address (the bit is a 1). The length field is not used because the address must be 6 bytes in length.

Functional Address:

The reference counts are decremented for those bits in the functional address that are one (meaning on). If the reference count for a bit goes to zero, the bit will be "turned off" in the functional address for the device.

If no functional addresses are active after receipt of this command, the TOK_RECEIVE_FUNC flag in the **ndd_flags** field is reset. If no functional or group addresses are active after receipt of this command, the NDD_ALTADDRS flag in the **ndd_flags** field is reset.

Group Address:

If the number of group address enabled is less than 256, the driver sends a command to the device to disable the receipt of the packets with the specified group address. Otherwise, the driver just deletes the group address from the group address table.

If there are less than 256 group addresses enabled after the receipt of this command, the promiscuous mode is turned off.

If no group address is active after receipt of this command, the TOK_RECEIVE_GROUP flag in the **ndd_flags** field is reset. If no functional or group addresses are active after receipt of this command, the NDD_ALTADDRS flag in the **ndd_flags** field is reset.

NDD_PRIORITY_ADDRESS

The driver returns the address of the device driver's priority transmit routine.

NDD_MIB_ADDR

The driver will return at least three addresses: device physical address (or alternate address specified by user) and two broadcast addresses (0xFFFF FFFF FFFF and 0xC000 FFFF FFFF). Additional addresses specified by the user, such as functional address and group addresses, may also be returned.

NDD_CLEAR_STATS

The counters kept by the device are zeroed.

NDD_GET_ALL_STATS

The *arg* parameter specifies the address of the **mon_all_stats_t** structure. This structure is defined in the **/usr/include/sys/cdli_tokuser.h** file.

The statistics returned include statistics obtained from the device. If the device is inoperable, the statistics returned do not contain the current device statistics. The copy of the **ndd_flags** field can be checked to determine the state of the device.

Trace Points and Error Log Templates for 8fa2 Token-Ring Device Driver

The Token-Ring device driver has four trace points. The IDs are defined in the **/usr/include/sys/cdli_tokuser.h** file.

The Token-Ring error log templates are :

ERRID_MPS_ADAP_CHECK

The microcode on the device performs a series of diagnostic checks when the device is idle. These checks can find errors and they are reported as adapter checks. If the device was connected to the network when this error occurred, the device driver goes into Network Recovery Mode to try to

recover from the error. The device is temporarily unavailable during the recovery procedure. User intervention is not required unless the problem persists.

ERRID_MPS_ADAP_OPEN

The device driver was unable to open the device. The device driver goes into Network Recovery Mode to try to recover from the error. The device is temporarily unavailable during the recovery procedure. User intervention is not required unless the problem persists.

ERRID_MPS_AUTO_RMV

An internal hardware error following the beacon automatic removal process has been detected. The device driver goes into Network Recovery Mode to try to recover from the error. The device is temporarily unavailable during the recovery procedure. User intervention is not required unless the problem persists.

ERRID_MPS_RING_SPEED

The ring speed (or ring data rate) is probably wrong. Contact the network administrator to determine the speed of the ring. The device driver only retries twice at 2 minute intervals when this error log entry is generated.

ERRID_MPS_DMAFAIL

The device detected an DMA error in a TX or RX operation. The device driver goes into Network Recovery Mode to try to recover from the error. The device is temporarily unavailable during the recovery procedure. User intervention is not required unless the problem persists.

ERRID_MPS_BUS_ERR

The device detected a Micro Channel bus error. The device driver goes into Network Recovery Mode to try to recover from the error. The device is temporarily unavailable during the recovery procedure. User intervention is not required unless the problem persists.

ERRID_MPS_DUP_ADDR

The device has detected that another station on the ring has an device address which is the same as the device address being tested. Contact the network administrator to determine why.

ERRID_MPS_MEM_ERR

An error occurred while allocating memory or timer control block structures.

ERRID_MPS_PERM_HW

The device driver could not reset the card. For example, it did not receive status from the adapter within the retry period.

ERRID_MPS_RCVRY_EXIT

The error which caused the device driver to go into error recovery mode has been corrected.

ERRID_MPS_RMV_ADAP

The device has received a remove ring station MAC frame indicating that a network management function has directed this device to get off the ring. Contact the network administrator to determine why.

ERRID_MPS_WIRE_FAULT

There is probably a loose (or bad) cable between the device and the MAU. There is some chance that it might be a bad device. The device driver goes

into Network Recovery Mode to try to recover from the error. The device is temporarily unavailable during the recovery procedure. User intervention is required for this error.

ERRID_MPS_RX_ERR

The device detected a receive error. The device driver goes into Network Recovery Mode to try to recover from the error. The device is temporarily unavailable during the recovery procedure. User intervention is not required unless the problem persists.

ERRID_MPS_TX_TIMEOUT

The transmit watchdog timer expired before transmitting a frame is complete. The device driver goes into Network Recovery Mode to try to recover from the error. The device is temporarily unavailable during the recovery procedure. User intervention is not required unless the problem persists.

ERRID_MPS_CTL_ERR

The IOCTL watchdog timer expired before the device driver received a response from the device. The device driver goes into Network Recovery Mode to try to recover from the error. The device is temporarily unavailable during the recovery procedure. User intervention is not required unless the problem persists.

PCI Token-Ring High Performance (14101800) Device Driver

The Token-Ring device driver is a dynamically loadable device driver that runs on AIX Version 4.1 (or later). The device driver is automatically loaded into the system at device configuration time as part of the configuration process. The interface to the device is through the kernel services known as Network Services.

Interfacing to the device driver is achieved by calling the device driver's entry points for opening the device, closing the device, transmitting data, doing a remote dump, and issuing device control commands.

The Token-Ring device driver interfaces with the PCI Token-Ring High-Performance Network Adapter (14101800). The adapter is IEEE 802.5 compatible and supports both 4 and 16 megabit per second networks. The adapter supports only an RJ-45 connection.

Configuration Parameters

Ring Speed

The device driver supports a user-configurable parameter that indicates if the token-ring is to run at 4 or 16 megabits per second.

The device driver supports a user-configurable parameter that selects the ring speed of the adapter. There are three options for the ring speed: 4, 16, or autosense.

1. If 4 is selected, the device driver opens the adapter with 4 Mbits. It returns an error if the ring speed does not match the network speed.
2. If 16 is selected, the device driver opens the adapter with 16 Mbits. It returns an error if the ring speed does not match the network speed.
3. If autosense is selected, the adapter guarantees a successful open, and the speed used to open is dependent on:
 - If it is opened on an existing network, in which case the speed is the ring speed of the network.
 - If it is opened on a new network, in which case 16 Mbits is used if the adapter is new; or if the adapter successfully opened, the ring speed is the speed of the last successful open.

Receive Queue

The device driver supports a user-configurable receive queue that can be set to store between 32 and 160 receive buffers. These buffers are **mbuf** clusters into which the device writes the received data.

Software Transmit Queue

The device driver supports a user-configurable transmit queue that can be set to store between 32 and 2048 transmit request pointers. Each transmit request pointer corresponds to a transmit request that may be for several buffers of data.

Software Priority Transmit Queue

The device driver supports a user-configurable priority transmit queue that can be set to store between 32 and 160 transmit request pointers. Each transmit request pointer corresponds to a transmit request that may be for several buffers of data.

Full Duplex

Indicates whether the adapter is operating in full-duplex or half-duplex mode. If this field is set yes, the device driver programs the adapter to be in full-duplex mode. The default value is half-duplex.

Attention MAC Frames

The device driver supports a user-configurable parameter that indicates if attention MAC frames should be received.

Beacon MAC Frames

The device driver supports a user-configurable parameter that indicates if beacon MAC frames should be received.

Priority Data Transmission

The device driver supports a user option to request priority transmission of the data packets.

Network Address

The driver supports the use of the device's hardware address as the network address or an alternate network address configured through software. When an alternate address is used, any valid Individual Address can be used. The most significant bit of the address must be set to zero.

Device Driver Configuration and Unconfiguration

The `tok_config()` entry point conforms to the AIX Version 4.1 (or later) kernel object file entry point.

Device Driver Open

The `tok_open()` function is called to open the specified network device.

The Token-Ring device driver does a synchronous open. The device is initialized at this time. When the resources are successfully allocated, the device starts the process of attaching the device to the network.

If the connection is successful, the `NDD_RUNNING` flag is set in the `ndd_flags` field, and an `NDD_CONNECTED` status block is sent.

If the device connection fails, the `NDD_LIMBO` flag is set in the `ndd_flags` field, and an `NDD_LIMBO_ENTRY` status block is sent.

If the device is eventually connected, the `NDD_LIMBO` flag is turned off, and the `NDD_RUNNING` flag is set in the `ndd_flags` field. Both `NDD_CONNECTED` and `NDD_LIMBO_EXIT` status blocks are set.

Device Driver Close

The `tok_close()` function is called to close the specified network device. This function resets the device to a known state and frees system resources associated with the device.

The device is not detached from the network until the device's transmit queue is allowed to drain.

Data Transmission

The `tok_output()` function transmits data using the network device.

The device driver does *not* support mbufs from user memory that have the `M_EXT` flag set.

If the destination address in the packet is a broadcast address, the `M_BCAST` flag in the `p_mbuf->m_flags` field should be set prior to entering this routine. A broadcast address is defined as `0xFFFF FFFF FFFF` or `0xC000 FFFF FFFF`. If the destination address in the packet is a multicast address, the `M_MCAST` flag in the `p_mbuf->m_flags` field should be set prior to entering this routine. A multicast address is defined as a non-individual address other than a broadcast address. The device driver keeps statistics based on the `M_BCAST` and `M_MCAST` flags.

If a packet is transmitted with a destination address that matches the adapter's address, the packet is received. This is true for the adapter's physical address, broadcast addresses (`0xC000 FFFF FFFF` or `0xFFFF FFFF FFFF`), enabled functional addresses, or an enabled group address.

Data Reception

When the Token-Ring device driver receives a valid packet from the network device, the Token-Ring device driver calls the `nd_receive()` function specified in the `ndd_t` structure of the network device. The `nd_receive()` function is part of a CDLI network demuxer. The packet is passed to the `nd_receive()` function in mbufs.

The Token-Ring device driver passes only one packet to the `nd_receive()` function at a time.

The device driver sets the `M_BCAST` flag in the `p_mbuf->m_flags` field when a packet is received that has an all-stations broadcast address. This address is defined as `0xFFFF FFFF FFFF` or `0xC000 FFFF FFFF`.

The device driver sets the `M_MCAST` flag in the `p_mbuf->m_flags` field when a packet is received that has a non-individual address that is different from the all-stations broadcast address.

The adapter does not pass invalid packets to the device driver.

Asynchronous Status

When a status event occurs on the device, the Token-Ring device driver builds the appropriate status block and calls the `nd_status()` function specified in the `ndd_t` structure of the network device. The `nd_status()` function is part of a CDLI network demuxer.

The following status blocks are defined for the Token-Ring device driver.

Hard Failure

When a hard failure occurs on the Token-Ring device, the following status blocks are returned by the Token-Ring device driver. One of these status blocks indicates that a fatal error occurred.

NDD_HARD_FAIL When a transmit error occurs, it tries to recover. If the error is unrecoverable, this status block is generated.

code Set to `NDD_HARD_FAIL`.

option[0]

Set to `NDD_HARD_FAIL`.

option[]

The remainder of the status block can be used to return additional status information.

Enter Network Recovery Mode

When the device driver detects an error that requires initiating recovery logic to make the device temporarily unavailable, the following status block is returned by the device driver.

Note: While the device driver is in this recovery logic, the device may not be fully functional. The device driver notifies users when the device is fully functional by way of an `NDD_LIMBO_EXIT` asynchronous status block:

code Set to `NDD_LIMBO_ENTER`.

option[0] Set to one of the following:

- NDD_CMD_FAIL
- NDD_ADAP_CHECK
- NDD_TX_ERR
- NDD_TX_TIMEOUT
- NDD_AUTO_RMV
- TOK_ADAP_OPEN
- TOK_ADAP_INIT
- TOK_DMA_FAIL
- TOK_RING_SPEED
- TOK_RMV_ADAP
- TOK_WIRE_FAULT

option[] The remainder of the status block can be used to return additional status information by the device driver.

Exit Network Recovery Mode

When the device driver has successfully completed recovery logic from the error that made the device temporarily unavailable, the following status block is returned by the device driver:

Note: The device is now fully functional.

code Set to NDD_LIMBO_EXIT.
option[] The option fields are not used.

Device Control Operations

The `tok_ctl()` function is used to provide device control functions.

NDD_GET_STATS

The user should pass in the `tok_nda_stats_t` structure as defined in `<sys/cdli_tokuser.h>`. The driver fails a call with a buffer smaller than the structure.

The structure must be in kernel heap so that the device driver can copy the statistics into it. Also, it must be pinned.

NDD_PROMISCUOUS_ON

Setting promiscuous mode will *not* cause non-LLC frames to be received by the driver unless the user also enables those filters (Attention MAC frames, Beacon MAC frames).

The driver maintains a counter of requests.

NDD_PROMISCUOUS_OFF

This command releases a request from a user to PROMISCUOUS_ON; it will not exit the mode on the adapter if more requests are outstanding.

NDD_MIB_QUERY

The `arg` parameter specifies the address of the `token_ring_all_mib_t` structure. This structure is defined in the `/usr/include/sys/tokenring_mibs.h` file.

The device driver does *not* support any variables for read_write or write only. If the syntax of a member of the structure is some integer type, the level of support

flag is stored in the whole field, regardless of the size of the field. For those fields that are defined as character arrays, the value is returned only in the first byte in the field.

NDD_MIB_GET

The **arg** parameter specifies the address of the **token_ring_all_mib_t** structure. This structure is defined in the **/usr/include/sys/tokenring_mibs.h** file.

NDD_ENABLE_ADDRESS

This command enables the receipt of packets with a functional or a group address. The functional address indicator (bit 0 "the MSB" of byte 2) indicates whether the address is a functional address (bit 0) or a group address (bit 1). The length field is not used because the address must be 6 bytes in length.

functional address

The specified address is ORed with the currently specified functional addresses, and the resultant address is set as the functional address for the device. Functional addresses are encoded in a bit-significant format, thereby allowing multiple individual groups to be designated by a single address.

The Token-Ring network architecture provides bit-specific functional addresses for widely used functions, such as configuration report server. Ring stations use functional address "masks" to identify these functions. For example, if function G is assigned a functional address of 0xC000 0008 0000, and function M is assigned a function address of 0xC000 0000 0040, then ring station Y, whose node contains function G and M, would have a mask of 0xC000 0008 0040. Ring station Y would receive packets addressed to either function G or M or to an address like 0xC000 0008 0048 since that address contains bits specified in the "mask."

The **NDD_ALTADDRS** and **TOK_RECEIVE_FUNC** flags in the **ndd_flags** field are set.

Since functional addresses are encoded in a bit-significant format, reference counts are kept on each of the 31 least significant bits of the address.

group address

The device supports 256 general group addresses. The promiscuous mode is turned on when the group address to be set is more than 256. The device driver maintains a reference count on this operation.

The device supports 256 general group addresses. The promiscuous mode is turned on when the group address needed to be set are more than 256. The device driver will maintain a reference count on this operation.

The **NDD_ALTADDRS** and **TOK_RECEIVE_GROUP** flags in the **ndd_flags** field are set.

NDD_DISABLE_ADDRESS

This command disables the receipt of packets with a functional or a group address. The functional address indicator (bit 0 "the MSB" of byte 2) indicates whether the address is a functional address (bit 0) or a group address (bit 1). The length field is not used because the address must be 6 bytes in length.

functional address

The reference counts are decremented for those bits in the functional address that are 1 (on). If the reference count for a bit goes to 0, the bit is "turned off" in the functional address for the device.

If no functional addresses are active after receipt of this command, the **TOK_RECEIVE_FUNC** flag in the `ndd_flags` field is reset. If no functional or group addresses are active after receipt of this command, the **NDD_ALTADDRS** flag in the `ndd_flags` field is reset.

group address

If group address enable is less than 256, the driver sends a command to the device to disable the receipt of the packets with the specified group address. Otherwise, the group address is deleted from the group address table.

If there are less than 256 group addresses enabled after the receipt of this command, the promiscuous mode is turned off.

If no group address is active after receipt of this command, the **TOK_RECEIVE_GROUP** flag in the `ndd_flags` field is reset. If no functional or group addresses are active after receipt of this command, the **NDD_ALTADDRS** flag in the `ndd_flags` field is reset.

NDD_PRIORITY_ADDRESS

The driver returns the address of the device driver's priority transmit routine.

NDD_MIB_ADDR

The driver returns at least three addresses that are device physical addresses (or alternate addresses specified by the user), two broadcast addresses (0xFFFFFFFF and 0xC000 FFFF FFFF), and any additional addresses specified by the user, such as functional addresses and group addresses.

NDD_CLEAR_STATS

The counters kept by the device are zeroed.

NDD_GET_ALL_STATS

The `arg` parameter specifies the address of the `mon_all_stats_t` structure. This structure is defined in the `/usr/include/sys/cdli_tokuser.h` file.

The statistics that are returned contain information obtained from the device. If the device is inoperable, the statistics returned are not the current device statistics. The copy of the `ndd_flags` field can be checked to determine the state of the device.

Reliability, Availability, and Serviceability (RAS)**Trace**

The Token-Ring device driver has four trace points. The IDs are defined in the `/sys/cdli_tokuser.h` file.

Error Logging

The Token-Ring error log templates are:

ERRID_STOK_ADAP_CHECK	The microcode on the device performs a series of diagnostic checks when the device is idle. These checks can find errors, and they are reported as adapter checks. If the device is connected to the network when this error occurs, the device driver goes into Network Recovery Mode in an attempt to recover from the error. The device is temporarily unavailable during the recovery procedure. User intervention is not required for this error unless the problem persists.
ERRID_STOK_ADAP_OPEN	Enables the device driver to open the device. The device driver goes into Network Recovery Mode in an attempt to recover from the error. The device is temporarily unavailable during the recovery procedure. User intervention is not required for this error unless the problem persists.
ERRID_STOK_AUTO_RMV	An internal hardware error following the beacon automatic removal process was detected. The device driver goes into Network Recovery Mode in an attempt to recover from the error. The device is temporarily unavailable during the recovery procedure. User intervention is not required for this error unless the problem persists.
ERRID_STOK_RING_SPEED	The ring speed (or ring data rate) is probably wrong. Contact the network administrator to determine the speed of the ring. The device driver only retries twice at 2-minute intervals after this error log entry is generated.
ERRID_STOK_DMAFAIL	The device detected a DMA error in a TX or RX operation. The device driver goes into Network Recovery Mode in an attempt to recover from the error. The device is temporarily unavailable during the recovery procedure. User intervention is not required unless the problem persists.
ERRID_STOK_BUS_ERR	The device detected a Micro Channel bus error. The device driver goes into Network Recovery Mode in an attempt to recover from the error. The device is temporarily unavailable during the recovery procedure. User intervention is not required for this error unless the problem persists.
ERRID_STOK_DUP_ADDR	The device detected that another station on the ring has a device address that is the same as the device address being tested. Contact the network administrator to determine why.
ERRID_STOK_MEM_ERR	An error occurred while allocating memory or timer control block structures.
ERRID_STOK_RCVRY_EXIT	The error that caused the device driver to go into error recovery mode was corrected.

ERRID_STOK_RMV_ADAP	The device received a remove ring station MAC frame indicating that a network management function directed this device to get off the ring. Contact the network administrator to determine why.
ERRID_STOK_WIRE_FAULT	There is a loose (or bad) cable between the device and the MAU. There is a chance that it might be a bad device. The device driver goes into Network Recover Mode in an attempt to recover from the error. The device is temporarily unavailable during the recovery procedure. User intervention is not required for this error unless the problem persists.
ERRID_STOK_TX_TIMEOUT	The transmit watchdog timer expired before transmitting a frame. The device driver goes into Network Recovery Mode in an attempt to recover from the error. The device is temporarily unavailable during the recovery procedure. User intervention is not required for this error unless the problem persists.
ERRID_STOK_CTL_ERR	The ioctl watchdog timer expired before the device driver received a response from the device. The device driver goes into Network Recovery Mode in an attempt to recover from the error. The device is temporarily unavailable during the recovery procedure. User intervention is not required for this error unless the problem persists.

Ethernet Device Drivers

The following AIX Version 4 Ethernet device drivers are dynamically loadable device drivers that run on systems running AIX Version 4. The device drivers are automatically loaded into the system at device configuration time as part of the configuration process.

- Ethernet High-Performance LAN Adapter Device Driver
- Integrated Ethernet Device Driver
- 10/100 Mbps Ethernet TX MCA Device Driver
- PCI Ethernet Device Driver
- Gigabit Ethernet-SX PCI Adapter Device Driver

Note: The 10/100 MBps Ethernet TX MCA device driver is available on AIX Version 4.1.5 (and later) systems.

The following information is provided about each of the ethernet device drivers:

- “Configuration Parameters” on page 159
- “Interface Entry Points” on page 164
- “Asynchronous Status” on page 166
- “Device Control Operations” on page 168
- “Reliability, Availability, and Serviceability (RAS)” on page 171

For each Ethernet device, the interface to the device driver is achieved by calling the entry points for opening, closing, transmitting data, and issuing device control

commands. The Integrated Ethernet, 10/100 Mbps Ethernet TX MCA (AIX Version 4.1.5 and later), and PCI Ethernet Device Drivers also provide an interface for doing remote system dumps.

There are a number of Ethernet device drivers in use. The IBM ISA 16-bit Ethernet Adapter is the only existing ISA driver. The Ethernet High-Performance LAN Adapters (8ef5 and 8f95) and the Integrated Ethernet Device Drivers (8ef2, 8ef3, 8f98) all provide microchannel-based connections to an Ethernet network. The 10/100 Mbps Ethernet TX MCA Device Driver (8f62) provides a microchannel-based connection using a PCI adapter and bridge chip. The PCI Ethernet Device Driver (22100020) and the PCI 10/100 Mbps Ethernet Device Driver (23100020) provide PCI-based connections to an Ethernet network. All drivers support both Standard and IEEE 802.3 Ethernet Protocols, with support for a transmission rate of 10 megabits per second. The 10/100 Mbps Ethernet TX MCA Device Driver and PCI 10/100 Mbps Ethernet device driver (23100020) also support a transmission rate of 100 megabits per second. The Gigabit Ethernet-SX PCI Adapter (14100401) will not support either the transmission rate of 10 or 100 megabits per second.

The Ethernet High-Performance LAN Adapter (8ef5) device driver interfaces with a 3COM microchannel adapter card installed in one of the microchannel slots located on the system. This adapter supports thick (10BASE5 or DIX) and thin (10BASE2 or BNC) Ethernet connections.

The 10 Mbps Ethernet Low-Cost High-Performance Adapter (8f95) device driver interfaces with a microchannel adapter installed in one of the microchannel slots located on the system. This adapter supports AUI, 10BASE2 and 10BASE-T Ethernet connections.

The Integrated Ethernet Device Drivers (8ef2, 8ef3, 8f98) interface with an Intel 82596 Ethernet coprocessor located on the CPU planar, and is hardwired to microchannel slot 14 on the desktop systems. These devices support thick, thin, or twisted-pair (10BASE-T) Ethernet connections.

The 10/100 Mbps Ethernet TX MCA Adapter (8f62) interfaces with an Am79C971 Ethernet chip through an Adaptec AIC960 bridge chip. This device supports MII (Media Independent Interface).

The PCI Ethernet Device Driver (22100020) interfaces with an Am79C970 Ethernet chip located either on the planar or in an adapter card installed in one of the PCI slots on the system. This device supports twisted-pair (10BASE-T) and thin Ethernet connections. On the planar, only the twisted-pair connection is available for this PCI Ethernet device.

The PCI 10/100 Mbps Ethernet Device Driver (23100020) interfaces with an Am79C971 Ethernet chip located either on the planar or in an adapter card installed in one of the PCI slots on the system. This driver supports MII (Media Independent Interface).

The Gigabit Ethernet-SX PCI Adapter (14100401) device driver interfaces with a custom Application Specific Integrated Circuit (ASIC) in an adapter card installed in one of the PCI slots on the system. This device supports an SX fiber connection.

Configuration Parameters

The following is the configuration parameter that is supported by all Ethernet device drivers:

Alternate Ethernet Addresses

The device drivers support the device's hardware address as the network address or an alternate network address configured through software. When an alternate address is used, any valid Individual Address can be used. The least significant bit of an Individual Address must be set to zero. A multicast address can not be defined as a network address. Two configuration parameters are provided to provide the alternate Ethernet address and enable the alternate address.

The following are configuration parameters that are supported by the Ethernet High-Performance LAN Adapter (8ef5 and 8f95) and the Integrated Ethernet Device Drivers (8ef2, 8ef3, 8f98):

Software Transmit Queue

The device drivers support a user-configurable transmit queue that can be set to store between 20 to 150 transmit request pointers. Each transmit request pointer corresponds to a transmit request which may be for several buffers of data.

Adapter Connector Type

The device drivers support a user-configurable adapter connection for both BNC and DIX (AUI for adapter (8f95)) physical connector types. The Ethernet High-Performance LAN Adapter (8f95) device driver also supports user-configurable adapter connections TP (twisted-pair) and AUTO (auto sense).

Note: This option is not supported on some systems that implement the Integrated Ethernet and have DIX as the default.

The Ethernet High-Performance LAN Adapter (8ef5) device driver supports the following additional configuration parameter:

Receive Buffer Pool Size

The Ethernet High-Performance LAN Adapter (8ef5) device driver supports a user-configurable receive buffer pool. With this attribute, the user can configure between 16 to 64 receive buffers that will be used during the reception of incoming packets from the network. Increasing from a default value of 37 results in a smaller transmit buffer pool. Decreasing from the default value increases the number of transmit buffers in the pool.

The Ethernet High-Performance LAN Adapter (8f95) device driver supports the following additional configuration parameters:

Transmit Interrupt Mode

The Ethernet High-Performance LAN Adapter (8f95) can be configured to operate in one of three transmit modes.

Delay (0)

Sends notification of transmit completion based on the number of packets transmitted.

Immediate (1)

Sends notification of transmit completion immediately upon completion of transmit.

Poll (2) Queries the adapter for transmit status based on the number of packets transmitted. This parameter is used for performance tuning and should be set according to network usage.

Note: Under **Delay** and **Poll** modes, a timer is used to ensure timely process completion of transmit packets.

Receive Interrupt Mode

The Ethernet High-Performance LAN Adapter (8f95) can be configured to operate in one of two receive modes.

Delay (0) Sends notification of an incoming packet based on the number of packets currently in the receive queue.
Note: Under Delay mode, a timer is used to ensure that all received packets are processed efficiently.

Immediate (1) Sends notification of an incoming packet immediately upon receipt of the packet.

Transmit Interrupt Threshold

Under delayed transmit mode for the Ethernet High-Performance LAN Adapter (8f95), the frequency of transmit complete interrupts can be controlled based on the *Transmit Interrupt Threshold* parameter. The adapter issues an interrupt when the number of transmitted packets exceeds this threshold. For example, if the transmit interrupt threshold parameter is **0**, the adapter issues an interrupt when 1 transmit packet is complete. If the transmit interrupt threshold parameter is **1**, the adapter issues an interrupt when 2 transmit packets are complete. This pattern continues until the *Transmit Interrupt Threshold* parameter reaches its maximum value of **31**.

Note: This parameter should be used for performance tuning only.

Receive Interrupt Threshold

Under delayed receive mode for the Ethernet High-Performance LAN Adapter (8f95), the frequency of receive complete interrupts can be controlled based on the *Receive Interrupt Threshold* parameter. The adapter issues an interrupt when the number of received packets exceeds this threshold. For example, if the *Receive Interrupt Threshold* parameter is **0**, the adapter issues an interrupt when 1 receive packet is complete. If the *Receive Interrupt Threshold* parameter is **1**, the adapter issues an interrupt when 2 receive packets are complete. This pattern continues until the *Receive Interrupt Threshold* parameter reaches its maximum value of **31**.

Note: This parameter should be used for performance tuning only.

Transmit Poll Threshold

Under transmit poll mode for the Ethernet High-Performance LAN Adapter (8f95), the frequency in which the device driver polls the adapter for completed transmit packets can be controlled based on the *Transmit Poll Threshold* parameter. The device driver polls for completed transmit status when the number of outstanding transmitted packets exceeds this threshold. If the *Transmit Poll Threshold* parameter is **0**, the device driver polls the adapter for status when 1 transmit packet status is pending. If the *Transmit Poll Threshold* parameter is **1**, the device driver polls the adapter

for status when status for 2 transmit packets is pending. This pattern continues until the *Transmit Poll Threshold* parameter reaches its maximum of 31.

Note: This parameter should be used for performance tuning only.

Receive Interval

Under receive delayed mode for the Ethernet High-Performance LAN Adapter (8f95), the maximum amount of time between receive interrupts can be controlled based on the *Receive Interval* parameter. The adapter guarantees that a receive interrupt is generated within $2^{**}(\text{receive Interval} + 7)/10$ microseconds after the last received packet, regardless of the value of the *Receive Interrupt Threshold* parameter. This timer is reset to zero by the adapter after each packet is received.

Duplex

The Ethernet High-Performance LAN Adapter (8f95) can be configured to operate in a full duplex 10BASET network. This mode of operation is only valid using the adapter's RJ-45 (10BASET) port. Duplex mode is not valid when using the AUI port or the BNC (10BASE2) port.

Beginning with AIX Version 4.1.5, the 10/100 Mbps Ethernet TX MCA device driver (8f62) supports the following additional configuration parameters:

Hardware Transmit Queue

The 10/100 Mbps Ethernet TX MCA device driver (8f62) supports a user-configurable transmit queue for the adapter. This is the actual queue the adapter uses to transmit packets. Each element corresponds to an Ethernet packet. It is configurable at 16, 32, 64, 128, and 256 elements. The default is 64.

Hardware Receive Queue

The 10/100 Mbps Ethernet TX MCA device driver (8f62) supports a user-configurable receive queue for the adapter. This is the actual queue the adapter uses to receive packets. Each element corresponds to an Ethernet packet. It is configurable at 16, 32, 64, 128, and 256 elements. The default is 64.

Media Speed

The 10/100 Mbps Ethernet TX MCA device driver (8f62) supports a user-configurable media speed for the adapter. The **media speed** attribute indicates the speed at which the adapter will attempt to operate. The available speeds are: 10 Mbps half-duplex, 10 Mbps full-duplex, 100 Mbps half-duplex, 100 Mbps full-duplex, and auto-negotiation. Select auto-negotiate when the adapter should use auto-negotiation across the network to determine the speed. When the network will not support auto-negotiation, the specific speed should be selected. The default is auto-negotiation.

Inter Packet Gap (IPG)

The 10/100 Mbps Ethernet TX MCA device driver (8f62) supports a user-configurable inter packet gap for the adapter. The **inter packet gap** attribute controls the aggressiveness of the adapter on the network. A small number increases the aggressiveness of the adapter, while a large number decreases the aggressiveness (and increases the fairness) of the adapter. A small number (more aggressive) could cause the adapter to capture the network by forcing other less aggressive nodes to defer. A larger number (less aggressive) could cause the adapter to defer more often than normal. If the statistics for other nodes on the network show a large number of

collisions and deferrals, try increasing this number. The default is 96, which results in an IPG of 9.6 microseconds for 10 Mbps and 0.96 microseconds for 100 Mbps media speed. Each unit of bit rate introduces an IPG of 100 nsec at 10 Mbps and 10 nsec at 100 Mbps media speed.

The PCI Ethernet Device Driver (22100020) supports the following additional configuration parameters:

Full Duplex

Indicates whether the adapter is operating in full-duplex or half-duplex mode. If this field is set to yes, the device driver programs the adapter to be in full-duplex mode. The default is half-duplex.

Note: Full duplex mode is valid for AIX Version 4.1.5 (and later).

Hardware Transmit Queue

Specifies the actual queue the adapter uses to transmit packets. Each element corresponds to an Ethernet packet. It is configurable at 16, 32, 64, 128, and 256 elements. The default is 64.

Hardware Receive Queue

Specifies the actual queue the adapter uses to receive packets. Each element corresponds to an Ethernet packet. It is configurable at 16, 32, 64, 128, and 256 elements. The default is 64.

The PCI 10/100 Mbps Ethernet Device Driver (23100020) supports the following additional configuration parameters:

Hardware Transmit Queue

The PCI 10/100 Mbps Ethernet Device Driver (23100020) supports a user-configurable transmit queue for the adapter. This is the actual queue the adapter uses to transmit packets. Each element corresponds to an Ethernet packet. It is configurable at 16, 32, 64, 128, and 256 elements, with a default of 64.

Hardware Receive Queue

The PCI 10/100 Mbps Ethernet Device Driver (23100020) supports a user-configurable receive queue for the adapter. This is the actual queue the adapter uses to receive packets. Each element corresponds to an Ethernet packet. It is configurable at 16, 32, 64, 128, and 256 elements, with a default of 32.

Media Speed

The PCI 10/100 Mbps Ethernet Device Driver (23100020) supports a user-configurable media speed for the adapter. The media speed attribute indicates the speed at which the adapter will attempt to operate. The available speeds are 10 Mbps half-duplex, 10 Mbps full-duplex, 100 Mbps half-duplex, 100 Mbps full-duplex and auto-negotiation, with a default of auto-negotiation. Select auto-negotiate when the adapter should use auto-negotiation across the network to determine the speed. When the network will not support auto-negotiation, the specific speed should be selected.

Inter Packet Gap

The PCI 10/100 Mbps Ethernet Device Driver (23100020) supports a user-configurable inter packet gap for the adapter. The inter packet gap attribute controls the aggressiveness of the adapter on the network. A small number will increase the aggressiveness of the adapter, while a large number will decrease the aggressiveness (and increase the fairness) of the

adapter. A small number (more aggressive) could cause the adapter to capture the network by forcing other less aggressive nodes to defer. A larger number (less aggressive) may cause the adapter to defer more often than normal. If the statistics for other nodes on the network show a large number of collisions and deferrals, then try increasing this number. The default is 96, which results in IPG of 9.6 micro seconds for 10 Mbps and 0.96 microseconds for 100 Mbps media speed. Each unit of bit rate introduces an IPG of 100 nsec at 10 Mbps, and 10 nsec at 100 Mbps media speed.

The Gigabit Ethernet-SX PCI Adapter (14100401) device driver supports the additional configuration parameters:

Software Transmit Queue Size

Indicates the number of transmit requests that can be queued for transmission by the device driver. Valid values range from 512 through 2048. The default value is 512.

Transmit Jumbo Frames

Setting this attribute to the yes value indicates that frames up to 9018 bytes in length may be transmitted on this adapter. If you specify the no value, the maximum size of frames transmitted is 1518 bytes. The default value is no. Frames up to 9018 bytes in length may always be received on this adapter.

Transmit and Receive Jumbo Frames

Setting this attribute to the yes value indicates that frames up to 9014 bytes in length may be transmitted or received on this adapter. If you specify the no value, the maximum size of frames transmitted or received is 1514 bytes. The default value is no.

Receive Buffer Pool Size

Indicates the number of **mbufs** to be used exclusively with this adapter. These **mbufs** will be used for receiving frames. They will be 4096 bytes long if yes is specified for the **Transmit Jumbo Frames** attribute (see 163). They will be 2048 bytes long otherwise. Valid values range from 256 through 2048. The default value is 768. The adapter has a receive queue of 512 entries. Each entry describes a **mbuf** where a frame (or part of a frame) will be received. The device driver will attempt to obtain a **mbuf** for the receive queue from this receive buffer pool. If the pool is empty the device driver will attempt to obtain a **mbuf** from the system buffer pool. After a frame is received the **mbuf** containing the frame will be passed to the user of that frame. A replacement **mbuf** will be obtained for the adapter receive queue. Thus more than 512 **mbuf** will be in use at any given time. The output of the `ntstat -d ent0` program contains statistics concerning use of this buffer pool. Use of **mbufs** from this pool will improve the performance of the adapter with a possible increase in system network memory usage.

Enable Hardware Receive Checksum

Setting this attribute to the yes value indicates that the adapter should calculate the checksum for received TCP frames. If you specify the no value, the checksum will be calculated by the appropriate software. The default value is yes.

Note: The **mbuf** describing a frame to be transmitted contains a flag which says if the adapter should calculate the checksum for the frame.

Interface Entry Points

Device Driver Configuration and Unconfiguration

The configuration entry points of the device drivers conform to the guidelines for AIX Version 4 kernel object file entry points. The configuration entry points are **en3com_config** for the Ethernet High-Performance LAN Adapter (8ef5), **ient_config** for the Integrated Ethernet, **kent_config** for the PCI Ethernet Device Driver (22100020), and **lce_config** for the Ethernet High-Performance LAN Adapter (8f95). Beginning with AIX Version 4.1.5, the **srent_config** entry point is available for the 10/100 Mbps Ethernet TX MCA (8f62) device driver. Beginning with AIX Version 4.1.5, the **phxent_config** entry point is available for the PCI 10/100 Mbps Ethernet (23100020) Device Driver. The Gigabit Ethernet-SX PCI Adapter (14100401) entry point is **gxent_config**.

Device Driver Open

The open entry point for the device drivers perform a synchronous open of the specified network device.

The device driver issues commands to start the initialization of the device. The state of the device now is OPEN_PENDING. The device driver invokes the open process for the device. The open process involves a sequence of events that are necessary to initialize and configure the device. The device driver will do the sequence of events in an orderly fashion to make sure that one step is finished executing on the adapter before the next step is continued. Any error during these sequence of events will make the open fail. The device driver requires about 2 seconds to open the device. When the whole sequence of events is done, the device driver verifies the open status and then returns to the caller of the open with a return code to indicate open success or open failure.

Once the device has been successfully configured and connected to the network, the device driver will set the device state to OPENED, the NDD_RUNNING flag in the NDD flags field will be turned on. In the case of unsuccessful open, both the NDD_UP and NDD_RUNNING flags in the NDD flags field will be off and a non-zero error code will be returned to the caller.

The open entry points are **en3com_open** for the Ethernet High-Performance LAN Adapter (8ef5), **ient_open** for the Integrated Ethernet, **kent_open** for the PCI Ethernet Device Driver (22100020), and **lce_open** for the Ethernet High-Performance LAN Adapter (8f95). Beginning with AIX Version 4.1.5, the **srent_open** entry point is available for the 10/100 Mbps Ethernet TX MCA (8f62) device driver. Beginning with AIX Version 4.1.5, the **phxent_open** entry point is available for the PCI 10/100 Mbps Ethernet (23100020) Device Driver. The Gigabit Ethernet-SX PCI Adapter (14100401) entry point is **gxent_open**.

Device Driver Close

The close entry point for the device drivers is called to close the specified network device. This function resets the device to a known state and frees system resources associated with the device.

The device will not be detached from the network until the device's transmit queue is allowed to drain. That is, the close entry point will not return until all packets have been transmitted or timed out. If the device is inoperable at the time of the close, the device's transmit queue does not have to be allowed to drain.

At the beginning of the close entry point, the device state will be set to be CLOSE_PENDING. The NDD_RUNNING flag in the ndd_flags will be turned off. After the outstanding transmit queue is all done, the device driver will start a sequence of operations to deactivate the adapter and to free up resources. Before the close entry point returns to the caller, the device state is set to CLOSED.

The close entry points are **en3com_close** for the Ethernet High-Performance LAN Adapter (8ef5), **ient_close** for the Integrated Ethernet, **kent_close** for the PCI Ethernet Device Driver (22100020), and **lce_close** for the Ethernet High-Performance LAN Adapter (8f95). Beginning with AIX Version 4.1.5, the **srent_close** entry point is available for the 10/100 Mbps Ethernet TX MCA (8f62) device driver. Beginning with AIX Version 4.1.5, the **phxent_close** entry point is available for the PCI 10/100 Mbps Ethernet (23100020) Device Driver. The Gigabit Ethernet-SX PCI Adapter (14100401) entry point is **gxent_close**.

Data Transmission

The output entry point transmits data using the specified network device.

The data to be transmitted is passed into the device driver by way of mbuf structures. The first mbuf in the chain must be of M_PKTHDR format. Multiple mbufs may be used to hold the frame. The mbufs should be linked using the **m_next** field of the **mbuf** structure.

Multiple packet transmits are allowed with the mbufs being chained using the **m_nextpkt** field of the **mbuf** structure. The **m_pkthdr.len** field must be set to the total length of the packet. The device driver does *not* support mbufs from user memory (which have the M_EXT flag set).

On successful transmit requests, the device driver is responsible for freeing all the mbufs associated with the transmit request. If the device driver returns an error, the caller is responsible for the mbufs. If any of the chained packets can be transmitted, the transmit is considered successful and the device driver is responsible for all of the mbufs in the chain.

If the destination address in the packet is a broadcast address the M_BCAST flag in the **m_flags** field should be set prior to entering this routine. A broadcast address is defined as 0xFFFF FFFF FFFF. If the destination address in the packet is a multicast address the M_MCAST flag in the **m_flags** field should be set prior to entering this routine. A multicast address is defined as a non-individual address other than a broadcast address. The device driver will keep statistics based upon the M_BCAST and M_MCAST flags.

For packets that are shorter than the Ethernet minimum MTU size (60 bytes), the device driver will pad them by adjusting the transmit length to the adapter so they can be transmitted as valid Ethernet packets.

Users will not be notified by the device driver about the status of the transmission. Various statistics about data transmission are kept by the driver in the **ndd** structure. These statistics will be part of the data returned by the NDD_GET_STATS control operation.

The output entry points are **en3com_output** for the Ethernet High-Performance LAN Adapter (8ef5), **ient_output** for the Integrated Ethernet, **kent_output** for the PCI Ethernet Device Driver (22100020), and **lce_output** for the Ethernet High-Performance LAN Adapter (8f95). Beginning with AIX Version 4.1.5, the **srent_output** entry point is available for the 10/100 Mbps Ethernet TX MCA (8f62)

device driver. Beginning with AIX Version 4.1.5, the **phxent_output** entry point is available for the PCI 10/100 Mbps Ethernet (23100020) Device Driver. The Gigabit Ethernet-SX PCI Adapter (14100401) entry point is **gxent_output**.

Data Reception

When the Ethernet device drivers receive a valid packet from the network device, the device drivers call the **nd_receive** function that is specified in the **ndd_t** structure of the network device. The **nd_receive** function is part of a CDLI network demuxer. The packet is passed to the **nd_receive** function in the form of a mbuf.

The Ethernet device drivers may pass multiple packets to the **nd_receive** function by chaining the packets together using the **m_nextpkt** field of the mbuf structure. The **m_pkthdr.len** field must be set to the total length of the packet. If the source address in the packet is a broadcast address the **M_BCAST** flag in the **m_flags** field should be set. If the source address in the packet is a multicast address the **M_MCAST** flag in the **m_flags** field should be set.

When the device driver initially configures the device to discard all invalid frames. A frame is considered to be invalid for the following reasons:

- The packet is too short
- The packet is too long
- The packet contains a CRC error
- The packet contains an alignment error.

If the asynchronous status for receiving invalid frames has been issued to the device driver, the device driver will configure the device to receive bad packets as well as good packets. Whenever a bad packet is received by the driver, an asynchronous status block **NDD_BAD_PKTS** is created and delivered to the appropriate user. The user must copy the contents of the mbuf to another memory area. The user must not modify the contents of the mbuf or free the mbuf. The device driver has the responsibility of releasing the mbuf upon returning from **nd_status**.

Various statistics about data reception on the device will be kept by the driver in the **ndd** structure. These statistics will be part of the data returned by the **NDD_GET_STATS** and **NDD_GET_ALL_STATS** control operations.

There is no specified entry point for this function. The device informs the device driver of a received packet via an interrupt. Upon determining that the interrupt was the result of a packet reception, the device driver's interrupt handler will invoke a completion routine to perform the tasks mentioned above. This is **en3com_rv_intr** for the Ethernet High-Performance LAN Adapter (8ef5), **ient_RU_complete** for the Integrated Ethernet, **rx_handler** for the 10/100 Mbps Ethernet TX MCA (8f62) device driver (AIX Version 4.1.5 and later) and the PCI Ethernet device driver (22100020), and **Ice_recv** for the Ethernet High-Performance LAN Adapter (8f95). Beginning with AIX Version 4.1.5, the **rx_handler** entry point is available for the PCI 10/100 Mbps Ethernet (23100020) Device Driver. The Gigabit Ethernet-SX PCI Adapter (14100401) entry point is **rx_handler**.

Asynchronous Status

When a status event occurs on the device, the Ethernet device drivers build the appropriate status block and call the **nd_status** function that is specified in the **ndd_t** structure of the network device. The **nd_status** function is part of a CDLI network demuxer.

The following Status Blocks are defined for the Ethernet device drivers.

Note: The PCI Ethernet Device Driver (22100020) and the Ethernet High-Performance LAN Adapter (8f95) only support the Bad Packets status block. The Gigabit Ethernet-SX PCI Adapter (14100401) does not support asynchronous status.

Hard Failure

When a hard failure has occurred on the Ethernet device, the following status blocks can be returned by the Ethernet device driver. These status blocks indicates that a fatal error occurred.

code	Set to NDD_HARD_FAIL.
option[0]	Set to one of the reason codes defined in <code><sys/ndd.h></code> and <code><sys/cdli_entuser.h></code> .

Enter Network Recovery Mode

When the device driver has detected an error which requires initiating recovery logic that will make the device temporarily unavailable, the following status block is returned by the device driver.

code	Set to NDD_LIMBO_ENTER.
option[0]	Set to one of the reason codes defined in <code><sys/ndd.h></code> and <code><sys/cdli_entuser.h></code> .

Note: While the device driver is in this recovery logic, the device may not be fully functional. The device driver will notify users when the device is fully functional by way of an NDD_LIMBO_EXIT asynchronous status block,

Exit Network Recovery Mode

When the device driver has successfully completed recovery logic from the error that made the device temporarily unavailable, the following status block is returned by the device driver.

code	Set to NDD_LIMBO_EXIT.
option[]	The option fields are not used.

Note: The device is now fully functional.

Network Device Driver Status

When the device driver has status or event information to report, the following status block is returned by the device driver.

code	Set to NDD_STATUS.
option[0]	May be any of the common or interface type specific reason codes.
option[]	The remainder of the status block may be used to return additional status information by the device driver.

Bad Packets

When the a bad packet has been received by a device driver (and a user has requested bad packets), the following status block is returned by the device driver.

code	Set to NDD_BAD_PKTS.
option[0]	Specifies the error status of the packet. These error numbers are defined in <code><sys/cdli_entuser.h></code> .
option[1]	Pointer to the mbuf containing the bad packet.
option[]	The remainder of the status block may be used to return additional status information by the device driver.

Note: The user will *not* own the mbuf containing the bad packet. The user must copy the mbuf (and the status block information if desired). The device driver will free the mbuf upon return from the `nd_status` function.

Device Connected

When the device is successfully connected to the network the following status block is returned by the device driver.

code	Set to NDD_CONNECTED.
option[]	The option fields are not used.

Note: Integrated Ethernet Only.

Device Control Operations

The `ndd_ctl` entry point is used to provide device control functions.

NDD_GET_STATS

The `NDD_GET_STATS` command returns statistics concerning the network device. General statistics are maintained by the device driver in the `ndd_genstats` field in the `ndd_t` structure. The `ndd_specstats` field in the `ndd_t` structure is a pointer to media-specific and device-specific statistics maintained by the device driver. Both sets of statistics are directly readable at any time by those users of the device that can access them. This command provides a way for any of the users of the device to access the general and media-specific statistics. The `NDD_GET_ALL_STATS` command provides a way to get the device-specific statistics also. Beginning with AIX Version 4.1, the `phxent_all_stats_t` structure is available for the PCI 10/100 Mbps Ethernet (23100020) Device Driver. This structure is defined in the device-specific include file `cdli_entuser.phxent.h`.

The `arg` and `length` parameters specify the address and length in bytes of the area where the statistics are to be written. The length specified *must* be the exact length of the general and media-specific statistics.

Note: The `ndd_speclen` field in the `ndd_t` structure plus the length of the `ndd_genstats_t` structure is the required length. The device-specific statistics may change with each new release of AIX, but the general and media-specific statistics are not expected to change.

The user should pass in the `ent_ndd_stats_t` structure as defined in `<sys/cdli_entuser.h>`. The driver fails a call with a buffer smaller than the structure.

The statistics which are returned contain statistics obtained from the device. If the device is inoperable, the statistics which are returned will not contain the current device statistics. The copy of the `ndd_flags` field can be checked to determine the state of the device.

NDD_MIB_QUERY

The `NDD_MIB_QUERY` operation is used to determine which device-specific MIBs are supported on the network device. The *arg* and *length* parameters specify the address and length in bytes of a device-specific MIB structure. The device driver will fill every member of that structure with a flag indicating the level of support for that member. The individual MIB variables that are not supported on the network device will be set to `MIB_NOT_SUPPORTED`. The individual MIB variables that may only be read on the network device will be set to `MIB_READ_ONLY`. The individual MIB variables that may be read and set on the network device will be set to `MIB_READ_WRITE`. The individual MIB variables that may only be set (not read) on the network device will be set to `MIB_WRITE_ONLY`. These flags are defined in the `/usr/include/sys/ndd.h` file.

The *arg* parameter specifies the address of the `ethernet_all_mib` structure. This structure is defined in the `/usr/include/sys/ethernet_mibs.h` file.

NDD_MIB_GET

The `NDD_MIB_GET` operation is used to get all MIBs on the specified network device. The *arg* and *length* parameters specify the address and length in bytes of the device specific MIB structure. The device driver will set any unsupported variables to zero (nulls for strings).

If the device supports the RFC 1229 receive address object, the corresponding variable is set to the number of receive addresses currently active.

The *arg* parameter specifies the address of the `ethernet_all_mib` structure. This structure is defined in the `/usr/include/sys/ethernet_mibs.h` file.

NDD_ENABLE_ADDRESS

The `NDD_ENABLE_ADDRESS` command enables the receipt of packets with an alternate (for example, multicast) address. The *arg* and *length* parameters specify the address and length in bytes of the alternate address to be enabled. The `NDD_ALTADDRS` flag in the `ndd_flags` field is set.

The device driver verifies that if the address is a valid multicast address. If the address is not a valid multicast address, the operation will fail with an `EINVAL` error. If the address is valid, the driver will add it to its multicast table and enable the multicast filter on the adapter. The driver will keep a reference count for each individual address. Whenever a duplicate address is registered, the driver simply increments the reference count of that address in its multicast table, no update of the adapter's filter is needed. There is a hardware limitation on the number of multicast addresses in the filter.

NDD_DISABLE_ADDRESS

The `NDD_DISABLE_ADDRESS` command disables the receiving packets with a specified alternate (for example, multicast) address. The *arg* and *length* parameters specify the address and length in bytes of the alternate address to be disabled. The `NDD_ALTADDRS` flag in the `ndd_flags` field is reset if this is the last alternate address.

The device driver verifies that if the address is a valid multicast address. If the address is not a valid multicast address, the operation will fail with an `EINVAL`

error. The device driver makes sure that the multicast address can be found in its multicast table. Whenever a match is found, the driver will decrement the reference count of that individual address in its multicast table. If the reference count becomes 0, the driver will delete the address from the table and update the multicast filter on the adapter.

NDD_MIB_ADDR

The `NDD_MIB_ADDR` operation is used to get all the addresses for which the specified device will accept packets or frames. The *arg* parameter specifies the address of the `ndd_mib_addr_t` structure. The *length* parameter specifies the length of the structure with the appropriate number of `ndd_mib_addr_t.mib_addr` elements. This structure is defined in the `/usr/include/sys/ndd.h` file. If the *length* is less than the length of the `ndd_mib_addr_t` structure, the device driver returns `EINVAL`. If the structure is not large enough to hold all the addresses, the addresses which fit will still be placed in the structure. The `ndd_mib_addr_t.count` field is set to the number of addresses returned and `E2BIG` is returned.

One of the following address types is returned:

- Device physical address (or alternate address specified by user)
- Broadcast addresses
- Multicast addresses

NDD_CLEAR_STATS

The counters kept by the device will be zeroed.

NDD_GET_ALL_STATS

The `NDD_GET_ALL_STATS` operation is used to gather all the statistics for the specified device. The *arg* parameter specifies the address of the statistics structure for the particular device type. This structure is `en3com_all_stats_t` for the Ethernet High-Performance LAN Adapter (8ef5), `ient_all_stats_t` for the Integrated Ethernet Device, `kent_all_stats_t` for the PCI Ethernet Device Driver (22100020), and `enlce_all_stats_t` for the Ethernet High-Performance LAN Adapter (8f95). Beginning with AIX Version 4.1.5, the `srent_all_stats_t` structure is available for the 10/100 Mbps Ethernet TX MCA (8f62) device driver. These structures are defined in the `/usr/include/sys/cdli_entuser.h` file.

The statistics which are returned contain statistics obtained from the device. If the device is inoperable, the statistics which are returned will not contain the current device statistics. The copy of the `ndd_flags` field can be checked to determine the state of the device.

NDD_ENABLE_MULTICAST

The `NDD_ENABLE_MULTICAST` command enables the receipt of packets with any multicast (or group) address. The *arg* and *length* parameters are not used. The `NDD_MULTICAST` flag in the `ndd_flags` field is set.

Note: Unlike the Integrated Ethernet and PCI Ethernet (22100020) Device Drivers, the Ethernet High-Performance LAN Adapter (8ef5) adapter does *not* support the "receive all multicast" function; this driver will enable the promiscuous mode on the adapter in order to bypass the multicast filtering existing on the adapter. The device driver performs additional packet filtering to discard packets which are not supposed to be received under this circumstance.

NDD_DISABLE_MULTICAST

The `NDD_DISABLE_MULTICAST` command disables the receipt of ALL packets with multicast addresses and only receives those packets whose multicast addresses were specified using the `NDD_ENABLE_ADDRESS` command. The *arg* and *length* parameters are not used. The `NDD_MULTICAST` flag in the `ndd_flags` field is reset only after the reference count for multicast addresses has reached zero.

NDD_PROMISCUOUS_ON

The `NDD_PROMISCUOUS_ON` command turns on promiscuous mode. The *arg* and *length* parameters are not used.

When the device driver is running in promiscuous mode, "all" network traffic is passed to the network demuxer. When the Ethernet device driver receives a valid packet from the network device, the Ethernet device driver calls the `nd_receive` function that is specified in the `ndd_t` structure of the network device. The `NDD_PROMISC` flag in the `ndd_flags` field is set. Promiscuous mode is considered to be valid packets only. See the `NDD_ADD_STATUS` command for information about how to request support for bad packets.

The device driver will maintain a reference count on this operation. The device driver increments the reference count for each operation. When this reference count is equal to one, the device driver issues commands to enable the promiscuous mode. If the reference count is greater than one, the device driver does not issue any commands to enable the promiscuous mode.

NDD_PROMISCUOUS_OFF

The `NDD_PROMISCUOUS_OFF` command terminates promiscuous mode. The *arg* and *length* parameters are not used. The `NDD_PROMISC` flag in the `ndd_flags` field is reset.

The device driver will maintain a reference count on this operation. The device driver decrements the reference count for each operation. When the reference count is not equal to zero, the device driver does not issue commands to disable the promiscuous mode. Once the reference count for this operation is equal to zero, the device driver issues commands to disable the promiscuous mode.

NDD_DUMP_ADDR

The `NDD_DUMP_ADDR` command returns the address of the device driver's remote dump routine. The *arg* parameter specifies the address where the dump routine's address is to be written. The *length* parameter is not used.

Note: The Ethernet High-Performance LAN Adapters (8ef5 and 8f95) Device Drivers do not support this.

Reliability, Availability, and Serviceability (RAS)

Trace

For LAN device drivers, trace points enable error monitoring as well as tracking packets as they move through the driver. The drivers issue trace points for some or all of the following conditions:

- Beginning and ending of main functions in the main path.
- Error conditions.
- Beginning and ending of each function that is tracking buffers outside of the main path.

- Debugging purposes. (These trace points are only enabled when the driver is compiled with **-DDEBUG** turned on, and therefore the driver can contain as many of these trace points as desired.)

The existing Ethernet device drivers each have either three or four trace points. The Trace Hook IDs for most of the device types are defined in the **sys/cdli_entuser.h** file. Other drivers have defined local **cdli_entuser.driver.h** files with the Trace Hook definitions.

Following is a list of trace hooks (and location of definition file) for the existing Ethernet device drivers:

- IBM ISA 16-bit Ethernet Adapter
 - Definition file: **cdli_entuser.h**
 - Trace Hook IDs:

Transmit	-330
Receive	-331
Errors	-332
Other	-333

- Ethernet High-Performance Adapter (8ef5)
 - Definition file: **cdli_entuser.h**
 - Trace Hook IDs:

Transmit	-351
Receive	-352
Errors	-353
Other	-354

- 10Mb MCA Low Cost High Performance Ethernet Device Driver (8f95)
 - Definition file: **cdli_entuser.h**
 - Trace Hook IDs:

Transmit	-327
Receive	-328
Errors	-37D
Other	-37E

- Integrated Ethernet Device Drivers (8f98, 8ef2, 8ef3)
 - Definition file: **cdli_entuser.h**
 - Trace Hook IDs:

Transmit	-320
Receive	-321
Errors	-322
Other	-323

- 10/100 Mbps Ethernet TX MCA Device Driver (8f62)
 - Definition file: **cdli_entuser.srent.h**
 - Trace Hook IDs:

Transmit	-2C3
Receive	-2C4
Other	-2C5

- PCI Ethernet Device Driver (22100020)

- Definition file: **cdli_entuser.h**

- Trace Hook IDs:

Transmit>	-2A4
Receive	-2A5
Other	-2A6

- PCI 10/100 Mbps Ethernet Device Driver (23100020)

- Definition file: **cdli_entuser.phxent.h**

- Trace Hook IDs:

Transmit	-2E6
Receive	-2E7
Other	-2E8

- Gigabit Ethernet-SX PCI Adapter (14100401)

- Definition file: **cdli_entuser.gxent.h**

- Trace Hook IDs:

Transmit	-2EA
Receive	-2EB
Other	-2EC

The device driver also has the following trace points to support the **netpmon** program:

WQUE An output packet has been queued for transmission

WEND The output of a packet is complete

RDAT An input packet has been received by the device driver

RNOT An input packet has been given to the demuxer

REND The demuxer has returned

For more information, see “Debug and Performance Tracing” on page 554.

Error Logging

The Error IDs for the Ethernet High-Performance LAN Adapter (8ef5) are as follows:

ERRID_EN3COM_TMOUT

The watchdog timer has expired while waiting on acknowledgement of either a control command or transmit command. The device driver will go into Network Recovery Mode in an attempt to recover from the error. The device is temporarily unavailable during the recovery procedure. User intervention is not required for this error unless the problem persists.

ERRID_EN3COM_FAIL

The device driver has detected an error that prevents the device from functioning. This message is normally preceded by another error log which indicates the specific fatal error that has occurred. The device driver may have gone through the Network Recovery Mode and failed to recover from the error. This message indicates that the device will not be available due to some hard failure and user intervention is required.

ERRID_EN3COM_UCODE

The device driver detected an error in the microcode on the adapter. The

device driver will log this error and indicate hardware failure. The device will not be available after this error is detected. User intervention is required in order to recover from this error.

ERRID_EN3COM_PARITY

The device detected a parity error. The device driver will log this error and go into Network Recovery Mode in an attempt to recover from the error. The device is temporarily unavailable during the recovery procedure. User intervention is not required for this error unless the problem persists.

ERRID_EN3COM_DMAFAIL

The device has detected a DMA channel error or a Micro Channel error has occurred. Normally, this error will be accompanied by another error that will indicate if this error is fatal or recoverable.

ERRID_EN3COM_NOBUFS

The device detected a memory shortage during the device initialization phase when the device driver attempted to allocate transmit and receive buffers from the host memory. The device driver will log this error and fail the device initialization. The device will not be available after this error is detected. User intervention is required in order to recover from this error.

ERRID_EN3COM_PIOFAIL

The device detected an I/O channel error or an error in a command the device driver issued, an error occurred during a PIO operation, or the device has detected an error in a packet given to the device. The device driver will retry the operation for three times. If they all failed, the device driver will log this error and indicate hardware failure. The device will not be available after this error is detected. User intervention is required in order to recover from this error.

The Error IDs for the Integrated Ethernet Device Driver are as follows:

ERRID_IENT_TMOU

The watchdog timer has expired while waiting on acknowledgement of either a control command or transmit command. The device driver will go into Network Recovery Mode in an attempt to recover from the error. The device is temporarily unavailable during the recovery procedure. User intervention is not required for this error unless the problem persists.

ERRID_IENT_PIOFAIL

The device detected an I/O channel error or an error in a command the device driver issued, an error occurred during a PIO operation, or the device has detected an error in a packet given to the device. The device driver will go into Network Recovery Mode in an attempt to recover from the error. The device is temporarily unavailable during the recovery procedure. User intervention is not required for this error unless the problem persists.

ERRID_IENT_DMAFAIL

The device has detected an DMA channel error or a Micro Channel error has occurred. The device driver will go into Network Recovery Mode in an attempt to recover from the error. The device is temporarily unavailable during the recovery procedure. User intervention is not required for this error unless the problem persists.

ERRID_IENT_FAIL

The device has detected an error that prevents the device from starting or restarting, such as **pincode** or **i_init** fails. If the device is restarting in Network Recovery Mode in an attempt to recover from an error, the device

will be temporarily unavailable during the recovery procedure. User intervention is not required for this error unless the problem persists.

Beginning with AIX Version 4.1.5, the Error IDs for the 10/100 Mbps Ethernet TX MCA (8f62) device driver are as follows:

ERRID_SRENT_ADAP_ERR

Indicates that the adapter is not responding to initialization commands. User intervention is necessary to fix the problem.

ERRID_SRENT_RCVRY

Indicates that the adapter hit a temporary error requiring that it enter network recovery mode. The adapter is reset in an attempt to fix the problem.

ERRID_SRENT_TX_ERR

Indicates that the device driver has detected a transmission error. User intervention is not required unless the problem persists.

ERRID_SRENT_PIO

Indicates that the device driver has detected a program IO error. User intervention is necessary to fix the problem.

ERRID_SRENT_DOWN

Indicates that the device driver has shutdown the adapter due to an unrecoverable error. The adapter is no longer functional. The error that caused the device to shutdown is logged immediately before this error log entry. User intervention is necessary to fix the problem.

ERRID_SRENT_EEPROM_ERR

Indicates that the device driver is in a defined state due to an invalid or bad EEPROM. The device driver will not become available. Contact your hardware support representative.

The Error IDs for the PCI Ethernet Device Driver (22100020) are as follows:

ERRID_KENT_ADAP_ERR

Indicates that the adapter is not responding to initialization commands. User intervention is necessary to fix the problem.

ERRID_KENT_RCVRY

Indicates that the adapter hit a temporary error requiring that it enter network recovery mode. It has reset the adapter in an attempt to fix the problem.

ERRID_KENT_TX_ERR

Indicates the the device driver has detected a transmission error. User intervention is not required unless the problem persists.

ERRID_KENT_PIO

Indicates that the device driver has detected a program IO error. The device driver was unable to fix the problem. User intervention is necessary to fix the problem.

ERRID_KENT_DOWN

Indicates that the device driver has shut down the adapter due to an unrecoverable error. The adapter is no longer functional due to the error. The error that caused the device to shut down is error logged immediately before this error log entry. User intervention is necessary to fix the problem.

Beginning with AIX Version 4.1.5, the Error IDs for the PCI 10/100 Mbps Ethernet Device Driver (23100020) are as follows:

ERRID_PHXENT_ADAP_ERR

Indicates that the adapter is not responding to initialization commands. User-intervention is necessary to fix the problem.

ERRID_PHXENT_TX_RCVRY

Indicates that the adapter hit a temporary error requiring that it enter network recovery mode. It has reset the adapter in an attempt to fix the problem.

ERRID_PHXENT_TX_ERR

Indicates that the device driver has detected a transmission error. User-intervention is not required unless the problem persists.

ERRID_PHXENT_PIO

Indicates that the device driver has detected a program IO error. The device driver was unable to fix the problem. User-intervention is necessary to fix the problem.

ERRID_PHXENT_DOWN

Indicates that the device driver has shutdown the adapter due to an unrecoverable error. The adapter is no longer functional due to the error. The error which caused the device shutdown is error logged immediately before this error log entry. User-intervention is necessary to fix the problem.

ERRID_PHXENT_EEPROM_ERR

Indicates that the device driver is in a defined state due to an invalid or bad EEPROM. The device driver will not become available. Hardware support should be contacted.

The Error IDs for the Ethernet High-Performance LAN Adapter (8f95) are as follows:

ERRID_ENLCE_TMOUT

Indicates status for a transmit packet was not received. The device will not be available during the error recovery process.

ERRID_ENLCE_FAIL

Indicates that the adapter has reported a hardware error. The device will not be available during the error recovery process.

ERRID_ENLCE_SWFAIL

Indicates the device driver has detected a software error. The current operation will not complete successfully.

ERRID_ENLCE_TXFAIL

Indicates a hardware/software transmit synchronization problem. The device will not be available during the error recovery process.

ERRID_ENLCE_RXFAIL

Indicates a hardware/software receive synchronization problem. The device will not be available during the error recovery process.

ERRID_ENLCE_MCFAIL

Indicates that the adapter has reported a Micro Channel error. The device will not be available during the error recovery process.

ERRID_ENLCE_VPDFAIL

Indicates that the device driver was unable to read the vital product data (VPD) from the adapter. The device will not be available after this error is detected.

ERRID_ENLCE_PARITY

Indicates that the adapter has reported a parity error.

ERRID_ENLCE_DMAFAIL

Indicates that the adapter has reported a DMA error.

ERRID_ENLCE_NOMEM

Indicates that not enough memory was available to complete the current operation.

ERRID_ENLCE_NOMBUFS

Indicates that no mbufs were available for a receive packet. The packet will be dropped.

ERRID_ENLCE_PIOFAIL

Indicates that the device driver has detected a PIO failure. The device will not be available after this error is detected.

The Error IDs for the Gigabit Ethernet-SX PCI Adapter (14100401) are as follows:

ERRID_GXENT_ADAP_ERR

Indicates that the adapter failed initialization commands. User intervention is necessary to fix the problem.

ERRID_GXENT_CMD_ERR

Indicates that the device driver has detected an error while issuing commands to the adapter. The device driver will enter an adapter recovery mode where it will attempt to recover from the error. If the device driver is successful, it will log ERRID_GXENT_RCVRY_EXIT. User intervention is not necessary for this error unless the problem persists.

ERRID_GXENT_DOWNLOAD_ERR

Indicates that an error occurred while downloading firmware to the adapter. User intervention is necessary to fix the problem.

ERRID_GXENT_EEPROM_ERR

Indicates that an error occurred while reading the adapter EEPROM. User intervention is necessary to fix the problem.

ERRID_GXENT_LINK_DOWN

Indicates that the link between the adapter and the network switch is down. The device driver will attempt to reestablish the connection once the physical link is reestablished. When the link is again established, the device driver will log ERRID_GXENT_RCVRY_EXIT. User intervention is necessary to fix the problem.

ERRID_GXENT_RCVRY_EXIT

Indicates that a temporary error (link down, command error, or transmission error) has been corrected.

ERRID_GXENT_TX_ERR

Indicates that the device driver has detected a transmission error. The device driver will enter an adapter recovery mode in an attempt to recover from the error. If the device driver is successful, it will log ERRID_GXENT_RCVRY_EXIT. User intervention is not necessary for this error unless the problem persists.

For more information, see “Error Logging” on page 546.

Chapter 8. Graphic Input Devices Subsystem

The graphic input devices subsystem includes the keyboard/sound, mouse, tablet, dials, and lighted programmable-function keys (LPFK) devices. These devices provide operator input primarily to graphic applications. However, the keyboard can provide system input by means of the console.

open and close Subroutines

An open subroutine call is used to create a channel between the caller and a graphic input device driver. The keyboard supports two such channels. The most recently created channel is considered the active channel. All other graphic input device drivers support only one channel. The open subroutine call is processed normally, except that the OFLAG and MODE parameters are ignored. The keyboard provides support for the `fp_open` subroutine call; however, only one kernel mode channel may be open at any given time. The `fp_open` subroutine call returns EACCES for all other graphic input devices.

The close subroutine is used to remove a channel created by the open subroutine call.

read and write Subroutines

The graphic input device drivers do not support read or write operations. A read or write to a graphic input device special file behaves as if a read or write was made to `/dev/null`.

ioctl Subroutines

The ioctl operations provide run-time services. The special files support the following ioctl operations:

Keyboard

IOCINFO	Returns the devinfo structure.
KSQUERYID	Queries the keyboard device identifier.
KSQUERYSV	Queries the keyboard service vector.
KSREGRING	Registers the input ring.
KSRFLUSH	Flushes the input ring.
KSLED	Sets and resets the keyboard LEDs.
KSCFGCLICK	Configures the clicker.
KSVOLUME	Sets the alarm volume.
KSALARM	Sounds the alarm.
KSTRATE	Sets the typematic rate.
KSTDELAY	Sets the typematic delay.
KSKAP	Enables and disables the keep-alive poll.
KSKAPACK	Acknowledges the keep-alive poll.
KSDIAGMODE	Enables and disables the diagnostics mode.

Notes:

1. A nonactive channel processes only **IOCINFO**, **KSQUERYID**, **KSQUERYSV**, **KSREGRING**, **KSRFLUSH**, **KSKAP**, and **KSKAPACK**. All other ioctl subroutine calls are ignored without error.
2. The **KSLED**, **KSCFGCLICK**, **KSVOLUME**, **KSALARM**, **KSTRATE**, and **KSTDELAY** ioctl subroutine calls return an **EBUSY** error in the **errno** global variable when the keyboard is in diagnostics mode.
3. The **KSQUERYSV** ioctl subroutine call is only available when the channel is open from kernel mode (with the **fp_open** kernel service).
4. The **KSKAP**, **KSKAPACK**, **KSDIAGMODE** ioctl subroutine calls are only available when the channel is open from user mode.

Mouse

IOCINFO	Returns the devinfo structure.
MQUERYID	Queries the mouse device identifier.
MREGRING	Registers the input ring.
MRFLUSH	Flushes the input ring.
MTHRESHOLD	Sets the mouse reporting threshold.
MRESOLUTION	Sets the mouse resolution.
MSCALE	Sets the mouse scale.
MSAMPLERATE	Sets the mouse sample rate.

Tablet

IOCINFO	Returns the devinfo structure.
TABQUERYID	Queries the tablet device identifier.
TABREGRING	Registers the input ring.
TABFLUSH	Flushes the input ring.
TABCONVERSION	Sets the tablet conversion mode.
TABRESOLUTION	Sets the tablet resolution.
TABORIGIN	Sets the tablet origin.
TABSAMPLERATE	Sets the tablet sample rate.
TABDEADZONE	Sets the tablet dead zones.

GIO (Graphics I/O) Adapter

IOCINFO	Returns the devinfo structure.
GIOQUERYID	Returns the ID of the attached devices.

Dials

IOCINFO	Returns the devinfo structure.
DIALREGRING	Registers the input ring.
DIALRFLUSH	Flushes the input ring.
DIALSETGRAND	Sets the dial granularity.

LPFK

IOCINFO	Returns the devinfo structure.
LPFKREGRING	Registers the input ring.

LPFKRFLUSH Flushes the input ring.
LPFKLIGHT Sets and resets the key lights.

Input Ring

Data is obtained from graphic input devices via a circular First-In First-Out (FIFO) queue or input ring, rather than with a **read** subroutine call. The memory address of the input ring is registered with an **ioctl** (or **fp_ioctl**) subroutine call. The program that registers the input ring is the owner of the ring and is responsible for allocating, initializing, and freeing the storage associated with the ring. The same input ring can be shared by multiple devices.

The input ring consists of the input ring header followed by the reporting area. The input ring header contains the reporting area size, the head pointer, the tail pointer, the overflow flag, and the notification type flag. Before registering an input ring, the ring owner must ensure that the head and tail pointers contain the starting address of the reporting area. The overflow flag must also be cleared and the size field set equal to the number of bytes in the reporting area. After the input ring has been registered, the owner can modify only the head pointer and the notification type flag.

Data stored on the input ring is structured as one or more event reports. Event reports are placed at the tail of the ring by the graphic input device drivers. Previously queued event reports are taken from the head of the input ring by the owner of the ring. The input ring is empty when the head and tail locations are the same. An overflow condition exists if placement of an event on the input ring would overwrite data that has not been processed. Following an overflow, new event reports are not placed on the input ring until the input ring is flushed via an **ioctl** subroutine or service vector call.

The owner of the input ring is notified when an event is available for processing via a **SIGMSG** signal or via callback if the channel was created by an **fp_open** subroutine call. The notification type flag in the input ring header specifies whether the owner should be notified each time an event is placed on the ring or only when an event is placed on an empty ring.

Management of Multiple Keyboard Input Rings

When multiple keyboard channels are opened, keyboard events are placed on the input ring associated with the most recently opened channel. When this channel is closed, the alternate channel is activated and keyboard events are placed on the input ring associated with that channel.

Event Report Formats

Each event report consists of an identifier followed by the report size in bytes, a time stamp (system time in milliseconds), and one or more bytes of device-dependent data. The value of the identifier is specified when the input ring is registered. The program requesting the input-ring registration is responsible for identifier uniqueness within the input-ring scope.

Note: Event report structures are placed on the input-ring without spacing. Data wraps from the end to the beginning of the reporting area. A report can be split on any byte boundary into two non-contiguous sections.

The event reports are as follows:

Keyboard

ID	Specifies the report identifier.
Length	Specifies the report length.
Time stamp	Specifies the system time (in milliseconds).
Key position code	Specifies the key position code.
Key scan code	Specifies the key scan code.
Status flags	Specifies the status flags.

Tablet

ID	Specifies the report identifier.
Length	Specifies the report length.
Time stamp	Specifies the system time (in milliseconds).
Absolute X	Specifies the absolute X coordinate.
Absolute Y	Specifies the absolute Y coordinate.

LPFK

ID	Specifies the report identifier.
Length	Specifies the report length.
Time stamp	Specifies the system time (in milliseconds).
Number of key pressed	Specifies the number of the key pressed.

Dials

ID	Specifies the report identifier.
Length	Specifies the report length.
Time stamp	Specifies the system time (in milliseconds).
Number of dial changed	Specifies the number of the dial changed.
Delta change	Specifies delta dial rotation.

Mouse

ID	Specifies the report identifier.
Length	Specifies the report length.
Time stamp	Specifies the system time (in milliseconds).
Delta X	Specifies the delta mouse motion along the X axis.
Delta Y	Specifies the delta mouse motion along the Y axis.
Button status	Specifies the button status.

Keyboard Service Vector

The keyboard service vector provides a limited set of keyboard-related and sound-related services for kernel extensions. The following services are available:

- Sound alarm
- Enable and disable secure attention key (SAK)
- Flush input queue

The address of the service vector is obtained with the `fp_ioctl` subroutine call during a non-critical period. The kernel extension can later invoke the service using an indirect call as follows:

```
(*ServiceVector[ServiceNumber]) (dev_t DeviceNumber, caddr_t Arg);
```


where:

- The service vector is a pointer to the service vector obtained by the **KSQUERYSV** `fp_ioctl` subroutine call.
- The *ServiceNumber* parameter is defined in the **inputdd.h** file.
- The *DeviceNumber* parameter specifies the major and minor numbers of the keyboard.
- The *Arg* parameter points to a **ksalarm** structure for alarm requests and a **uint** variable for SAK enable and disable requests. The *Arg* parameter is NULL for flush queue requests.

If successful, the function returns a value of 0 is returned. Otherwise, the function returns an error number defined in the **errno.h** file. Flush-queue and enable/disable-SAK requests are always processed, but alarm requests are ignored if the kernel extension's channel is inactive.

The following example uses the service vector to sound the alarm:

```
/* pinned data structures */
/* This example assumes that pinning is done elsewhere. */
int (**ksvtbl) ();
struct ksalarm alarm;
dev_t devno;
/* get address of service vector */
/* This should be done in a noncritical section */
if (fp_ioctl(fp, KSQUERYSV, &ksvtbl, 0)) {
    /* error recovery */
}
.
.
.
/* critical section */
/* sound alarm for 1 second using service vector */
alarm.duration = 128;
alarm.frequency = 100;
if ((*ksvtbl[KSVALARMS]) (devno, &alarm)) {
    /* error recovery */
}
```

Special Keyboard Sequences

Special keyboard sequences are provided for the Secure Attention Key (SAK) and the Keep Alive Poll (KAP).

Secure Attention Key

The user requests a secure shell by keying a secure attention. The keyboard driver interprets the key sequence CTRL x r as the SAK. An indirect call using the keyboard service vector enables and disables the detection of this key sequence. If detection of the SAK is enabled, a SAK causes the SAK callback to be invoked. The SAK callback is invoked even if the input ring is inactive due to a user process issuing an open to the keyboard special file. The SAK callback runs within the interrupt environment.

Keep Alive Poll

The keyboard device driver supports a special key sequence that kills the process which owns the keyboard. This sequence must first be defined with a **KSKAP** `ioctl` operation. After this sequence is defined, the keyboard device driver sends a **SIGKAP** signal to the process which owns the keyboard when the special sequence is entered on the keyboard. The process which owns the keyboard must acknowledge the **KSKAP** signal with a **KSKAPACK** `ioctl` within 30 seconds or the

keyboard driver will terminate the process with a **SIGKILL** signal. The KAP is enabled on a per-channel basis and is unavailable if the channel is owned by a kernel extension.

Chapter 9. Low Function Terminal Subsystem

This chapter discusses the following topics:

- “Low Function Terminal Interface Functional Description”
- “Components Affected by the Low Function Terminal Interface” on page 186
- “Accented Characters” on page 188

The low function terminal (lft) interface is a pseudo device driver that interfaces with device drivers for the system keyboard and display adapters. Beginning with AIX Version 4.1, the lft interface adheres to all standard requirements for pseudo device drivers and has all the entry points and configuration code as required by the AIX Version 4.1 (or later) device driver architecture. This section gives a high-level description of the various configuration methods and entry points provided by the lft interface.

All the device drivers controlled by the lft interface are also used by AIXwindows. Consequently, along with the functions required for the tty subsystem interface, the lft interface provides the functions required by AIXwindows interfaces with display device driver adapters.

Low Function Terminal Interface Functional Description

Configuration

The lft interface uses the common define, undefine, and unconfig methods standard for most devices.

Note: The lft interface does not support any change method for dynamically changing the lft configuration. Instead, use the **-P** flag with the **chdev** command. The changes become effective the next time the lft interface is configured.

The configuration process for the lft opens all display device drivers. To define the default display and console, select the default display and console during the console configuration process. If a graphics display is chosen as the system console, it automatically becomes the default display. The lft interface displays text on the default display.

The configuration process for the lft interface queries the ODM database for the available fonts and software keyboard map for the current session.

Terminal Emulation

The lft interface is a stream-based tty subsystem. The lft interface provides VT100 (or IBM 3151) terminal emulation for the standard part of the ANSI 3.64 data stream. All line discipline handling is performed in the layers above the lft interface. The lft interface does not support virtual terminals.

The lft interface supports multiple fonts to handle the different screen sizes and resolutions necessary in providing a 25x80 character display on various display adapters.

Note: Applications requiring hft extensions need to use aixterm.

IOCTLS Needed for AIXwindow Support

AIXwindows and the lft interface share the system keyboard and display device drivers. To prevent screen and keyboard inconsistencies, a set of ioctls coordinates usage between AIXwindows and the lft interface. On a system with multiple displays, the lft interface can still use the default display as long as AIXwindows is using another display.

Note: The lft interface provides ioctl support to set and change the default display.

Low Function Terminal to System Keyboard Interface

The lft interface with the system keyboard uses an input ring mechanism. The details of the keyboard driver ioctls, as well as the format and description of this input ring, are provided in the “Chapter 8. Graphic Input Devices Subsystem” on page 179. The keyboard device driver passes raw keystrokes to the lft interface. These keystrokes are converted to the appropriate code point using keyboard tables. The use of keyboard-language-dependent keyboard tables ensures that the lft interface provides National Language Support.

Note: The keystroke conversion and the keyboard tables are the same used by the hft interface in AIX Version 3.

Low Function Terminal to Display Device Driver Interface

The lft uses a device independent interface known as the virtual display driver (vdd) interface. Because the lft interface has no virtual terminal or monitor mode support, some of the vdd entry points are not used by the lft.

The display drivers might enqueue font request through the font process started during lft initialization. The font process pins and unpins the requested fonts for DMA to the display adapter.

Low Function Terminal Device Driver Entry Points

The lft interface supports the open, close, read, write, ioctl, and configuration entry points.

Components Affected by the Low Function Terminal Interface

Configuration User Commands

The lft interface is a pseudo device driver. Consequently, the system configuration process does not detect the lft interface as it does an adapter. The system provides for pseudo device drivers to be started through **Config_Rules**. To start the lft interface, use the **startlft** program.

Supported commands include:

- **lsfont**
- **mkfont**

- **chfont**
- **lskbd**
- **chkbd**
- **lsdisp** (see note)
- **chdisp** (see note)

Notes:

1. *lsdisp* outputs the logical device name instead of the instance number.
2. *chdisp* uses the *ioctl* interface to the *lft* to set the requested display.

Display Device Driver

Beginning with AIX Version 4.1, a display device driver is required for each supported display adapter.

The display device drivers provide all the standard interfaces (such as *config*, *initialize*, *terminate*, and so forth) required in any AIX Version 4.1 (or later) device drivers. The only device switch table entries supported are *open*, *close*, *config*, and *ioctl*. All other device switch table entries are set to *nodev*. In addition, the display device drivers provide a set of *ioctls* for use by AIXwindows and diagnostics to perform device specific functions such as get bus access, bus memory address, DMA operations, and so forth.

Rendering Context Manager

The Rendering Context Manager (RCM) is a loadable module.

Note: Previously, the *hft* interface provided AIXwindows with the ***gsc_handle***. This handle is used in all of the ***aixgsc*** system calls. The RCM provides this service for the *lft* interface.

To ensure that *lft* can recover the display in case AIXwindows should terminate abnormally, AIXwindows issues the *ioctl* to RCM after opening the pseudo device. RCM passes on the *ioctl* to the *lft*. This way, the *close* function in RCM is invoked (Because AIXwindows is the only application that has opened RCM), and RCM notifies the *lft* interface to start reusing the display. To support this communication, the RCM provides the required *ioctl* support.

The RCM to lft Interface Initialization:

1. RCM performs the *open /dev/lft*.
2. Upon receiving a list of displays from X, RCM passes the information to the *lft* through an *ioctl*.
3. RCM resets the adapter.

If AIXwindows terminates abnormally:

1. RCM receives notification from X about the displays it was using.
2. RCM resets the adapter.
3. RCM passes the information to the *lft* via an *ioctl*.

The AIXwindows to lft Initialization includes:

1. AIXwindows opens */dev/rcm*.
2. AIXwindows gets the ***gsc_handle*** from RCM via an *ioctl*.

3. AIXwindows becomes a graphics process aixgsc (MAKE_GP, ...)
4. AIXwindows, through an ioctl, informs RCM about the displays it wishes to use.
5. AIXwindows opens all of the input devices it needs and passes the same input ring to each of them.

Upon normal termination:

1. X issues a close to all of the input devices it opened.
2. X informs RCM, through an ioctl, about the displays it was using.

Diagnostics

Diagnostics and other applications that require access to the graphics adapter use the AIXwindows to lft interface.

Accented Characters

Here are the valid sets of characters for each of the diacritics that the Low Function Terminal (LFT) subsystem uses to validate the two-key nonspacing character sequence.

List of Diacritics Supported by the HFT LFT Subsystem

There are seven diacritic characters for which sets of characters are provided:

- Acute (188)
- Grave (188)
- Circumflex (188)
- Umlaut (188)
- Tilde (188)
- Overcircle (188)
- Cedilla (188)

Valid Sets of Characters (Categorized by Diacritics)

Acute Function	Code Value
Acute accent	0xef
Apostrophe (acute)	0x27
e Acute small	0x82
e Acute capital	0x90
a Acute small	0xa0
i Acute small	0xa1
o Acute small	0xa2
u Acute small	0xa3
a Acute capital	0xb5
i Acute capital	0xd6
y Acute small	0xec
y Acute capital	0xed
o Acute capital	0xe0
u Acute capital	0xe9

Grave Function	Code Value
Grave accent	0x60

Grave Function	Code Value
a Grave small	0x85
e Grave small	0x8a
i Grave small	0x8d
o Grave small	0x95
u Grave small	0x97
a Grave capital	0xb7
e Grave capital	0xd4
i Grave capital	0xde
o Grave capital	0xe3
u Grave capital	0xeb

Circumflex Function	Code Value
^ Circumflex accent	0x5e
a Circumflex small	0x83
e Circumflex small	0x88
i Circumflex small	0x8c
o Circumflex small	0x93
u Circumflex small	0x96
a Circumflex capital	0xb6
e Circumflex capital	0xd2
i Circumflex capital	0xd7
o Circumflex capital	0xe2
u Circumflex capital	0xea

Umlaut Function	Code Value
Umlaut accent	0xf9
u Umlaut small	0x81
a Umlaut small	0x84
e Umlaut small	0x89
i Umlaut small	0x8b
a Umlaut capital	0x8e
O Umlaut capital	0x99
u Umlaut capital	0x9a
e Umlaut capital	0xd3
i Umlaut capital	0xd8

Tilde Function	Code Value
Tilde accent	0x7e
n Tilde small	0xa4
n Tilde capital	0xa5
a Tilde small	0xc6
a Tilde capital	0xc7
o Tilde small	0xe4
o Tilde capital	0xe5
Overcircle Function	Code Value
Overcircle accent	0x7d
a Overcircle small	0x86
a Overcircle capital	0x8f
Cedilla Function	Code Value
Cedilla accent	0xf7
c Cedilla capital	0x80
c Cedilla small	0x87

Chapter 10. Logical Volume Subsystem

Logical volume subsystem provides flexible access and control for complex physical storage systems.

The following topics describe how the logical volume device driver (LVDD) interacts with physical volumes:

- Logical Volume Subsystem
 - “Direct Access Storage Devices (DASDs)”
 - “Physical Volumes”
- Understanding the Logical Volume Device Driver
 - “Interface to Physical Disk Device Drivers” on page 198
- “Understanding Logical Volumes and Bad Blocks” on page 199
- “Changing the mwcc_entries Variable” on page 200

Direct Access Storage Devices (DASDs)

Direct access storage devices (DASDs) are *fixed* or *removable* storage devices. Typically, these devices are hard disks. A fixed storage device is any storage device defined during system configuration to be an integral part of the system DASD. The operating system detects an error if a fixed storage device is not available at some time during normal operation.

A removable storage device is any storage device defined by the person who administers your system during system configuration to be an optional part of the system DASD. The removable storage device can be removed from the system at any time during normal operation. As long as the device is logically unmounted first, the operating system does not detect an error.

The following types of devices are not considered DASD and are not supported by the logical volume manager (LVM):

- Diskettes
- CD-ROM (compact disk read-only memory)
- WORM (write-once read-many)

For a description of the DASD device block level, see “DASD Device Block Level Description” on page 299.

Physical Volumes

A logical volume is a portion of a physical volume viewed by the system as a volume. Logical records are records defined in terms of the information they contain rather than physical attributes.

A physical volume is a DASD structured for requests at the physical level, that is, the level at which a processing unit can request device-independent operations on a physical block address basis. A physical volume is composed of the following:

- A device-dependent reserved area
- A variable number of physical blocks that serve as DASD descriptors

- An integral number of partitions, each containing a fixed number of physical blocks

When performing I/O at a physical level, no bad-block relocation is supported. Bad blocks are not hidden at this level as they are at the logical level (see “Understanding Logical Volumes and Bad Blocks” on page 199). Typical operations at the physical level are `read-physical-block` and `write-physical-block`.

The following are terms used when discussing DASD volumes:

block	A contiguous, 512-byte region of a physical volume that corresponds in size to a DASD sector
partition	A set of blocks (with sequential cylinder, head, and sector numbers) contained within a single physical volume

The number of blocks in a partition, as well as the number of partitions in a given physical volume, are fixed when the physical volume is installed in a volume group. Every physical volume in a volume group has exactly the same partition size. There is no restriction on the types of DASDs (for example, Small Computer Systems Interface (SCSI), Enhanced Small Device Interface (ESDI), or IPI) that can be placed in a given volume group.

Note: A given physical volume must be assigned to a volume group before that physical volume can be used by the LVM.

Physical Volume Implementation Limitations

When composing a physical volume from a DASD, the following implementation restrictions apply to DASD characteristics:

- 1 to 32 physical volumes per volume group
- The partition size is restricted to 2^n bytes, for $20 \leq n \leq 30$
- The physical block size is restricted to 512 bytes

Physical Volume Layout

A physical volume consists of a logically contiguous string of physical sectors. Sectors are numbered 0 through the last physical sector number (LPSN) on the physical volume. The total number of physical sectors on a physical volume is $LPSN + 1$. The actual physical location and physical order of the sectors are transparent to the sector numbering scheme.

Note: Sector numbering applies to user-accessible data sectors only. Spare sectors and Customer-Engineer (CE) sectors are not included. CE sectors are reserved for use by diagnostic test routines or microcode.

Reserved Sectors on a Physical Volume

A physical volume reserves the first 128 sectors to store various types of DASD configuration and operation information. The `/usr/include/sys/hd_psn.h` file describes the information stored on the reserved sectors. The locations of the items in the reserved area are expressed as physical sector numbers in this file, and the lengths of those items are in number of sectors.

The 128-sector reserved area of a physical volume includes a boot record, the bad-block directory, the LVM record, and the mirror write consistency (MWC) record. The boot record consists of one sector containing information that allows the read-only system (ROS) to boot the system. A description of the boot record can be found in the `/usr/include/sys/bootrecord.h` file.

The boot record also contains the `pv_id` field. This field is a 64-bit number uniquely identifying a physical volume. This identifier is assigned by the manufacturer of the physical volume. However, if a physical volume is part of a volume group, the `pv_id` field may be assigned by the LVM.

The bad-block directory records the blocks on the physical volume that have been diagnosed as unusable (see “Understanding Logical Volumes and Bad Blocks” on page 199). The structure of the bad-block directory and its entries can be found in the `/usr/include/sys/bbdir.h` file.

The LVM record consists of one sector and contains information used by the LVM when the physical volume is a member of the volume group. The LVM record is described in the `/usr/include/lvmrec.h` file.

The MWC record consists of one sector. It identifies which logical partitions may be inconsistent if the system is not shut down properly. When the volume group is varied back online for use, this information is used to make logical partitions consistent again.

Sectors Reserved for the Logical Volume Manager (LVM)

If a physical volume is part of a volume group, the physical volume is used by the LVM and contains two additional reserved areas. One area contains the volume group descriptor area/volume group status area and follows the first 128 reserved sectors. The other area is at the end of the physical volume reserved as a relocation pool for bad blocks that must be software-relocated. Both of these areas are described by the LVM record. The space between these last two reserved areas is divided into equal-sized partitions.

The volume group descriptor area (VGDA) is divided into the following:

- The volume group header. This header contains general information about the volume group and a time stamp used to verify the consistency of the VGDA.
- A list of logical volume entries. The logical volume entries describe the states and policies of logical volumes. This list defines the maximum number of logical volumes allowed in the volume group. The maximum is specified when a volume group is created.
- A list of physical volume entries. The size of the physical volume list is variable because the number of entries in the partition map can vary for each physical volume. For example, a 200 MB physical volume with a partition size of 1 MB has 200 partition map entries.
- A name list. This list contains the special file names of each logical volume in the volume group.
- A volume group trailer. This trailer contains an ending time stamp for the volume group descriptor area.

When a volume group is varied online, a majority (also called a quorum) of VGDA's must be present to perform recovery operations unless you have specified the **force** flag. (The vary-on operation, performed by using the **varyonvg** command, makes a volume group available to the system.) See “Logical Volume

Storage Overview" in *AIX Version 4.3 System Management Guide: Operating System and Devices* for introductory information about the vary-on process and quorums.

Note: Use of the **force** flag can result in data inconsistency.

A volume group with only one physical volume must contain two copies of the physical volume group descriptor area. For any volume group containing more than one physical volume, there are at least three on-disk copies of the volume group descriptor area. The default placement of these areas on the physical volume is as follows:

- For the first physical volume installed in a volume group, two copies of the volume group descriptor area are placed on the physical volume.
- For the second physical volume installed in a volume group, one copy of the volume group descriptor area is placed on the physical volume.
- For the third physical volume installed in a volume group, one copy of the volume group descriptor area is placed on the physical volume. The second copy is removed from the first volume.
- For additional physical volumes installed in a volume group, one copy of the volume group descriptor area is placed on the physical volume.

When a vary-on operation is performed, a majority of copies of the volume group descriptor area must be able to come online before the vary-on operation is considered successful. A quorum ensures that at least one copy of the volume group descriptor areas available to perform recovery was also one of the volume group descriptor areas that were online during the previous vary-off operation. If not, the consistency of the volume group descriptor area cannot be ensured.

The volume group status area (VGSA) contains the status of all physical volumes in the volume group. This status is limited to active or missing. The VGSA also contains the state of all allocated physical partitions (PP) on all physical volumes in the volume group. This state is limited to active or stale. A PP with a stale state is not used to satisfy a read request and is not updated on a write request.

A PP changes from active to stale after a successful resynchronization of the logical partition (LP) that has multiple copies, or mirrors, and is no longer consistent with its peers in the LP. This inconsistency can be caused by a write error or by not having a physical volume available when the LP is written to or updated.

A PP changes from stale to active after a successful resynchronization of the LP. A resynchronization operation issues resynchronization requests starting at the beginning of the LP and proceeding sequentially through its end. The LVDD reads from an active partition in the LP and then writes that data to any stale partition in the LP. When the entire LP has been traversed, the partition state is changed from stale to active.

Normal I/O can occur concurrently in an LP that is being resynchronized.

Note: If a write error occurs in a stale partition while a resynchronization is in progress, that partition remains stale.

If all stale partitions in an LP encounter write errors, the resynchronization operation is ended for this LP and must be restarted from the beginning.

The vary-on operation uses the information in the VGSA to initialize the LVDD data structures when the volume group is brought online.

Understanding the Logical Volume Device Driver

The Logical Volume Device Driver (LVDD) is a pseudo-device driver that operates on logical volumes through the `/dev/lvn` special file. Like the physical disk device driver, this pseudo-device driver provides character and block entry points with compatible arguments. Each volume group has an entry in the kernel device switch table. Each entry contains entry points for the device driver and a pointer to the volume group data structure. The logical volumes of a volume group are distinguished by their minor numbers.

Note: Each logical volume has a control block located in the first 512 bytes. Data begins in the second 512-byte block. Care must be taken when reading and writing directly to the logical volume, such as when using applications that write to raw logical volumes, because the control block is not protected from such writes. If the control block is overwritten, commands that use it can no longer be used.

Character I/O requests are performed by issuing a read or write request on a `/dev/rlvn` character special file for a logical volume. The read or write is processed by the file system SVC handler, which calls the LVDD `ddread` or `ddwrite` entry point. The `ddread` or `ddwrite` entry point transforms the character request into a block request. This is done by building a buffer for the request and calling the LVDD `ddstrategy` entry point.

Block I/O requests are performed by issuing a read or write on a block special file `/dev/lvn` for a logical volume. These requests go through the SVC handler to the `bread` or `bwrite` block I/O kernel services. These services build buffers for the request and call the LVDD `ddstrategy` entry point. The LVDD `ddstrategy` entry point then translates the logical address to a physical address (handling bad block relocation and mirroring) and calls the appropriate physical disk device driver.

On completion of the I/O, the physical disk device driver calls the `iodone` kernel service on the device interrupt level. This service then calls the LVDD I/O completion-handling routine. Once this is completed, the LVDD calls the `iodone` service again to notify the requester that the I/O is completed.

The LVDD is logically split into top and bottom halves. The top half contains the `ddopen`, `ddclose`, `ddread`, `ddwrite`, `ddioctl`, and `ddconfig` entry points. The bottom half contains the `ddstrategy` entry point, which contains block read and write code. This is done to isolate the code that must run fully pined and has no access to user process context. The bottom half of the device driver runs on interrupt levels and is not permitted to page fault. The top half runs in the context of a process address space and can page fault.

Data Structures

The interface to the `ddstrategy` entry point is one or more logical `buf` structures in a list. The logical `buf` structure is defined in the `/usr/include/sys/buf.h` file and contains all needed information about an I/O request, including a pointer to the data buffer. The `ddstrategy` entry point associates one or more (if mirrored) physical `buf` structures (or `pbufs`) with each logical `buf` structure and passes them to the appropriate physical device driver.

The **pbuf** structure is a standard **buf** structure with some additional fields. The LVDD uses these fields to track the status of the physical requests that correspond to each logical I/O request. A pool of pinned **pbuf** structures is allocated and managed by the LVDD.

There is one device switch entry for each volume group defined on the system. Each volume group entry contains a pointer to the volume group data structure describing it.

Top Half of LVDD

The top half of the LVDD contains the code that runs in the context of a process address space and can page fault. It contains the following entry points:

ddopen	Called by the file system when a logical volume is mounted, to open the logical volume specified.
ddclose	Called by the file system when a logical volume is unmounted, to close the logical volume specified.
ddconfig	Initializes data structures for the LVDD.
ddread	Called by the read subroutine to translate character I/O requests to block I/O requests. This entry point verifies that the request is on a 512-byte boundary and is a multiple of 512 bytes in length.

When a character request spans partitions or logical tracks (32 pages of 4K bytes each), the LVDD **ddread** routine breaks it into multiple requests. The routine then builds a buffer for each request and passes it to the LVDD **ddstrategy** entry point, which handles logical block I/O requests.

If the *ext* parameter is set (called by the **readx** subroutine), the **ddread** entry point passes this parameter to the LVDD **ddstrategy** routine in the *b_options* field of the buffer header.

ddwrite	Called by the write subroutine to translate character I/O requests to block I/O requests. The LVDD ddwrite routine performs the same processing for a write request as the LVDD ddread routine does for read requests.
----------------	---

ddioctl	Supports the following operations:
----------------	------------------------------------

CACLNUP

Causes the mirror write consistency (MWC) cache to be written to all physical volumes (PVs) in a volume group.

IOCINFO, XLATE, GETVGS

Return LVM configuration information and PP status information.

LV_INFO

Provides information about a logical volume. This ioctl operation is available beginning with AIX Version 4.2.1.

PBUFCNT

Increases the number of physical buffer headers (pbufs) in the LVM pbuf pool.

Bottom Half of the LVDD

The bottom half of the device driver supports the **ddstrategy** entry point. This entry point processes all logical block requests and performs the following functions:

- Validates I/O requests.

- Checks requests for conflicts (such as overlapping block ranges) with requests currently in progress.
- Translates logical addresses to physical addresses.
- Handles mirroring and bad-block relocation.

The bottom half of the LVDD runs on interrupt levels and, as a result, is not permitted to page fault. The bottom half of the LVDD is divided into the following three layers:

- “Strategy Layer”
- “Scheduler Layer”
- “Physical Layer”

Each logical I/O request passes down through the bottom three layers before reaching the physical disk device driver. Once the I/O is complete, the request returns back up through the layers to handle the I/O completion processing at each layer. Finally, control returns to the original requestor.

Strategy Layer

The strategy layer deals only with logical requests. The **ddstrategy** entry point is called with one or more logical **buf** structures. A list of **buf** structures for requests that are not blocked are passed to the second layer, the scheduler.

Scheduler Layer

The scheduler layer schedules physical requests for logical operations and handles mirroring and the MWC cache. For each logical request the scheduler layer schedules one or more physical requests. These requests involve translating logical addresses to physical addresses, handling mirroring, and calling the LVDD physical layer with a list of physical requests.

When a physical I/O operation is complete for one phase or mirror of a logical request, the scheduler initiates the next phase (if there is one). If no more I/O operations are required for the request, the scheduler calls the strategy termination routine. This routine notifies the originator that the request has been completed.

The scheduler also handles the MWC cache for the volume group. If a logical volume is using mirror write consistency, then requests for this logical volume are held within the scheduling layer until the MWC cache blocks can be updated on the target physical volumes. When the MWC cache blocks have been updated, the request proceeds with the physical data write operations.

When MWC is being used, system performance can be adversely affected. This is caused by the overhead of logging or journaling that a write request is active in a logical track group (LTG) (32 4K-byte pages or 128K bytes). This overhead is for mirrored writes only. It is necessary to guarantee data consistency between mirrors particularly if the system crashes before the write to all mirrors has been completed.

Mirror write consistency can be turned off for an entire logical volume. It can also be inhibited on a request basis by turning on the **NO_MWC** flag as defined in the **/usr/include/sys/lvdd.h** file.

Physical Layer

The physical layer of the LVDD handles startup and termination of the physical request. The physical layer calls a physical disk device driver’s **ddstrategy** entry

point with a list of **buf** structures linked together. In turn, the physical layer is called by the **iodone** kernel service when each physical request is completed.

This layer also performs bad-block relocation and detection/correction of bad blocks, when necessary. These details are hidden from the other two layers.

Interface to Physical Disk Device Drivers

Physical disk device drivers adhere to the following criteria if they are to be accessed by the LVDD:

- Disk block size must be 512 bytes.
- The physical disk device driver needs to accept a list of requests defined by **buf** structures, which are linked together by the **av_forw** field in each **buf** structure.
- For unrecoverable media errors, physical disk device drivers need to set the following:
 - The **B_ERROR** flag must be set to on (defined in the **/usr/include/sys/buf.h** file) in the **b_flags** field.
 - The **b_error** field must be set to **E_MEDIA** (defined in the **/usr/include/sys/errno.h** file).
 - The **b_resid** field must be set to the number of bytes in the request that were not read or written successfully. The **b_resid** field is used to determine the block in error.

Note: For write requests, the LVDD attempts to hardware-relocate the bad block. If this is unsuccessful, then the block is software-relocated. For read requests, the information is recorded and the block is relocated on the next write request to that block.

- For a successful request that generated an excessive number of retries, the device driver can return good data. To indicate this situation it must set the following:
 - The **b_error** field is set to **ESOFT**; this is defined in the **/usr/include/sys/errno.h** file.
 - The **b_flags** field has the **B_ERROR** flag set to on.
 - The **b_resid** field is set to a count indicating the first block in the request that had excessive retries. This block is then relocated.
- The physical disk device driver needs to accept a request of one block with **HWRELOC** (defined in the **/usr/include/sys/lvdd.h** file) set to on in the **b_options** field. This indicates that the device driver is to perform a hardware relocation on this request. If the device driver does not support hardware relocation the following should be set:
 - The **b_error** field is set to **EIO**; this is defined in the **/usr/include/sys/errno.h** file.
 - The **b_flags** field has the **B_ERROR** flag set on.
 - The **b_resid** field is set to a count indicating the first block in the request that has excessive retries.
- The physical disk device driver should support the system dump interface as defined.
- The physical disk device driver must support write verification on an I/O request. Requests for write verification are made by setting the **b_options** field to **WRITEV**. This value is defined in the **/usr/include/sys/lvdd.h** file.

Understanding Logical Volumes and Bad Blocks

The physical layer of the LVDD initiates all bad-block processing and isolates all of the decision making from the physical disk device driver. This happens so the physical disk device driver does not need to handle mirroring, which is the duplication of data transparent to the user. (See “Physical Layer” on page 197.)

Relocating Bad Blocks

The physical layer of the logical volume device driver (LVDD) checks each physical request to see if there are any known software-relocated bad blocks in the request. The LVDD determines if a request contains known software-relocated bad blocks by hashing the physical address. Then a hash chain of the LVDD defects directory is searched to see if any bad-block entries are in the address range of the request.

If bad blocks exist in a physical request, the request is split into pieces. The first piece contains any blocks up to the relocated block. The second piece contains the relocated block (the relocated address is specified in the bad-block entry) of the defects directory. The third piece contains any blocks after the relocated block to the end of the request or to the next relocated block. These separate pieces are processed sequentially until the entire request has been satisfied.

Once the I/O for the first of the separate pieces has completed, the **iodone** kernel service calls the LVDD physical layer’s termination routine (specified in the **b_done** field of the **buf** structure). The termination routine initiates I/O for the second piece of the original request (containing the relocated block), and then for the third piece. When the entire physical operation is completed, the appropriate scheduler’s policy routine (in the second layer of the LVDD) is called to start the next phase of the logical operation.

Detecting and Correcting Bad Blocks

If a logical volume is mirrored, a newly detected bad block is fixed by relocating that block. A good mirror is read and then the block is relocated using data from the good mirror. With mirroring, the user does not need to know when bad blocks are found. However, the physical disk device driver does log permanent I/O errors so the user can determine the rate of media surface errors.

When a bad block is detected during I/O, the physical disk device driver sets the error fields in the **buf** structure to indicate that there was a media surface error. The physical layer of the LVDD then initiates any bad-block processing that must be done.

If the operation was a nonmirrored read, the block is not relocated because the data in the relocated block is not initialized until a write is performed to the block. To support this delayed relocation, an entry for the bad block is put into the LVDD defects directory and into the bad-block directory on disk. These entries contain no relocated block address and the status for the block is set to indicate that relocation is desired.

On each I/O request, the physical layer checks whether there are any bad blocks in the request. If the request is a write and contains a block that is in a relocation-desired state, the request is sent to the physical disk device driver with safe hardware relocation requested. If the request is a read, a read of the known defective block is attempted.

If the operation was a read operation in a mirrored LP, a request to read one of the other mirrors is initiated. If the second read is successful, then the read is turned into a write request and the physical disk device driver is called with safe hardware relocation specified to fix the bad mirror.

If the hardware relocation fails or the device does not support safe hardware relocation, the physical layer of the LVDD attempts software relocation. At the end of each volume is a reserved area used by the LVDD as a pool of relocation blocks. When a bad block is detected and the disk device driver is unable to relocate the block, the LVDD picks the next unused block in the relocation pool and writes to this new location. A new entry is added to the LVDD defects directory in memory (and to the bad-block directory on disk) that maps the bad-block address to the new relocation block address. Any subsequent I/O requests to the bad-block address are routed to the relocation address.

Attention: Formatting a fixed disk deletes any data that may be on the disk. Format a fixed disk only when absolutely necessary and preferably after backing up all data on the disk.

If you need to format a fixed disk completely (including reinitializing any bad blocks), use the formatting function supplied by the **diag** command. (The **diag** command typically, but not necessarily, writes over all data on a fixed disk. Refer to the documentation that comes with the fixed disk to determine the effect of formatting with the **diag** command.)

Changing the `mwcc_entries` Variable

The default for the number of the logical volume manager mirror write consistency cache (MWCC) is 62, or `0x3e` is the hexadecimal. This number is double the original default and improves the user's write performance, but it also increases the time needed to make all mirrors consistent again at volume-group vary-on time after a crash. These variables are all system load-dependent.

Note: This procedure modifies the LVM device driver binary code using the **adb** command. Care should be taken when following this procedure.

Prerequisite Tasks or Conditions

- You must have root user authority.

Procedure

1. Change to the `/usr/lib/drivers` directory.
2. At the command line, type:

```
dump -h hd_pin
```

In the **.data** section header is the **RAWptr** file, which contains a hex address. Record this address to be used later. An example hex address is `0x0000fc00`.

3. At the command line, type:

```
dump -n hd_pin | grep mwcc_entries
```

The second field displayed is the offset for the variable. An example is `0x000003f8`.

4. Add the hex address found in the **RAWptr** file to the offset for the variable to get the address of the **mwcc_entries** variable. For example:

$0x0000fc00 + 0x000003f8 = 0x0000fff8$

5. Make a copy of the **hd_pin** file by typing the following at the command line:

```
cp hd_pin hd_pin.orig
```

6. Use the **adb** command to modify the **hd_pin** file binary by typing the following at the command line:

```
abd -w hd_pin
```

Note: The **adb** command issues a warning that the string table is missing or the object is being stripped.

7. Issue the following command in response to the **adb** command to verify you have the correct address:

```
0xADDR/X
```

where ADDR is the address you generated in step 4.

If the **hd_pin** file has not been modified in this way, the **adb** command responds with:

```
ADDR: 3e
```

If this procedure has been done, the **adb** command responds with:

```
ADDR: zz
```

where zz is the current value, from 0x1 to 0x3e, for the number of MWCC entries. If the value is not between 0x1 and 0x3e, check that you are using the correct address.

8. Modify the address to the value you want for the number of MWCC entries by typing the following at the command line:

```
0xADDR/W zz
```

where ADDR is the address derived in step 4 and zz is a hex number between 0x1 and 0x3e.

9. Exit the **adb** command by using the Ctrl-D key sequence.
10. Rebuild the startup logical volume by typing the following at the command line:

```
bosboot -a
```

11. Shut down the system by typing the following at the command line:

```
shutdown -F
```

12. Restart the system.

The system runs with the size of the mirror write consistency cache set to the new value.

Note: The new **mwcc_entries** value must be from 0x1 to 0x3e, inclusive. Unpredictable results occur if these bounds are violated.

Chapter 11. Printer Addition Management Subsystem

If you are configuring a printer for your system, there are basically two types of printers: printers already supported by the operating system and new printer types. "Printer Support" in *AIX Version 4.3 Guide to Printers and Printing* lists printers that are already supported.

Printer Types Currently Supported

To configure a supported type of printer, you need only to run the **mkvirprt** command to create a customized printer file for your printer. This customized printer file, which is in the `/var/spool/lpd/pio/@local/custom` directory, describes the specific parameters for your printer. For more information see "Configuring a Printer without Adding a Queue" in *AIX Version 4.3 Guide to Printers and Printing*.

Printer Types Currently Unsupported

To configure a currently unsupported type of printer, you must develop and add a Predefined printer definition for your printer. This new option is then entered in the list of available choices when the user selects a printer to configure for the system. The actual data used by the printer subsystem comes from the Customized printer definition created by the **mkvirprt** command.

"Adding a New Printer Type to Your System" provides general instructions for adding an undefined printer. To add an undefined printer, you modify an existing printer definition. Undefined printers fall into two categories:

- Printers that closely emulate a supported printer. You can use SMIT or the virtual printer commands to make the changes you need.
- Printers that do not emulate a supported printer or that emulate several data streams. It is simpler to make the necessary changes for these printers by editing the printer colon file. See "Adding a Printer Using the Printer Colon File" in *AIX Version 4.3 Guide to Printers and Printing*.

"Adding an Unsupported Device to the System" on page 92 offers an overview of the major steps required to add an unsupported device of any type to your system.

Adding a New Printer Type to Your System

To add an unsupported printer to your system, you must add a new Printer definition to the printer directories. For more complicated scenarios, you might also need to add a new printer-specific formatter to the printer backend.

"Example of Print Formatter" in *AIX Version 4.3 Guide to Printers and Printing* shows how the print formatter interacts with the printer formatter subroutines.

Additional Steps for Adding a New Printer Type

However, if you want the new Printer definition to carry the name of the new printer, you must develop a new Predefined definition to carry the new printer information besides adding a new Printer definition. Use the **piopredef** command to do this.

Steps for adding a new printer-specific formatter to the printer backend are discussed in "Adding a Printer Formatter to the Printer Backend" on page 205. "Example of Print Formatter" in *AIX Version 4.3 Guide to Printers and Printing* shows how print formatters can interact with the printer formatter subroutines.

Note: These instructions apply to the addition of a new printer definition to the system, not to the addition of a physical printer device itself. For information on adding a new printer device, refer to device configuration and management. If your new printer requires an interface other than the parallel or serial interface provided by the operating system, you must also provide a new device driver.

If the printer being added does not emulate a supported printer or if it emulates several data streams, you need to make more changes to the Printer definition. It is simpler to make the necessary changes for these printers by editing the printer colon file. See "Adding a Printer Using the Printer Colon File" in *AIX Version 4.3 Guide to Printers and Printing*.

Modifying Printer Attributes

Edit the customized file (`/var/spool/lpd/pio/custom` `/var/spool/lpd/pio/@local/custom` `QueueName:QueueDeviceName`), adding or changing the printer attributes to match the new printer.

For example, assume that you created a new file based on the existing 4201-3 printer. The customized file for the 4201-3 printer contains the following template that the printer formatter uses to initialize the printer:

```
%I[ez,em,eA,cv,eC,e0,cp,cc, . . .
```

The formatter fills in the string as directed by this template and sends the resulting sequence of commands to the 4201-3 printer. Specifically, this generates a string of escape sequences that initialize the printer and set such parameters as vertical and horizontal spacing and page length. You would construct a similar command string to properly initialize the new printer and put it into 4201-emulation mode. While many of the escape sequences might be the same, at least one will be different: the escape sequence that is the command to put the printer into the specific printer-emulation mode. Assume that you added an **ep** attribute that specifies the string to initialize the printer to 4201-3 emulation mode, as follows:

```
\033\012\013
```

The Printer Initialization field will then be:

```
%I[ep,ez,em,eA,cv,eC,e0,cp,cc, . . .
```

You must create a virtual printer for each printer-emulation mode you want to use. See "Real and Virtual Printers" in *AIX Version 4.3 Guide to Printers and Printing*.

Adding a Printer Definition

To add a new printer to the system, you must first create a description of the printer by adding a new printer definition to the printer definition directories.

Typically, to add a new printer definition to the database, you first modify an existing printer definition and then create a customized printer definition in the Customized Printer Directory.

Once you have added the new customized printer definition to the directory, the **mkvirprt** command uses it to present the new printer as a choice for printer addition and selection. Since the new printer definition is a customized printer definition, it appears in the list of printers under the name of the original printer from which it was customized.

A totally new printer must be added as a predefined printer definition in the `/usr/lib/lpd/pio/predef` directory. If the user chooses to work with printers once this new predefined printer definition is added to the Predefined Printer Directory, the **mkvirprt** command can then list all the printers in that directory. The added printer appears on the list of printers given to the user as if it had been supported all along. Specific information about this printer can then be extended, added, modified, or deleted, as necessary.

"Printer Support" in *AIX Version 4.3 Guide to Printers and Printing* lists the supported printer types and names of representative printers.

Adding a Printer Formatter to the Printer Backend

If your new printer's data stream differs significantly from one of the numerous printer data streams currently handled by the operating system, you must define a new backend formatter. Adding a new formatter does not require the addition of a new backend. Instead, all you typically need are modifications to the formatter commands associated with that printer under the supervision of the existing printer backend. If a new backend is required, see "Printer Backend Overview for Programming" in *AIX Version 4.3 Guide to Printers and Printing*.

Understanding Embedded References in Printer Attribute Strings

The attribute string retrieved by the **piocmdout**, **piogetstr**, and **piogetvals** subroutines can contain embedded references to other attribute strings or integers. The attribute string can also contain embedded logic that dynamically determines the content of the constructed string. This allows the constructed string to reflect the state of the formatter environment when one of these subroutines is called.

Embedded references and logic are defined with escape sequences that are placed at appropriate locations in the attribute string. The first character of each escape sequence is always the % character. This character indicates the beginning of an escape sequence. The second character (and sometimes subsequent characters) define the operation to be performed. The remainder of the characters (if any) in the escape sequence are operands to be used in performing the specified operation.

The escape sequences that can be specified in an attribute string are based on the **terminfo** parameterized string escape sequences for terminals. These escape sequences have been modified and extended for printers.

The attribute names that can be referenced by attribute strings are:

- The names of all attribute variables (which can be integer or string variables) defined to the **piogetvals** subroutine. When references are made to these variables, the **piogetvals**-defined versions are the values used.
- All other attributes names in the database. These attributes are considered string constants.

Any attribute value (integer variable, string variable, or string constant) can be referenced by any attribute string. Consequently, it is important that the formatter ensures that the values for all the integer variables and string variables defined to the **piogetvals** subroutine are kept current.

The formatter must not assume that the particular attribute string whose name it specifies to the **piogetstr** or **piocmdout** subroutine does not reference certain variables. The attribute string is retrieved from the database that is external to the formatter. The values in the database represented by the string can be changed to reference additional variables without the formatter's knowledge.

Chapter 12. Small Computer System Interface Subsystem

This overview describes the interface between a small computer system interface (SCSI) device driver and a SCSI adapter device driver. It is directed toward those wishing to design and write a SCSI device driver that interfaces with an existing SCSI adapter device driver. It is also meant for those wishing to design and write a SCSI adapter device driver that interfaces with existing SCSI device drivers.

SCSI Subsystem Overview

The main topics covered in this overview are:

- “Responsibilities of the SCSI Adapter Device Driver”
- “Responsibilities of the SCSI Device Driver”
- “Initiator-Mode Support” on page 208
- “Target-Mode Support” on page 208

This section frequently refers to both a *SCSI device driver* and a *SCSI adapter device driver*. These two distinct device drivers work together in a layered approach to support attachment of a range of SCSI devices. The SCSI adapter device driver is the *lower* device driver of the pair, and the SCSI device driver is the *upper* device driver.

Responsibilities of the SCSI Adapter Device Driver

The SCSI adapter device driver (the lower layer) is the software interface to the system hardware. This hardware includes the SCSI bus hardware plus any other system I/O hardware required to run an I/O request. The SCSI adapter device driver hides the details of the I/O hardware from the SCSI device driver. The design of the software interface allows a user with limited knowledge of the system hardware to write the upper device driver.

The SCSI adapter device driver manages the SCSI bus but not the SCSI devices. It can send and receive SCSI commands, but it cannot interpret the contents of the command. The lower driver also provides recovery and logging for errors related to the SCSI bus and system I/O hardware. Management of the device specifics is left to the SCSI device driver. The interface of the two drivers allows the upper driver to communicate with different SCSI bus adapters without requiring special code paths for each adapter.

Responsibilities of the SCSI Device Driver

The SCSI device driver (the upper layer) provides the rest of the operating system with the software interface to a given SCSI device or device class. The upper layer recognizes which SCSI commands are required to control a particular SCSI device or device class. The SCSI device driver builds I/O requests containing device SCSI commands and sends them to the SCSI adapter device driver in the sequence needed to operate the device successfully. The SCSI device driver cannot manage adapter resources or give the SCSI command to the adapter. Specifics about the adapter and system hardware are left to the lower layer.

The SCSI device driver also provides recovery and logging for errors related to the SCSI device it controls.

The operating system provides several kernel services allowing the SCSI device driver to communicate with SCSI adapter device driver entry points without having the actual name or address of those entry points. The description contained in “Logical File System Kernel Services” on page 51 can provide more information.

Communication between SCSI Devices

When two SCSI devices communicate, one assumes the initiator-mode role, and the other assumes the target-mode role. The initiator-mode device generates the SCSI command, which requests an operation, and the target-mode device receives the SCSI command and acts. It is possible for a SCSI device to perform both roles simultaneously.

When writing a new SCSI adapter device driver, the writer must know which mode or modes must be supported to meet the requirements of the SCSI adapter and any interfaced SCSI device drivers. When a SCSI adapter device driver is added so that a new SCSI adapter works with all existing SCSI device drivers, both initiator-mode and target-mode must be supported in the SCSI adapter device driver.

Initiator-Mode Support

The interface between the SCSI device driver and the SCSI adapter device driver for initiator-mode support (that is, the attached device acts as a target) is accessed through calls to the SCSI adapter device driver **open**, **close**, **ioctl**, and **strategy** routines. I/O requests are queued to the SCSI adapter device driver through calls to its strategy entry point.

Communication between the SCSI device driver and the SCSI adapter device driver for a particular initiator I/O request is made through the **sc_buf** structure (see “Understanding the sc_buf Structure” on page 218), which is passed to and from the strategy routine in the same way a standard driver uses a **struct buf** structure.

Target-Mode Support

The interface between the SCSI device driver and the SCSI adapter device driver for target-mode support (that is, the attached device acts as an initiator) is accessed through calls to the SCSI adapter device driver **open**, **close**, and **ioctl** subroutines. Buffers that contain data received from an attached initiator device are passed from the SCSI adapter device driver to the SCSI device driver, and back again, in **tm_buf** structures.

Communication between the SCSI adapter device driver and the SCSI device driver for a particular data transfer is made by passing the **tm_buf** structures by pointer directly to routines whose entry points have been previously registered. This registration occurs as part of the sequence of commands the SCSI device driver executes using calls to the SCSI adapter device driver when the device driver opens a target-mode device instance.

Understanding SCSI Asynchronous Event Handling

Note: This operation is not supported by all SCSI I/O controllers.

A SCSI device driver can register a particular device instance for receiving asynchronous event status by calling the **SCIOEVENT** ioctl operation for the SCSI-adapter device driver. When an event covered by the **SCIOEVENT** ioctl operation is detected by the SCSI adapter device driver, it builds an **sc_event_info** structure and passes a pointer to the structure and to the asynchronous event-handler routine entry point, which was previously registered. The fields in the structure are filled in by the SCSI adapter device driver as follows:

id	For initiator mode, this is set to the SCSI ID of the attached SCSI target device. For target mode, this is set to the SCSI ID of the attached SCSI initiator device.
lun	For initiator mode, this is set to the SCSI LUN of the attached SCSI target device. For target mode, this is set to 0).
mode	Identifies whether the initiator or target mode device is being reported. The following values are possible: SC_IM_MODE An initiator mode device is being reported. SC_TM_MODE A target mode device is being reported.
events	This field is set to indicate what event or events are being reported. The following values are possible, as defined in the /usr/include/sys/scsi.h file: SC_FATAL_HDW_ERR A fatal adapter hardware error occurred. SC_ADAP_CMD_FAILED An unrecoverable adapter command failure occurred. SC_SCSI_RESET_EVENT A SCSI bus reset was detected. SC_BUFS_EXHAUSTED In target-mode, a maximum buffer usage event has occurred.
adap_devno	This field is set to indicate the device major and minor numbers of the adapter on which the device is located.
async_correlator	This field is set to the value passed to the SCSI adapter device driver in the sc_event_struct structure. The SCSI device driver may optionally use this field to provide an efficient means of associating event status with the device instance it goes with. Alternatively, the SCSI device driver would use the combination of the id , lun , mode , and adap_devno fields to identify the device instance.

Note: Reserved fields should be set to 0 by the SCSI adapter device driver.

The information reported in the **sc_event_info.events** field does not queue to the SCSI device driver, but is instead reported as one or more flags as they occur. Since the data does not queue, the SCSI adapter device driver writer can use a single **sc_event_info** structure and pass it one at a time, by pointer, to each asynchronous event handler routine for the appropriate device instance. After determining for

which device the events are being reported, the SCSI device driver must copy the `sc_event_info.events` field into local space and must not modify the contents of the rest of the `sc_event_info` structure.

Since the event status is optional, the SCSI device driver writer determines what action is necessary to take upon receiving event status. The writer may decide to save the status and report it back to the calling application, or the SCSI device driver or application level program can take error recovery actions.

Defined Events and Recovery Actions

The adapter fatal hardware failure event is intended to indicate that no further commands to or from this SCSI device are likely to succeed, since the adapter it is attached to has failed. It is recommended that the application end the session with the device.

The unrecoverable adapter command failure event is not necessarily a fatal condition, but it can indicate that the adapter is not functioning properly. Possible actions by the application program include:

- Ending of the session with the device in the near future.
- Ending of the session after multiple (two or more) such events.
- Attempt to continue the session indefinitely.

The SCSI Bus Reset detection event is mainly intended as information only, but may be used by the application to perform further actions, if necessary.

The maximum buffer usage detected event only applies to a given target-mode device; it will not be reported for an initiator-mode device. This event indicates to the application that this particular target-mode device instance has filled its maximum allotted buffer space. The application should perform **read** system calls fast enough to prevent this condition. If this event occurs, data is not lost, but it is delayed to prevent further buffer usage. Data reception will be restored when the application empties enough buffers to continue reasonable operations. The **num_bufs** attribute may need to be increased to help minimize this problem. Also, it is possible that regardless of the number of buffers, the application simply is not processing received data fast enough. This may require some fine tuning of the application's data processing routines.

Asynchronous Event-Handling Routine

The SCSI-device driver asynchronous event-handling routine is typically called directly from the hardware interrupt-handling routine for the SCSI adapter device driver. The SCSI device driver writer must be aware of how this affects the design of the SCSI device driver.

Since the event handling routine is running on the hardware interrupt level, the SCSI device driver must be careful to limit operations in that routine. Processing should be kept to a minimum. In particular, if any error recovery actions are performed, it is recommended that the event-handling routine set state or status flags only and allow a process level routine to perform the actual operations.

The SCSI device driver must be careful to disable interrupts at the correct level in places where the SCSI device driver's lower execution priority routines manipulate variables that are also modified by the event-handling routine. To allow the SCSI device driver to disable at the correct level, the SCSI adapter device driver writer

must provide a configuration database attribute that defines the interrupt class, or priority, it runs on. This attribute must be named **intr_priority** so that the SCSI device driver configuration method knows which attribute of the parent adapter to query. The SCSI device driver configuration method should then pass this interrupt priority value to the SCSI device driver along with other configuration data for the device instance.

The SCSI device driver writer must follow any other general system rules for writing a routine which must execute in an interrupt environment. For example, the routine must not attempt to sleep or wait on I/O operations. It can perform wakeups to allow the process level to handle those operations.

Since the SCSI device driver copies the information from the **sc_event_info.events** field on each call to its asynchronous event-handling routine, there is no resource to free or any information which must be passed back later to the SCSI adapter device driver.

SCSI Error Recovery

The SCSI error-recovery process handles different issues depending on whether the SCSI device is in initiator mode or target mode. If the device is in initiator mode, the error-recovery process varies depending on whether or not the device is supporting command queuing.

SCSI Initiator-Mode Recovery When Not Command Tag Queuing

If an error such as a check condition or hardware failure occurs, transactions queued within the SCSI adapter device driver are terminated abnormally with **iodone** calls. The transaction active during the error is returned with the **sc_buf.bufstruct.b_error** field set to **EIO**. Other transactions in the queue are returned with the **sc_buf.bufstruct.b_error** field set to **ENXIO**. The SCSI device driver should process or recover the condition, rerunning any mode selects or device reservations to recover from this condition properly. After this recovery, it should reschedule the transaction that had the error. In many cases, the SCSI device driver only needs to retry the unsuccessful operation.

The SCSI adapter device driver should never retry a SCSI command on error after the command has successfully been given to the adapter. The consequences for retrying a SCSI command at this point range from minimal to catastrophic, depending upon the type of device. Commands for certain devices cannot be retried immediately after a failure (for example, tapes and other sequential access devices). If such an error occurs, the failed command returns an appropriate error status with an **iodone** call to the SCSI device driver for error recovery. Only the SCSI device driver that originally issued the command knows if the command can be retried on the device. The SCSI adapter device driver must only retry commands that were never successfully transferred to the adapter. In this case, if retries are successful, the **sc_buf** status should not reflect an error. However, the SCSI adapter device driver should perform error logging on the retried condition.

The first transaction passed to the SCSI adapter device driver during error recovery must include a special flag. This **SC_RESUME** flag in the **sc_buf.flags** field must be set to inform the SCSI adapter device driver that the SCSI device driver has recognized the fatal error and is beginning recovery operations. Any transactions passed to the SCSI adapter device driver, after the fatal error occurs

and before the **SC_RESUME** transaction is issued, should be flushed; that is, returned with an error type of **ENXIO** through an **iodone** call.

Note: If a SCSI device driver continues to pass transactions to the SCSI adapter device driver after the SCSI adapter device driver has flushed the queue, these transactions are also flushed with an error return of **ENXIO** through the **iodone** service. This gives the SCSI device driver a positive indication of all transactions flushed.

If the SCSI device driver is executing a gathered write operation, the error-recovery information mentioned previously is still valid, but the caller must restore the contents of the `sc_buf.resvdl` field and the `uio` struct that the field pointed to before attempting the retry (see “Gathered Write Commands” on page 217). The retry must occur from the SCSI device driver’s process level; it cannot be performed from the caller’s **iodone** subroutine. Also, additional return codes of **EFAULT** and **ENOMEM** are possible in the `sc_buf.bufstruct.b_error` field for a gathered write operation.

SCSI Initiator-Mode Recovery During Command Tag Queuing

If the SCSI device driver is queuing multiple transactions to the device and either a check condition error or a command terminated error occurs, the SCSI adapter driver does not clear all transactions in its queues for the device. It returns the failed transaction to the SCSI device driver with an indication that the queue for this device is not cleared by setting the **SC_DID_NOT_CLEAR_Q** flag in the `sc_buf.adap_q_status` field. The SCSI adapter driver halts the queue for this device awaiting error recovery notification from the SCSI device driver. The SCSI device driver then has three options to recover from this error:

- Send one error recovery command (request sense) to the device.
- Clear the SCSI adapter driver’s queue for this device.
- Resume the SCSI adapter driver’s queue for this device.

When the SCSI adapter driver’s queue is halted, the SCSI device driver can get sense data from a device by setting the **SC_RESUME** flag in the `sc_buf.flags` field and the **SC_NO_Q** flag in `sc_buf.q_tag_msg` field of the request-sense `sc_buf`. This action notifies the SCSI adapter driver that this is an error-recovery transaction and should be sent to the device while the remainder of the queue for the device remains halted. When the request sense completes, the SCSI device driver needs to either clear or resume the SCSI adapter driver’s queue for this device.

The SCSI device driver can notify the SCSI adapter driver to clear its halted queue by sending a transaction with the **SC_Q_CLR** flag in the `sc_buf.flags` field. This transaction must not contain a SCSI command because it is cleared from the SCSI adapter driver’s queue without being sent to the adapter. However, this transaction must have the SCSI ID field (`sc_buf.scsi_command.scsi_id`) and the LUN fields (`sc_buf.scsi_command.scsi_cmd.lun` and `sc_buf.lun`) filled in with the device’s SCSI ID and logical unit number (LUN). If addressing LUNs 8 - 31, the `sc_buf.lun` field should be set to the logical unit number and the `sc_buf.scsi_command.scsi_cmd.lun` field should be zeroed out. See the descriptions of these fields for further explanation. Upon receiving an **SC_Q_CLR** transaction, the SCSI adapter driver flushes all transactions for this device and sets their `sc_buf.bufstruct.b_error` fields to **ENXIO**. The SCSI device driver must wait until the `sc_buf` with the **SC_Q_CLR** flag set is returned before it resumes issuing transactions. The first transaction sent by the SCSI device driver after it receives the returned **SC_Q_CLR** transaction must have the **SC_RESUME** flag set in the `sc_buf.flags` fields.

If the SCSI device driver wants the SCSI adapter driver to resume its halted queue, it must send a transaction with the **SC_Q_RESUME** flag set in the `sc_buf.flags` field. This transaction can contain an actual SCSI command, but it is not required. However, this transaction must have the `sc_buf.scsi_command.scsi_id`, `sc_buf.scsi_command.scsi_cmd.lun`, and the `sc_buf.lun` fields filled in with the device's SCSI ID and logical unit number. See the description of these fields for further details. If this is the first transaction issued by the SCSI device driver after receiving the error (indicating that the adapter driver's queue is halted), then the **SC_RESUME** flag must be set as well as the **SC_Q_RESUME** flag.

Analyzing Returned Status

The following order of precedence should be followed by SCSI device drivers when analyzing the returned status:

1. If the `sc_buf.bufstruct.b_flags` field has the **B_ERROR** flag set, then an error has occurred and the `sc_buf.bufstruct.b_error` field contains a valid **errno** value.

If the `b_error` field contains the **ENXIO** value, either the command needs to be restarted or it was canceled at the request of the SCSI device driver.

If the `b_error` field contains the **EIO** value, then either one or no flag is set in the `sc_buf.status_validity` field. If a flag is set, an error in either the `scsi_status` or `general_card_status` field is the cause.

If the `status_validity` field is 0, then the `sc_buf.bufstruct.b_resid` field should be examined to see if the SCSI command issued was in error. The `b_resid` field can have a value without an error having occurred. To decide whether an error has occurred, the SCSI device driver must evaluate this field with regard to the SCSI command being sent and the SCSI device being driven.

If the SCSI device driver is queuing multiple transactions to the device and if either **SC_CHECK_CONDITION** or **SC_COMMAND_TERMINATED** is set in `scsi_status`, then the value of `sc_buf.adap_q_status` must be analyzed to determine if the adapter driver has cleared its queue for this device. If the SCSI adapter driver has not cleared its queue after an error, then it holds that queue in a halted state.

If `sc_buf.adap_q_status` is set to 0, the SCSI adapter driver has cleared its queue for this device and any transactions outstanding are flushed back to the SCSI device driver with an error of **ENXIO**.

If the **SC_DID_NOT_CLEAR_Q** flag is set in the `sc_buf.adap_q_status` field, the adapter driver has not cleared its queue for this device. When this condition occurs, the SCSI adapter driver allows the SCSI device driver to send one error recovery transaction (request sense) that has the field `sc_buf.q_tag_msg` set to **SC_NO_Q** and the field `sc_buf.flags` set to **SC_RESUME**. The SCSI device driver can then notify the SCSI adapter driver to clear or resume its queue for the device by sending a **SC_Q CLR** or **SC_Q_RESUME** transaction.

If the SCSI device driver does not queue multiple transactions to the device (that is, the **SC_NO_Q** is set in `sc_buf.q_tag_msg`), then the SCSI adapter clears its queue on error and sets `sc_buf.adap_q_status` to 0.

2. If the `sc_buf.bufstruct.b_flags` field does not have the **B_ERROR** flag set, then no error is being reported. However, the SCSI device driver should examine the `b_resid` field to check for cases where less data was transferred than expected. For some SCSI commands, this occurrence may not represent an error. The SCSI device driver must determine if an error has occurred.

If a nonzero `b_resid` field does represent an error condition, then the device queue is not halted by the SCSI adapter device driver. It is possible for one or

more succeeding queued commands to be sent to the adapter (and possibly the device). Recovering from this situation is the responsibility of the SCSI device driver.

3. In any of the above cases, if `sc_buf.bufstruct.b_flags` field has the **B_ERROR** flag set, then the queue of the device in question has been halted. The first `sc_buf` structure sent to recover the error (or continue operations) must have the **SC_RESUME** bit set in the `sc_buf.flags` field.

Target-Mode Error Recovery

If an error occurs during the reception of `send` command data, the SCSI adapter device driver sets the **TM_ERROR** flag in the `tm_buf.user_flag` field. The SCSI adapter device driver also sets the **SC_ADAPTER_ERROR** bit in the `tm_buf.status_validity` field and sets a single flag in the `tm_buf.general_card_status` field to indicate the error that occurred.

In the SCSI subsystem, an error during a `send` command does not affect future target-mode data reception. Future `send` commands continue to be processed by the SCSI adapter device driver and queue up, as necessary, after the data with the error. The SCSI device driver continues processing the `send` command data, satisfying user read requests as usual except that the error status is returned for the appropriate user request. Any error recovery or synchronization procedures the user requires for a target-mode received-data error must be implemented in user-supplied software.

A Typical Initiator-Mode SCSI Driver Transaction Sequence

A simplified sequence of events for a transaction between a SCSI device driver and a SCSI adapter device driver follows. In this sequence, routine names preceded by a `dd_` are part of the SCSI device driver, while those preceded by a `sc_` are part of the SCSI adapter device driver.

1. The SCSI device driver receives a call to its `dd_strategy` routine; any required internal queuing occurs in this routine. The `dd_strategy` entry point then triggers the operation by calling the `dd_start` entry point. The `dd_start` routine invokes the `sc_strategy` entry point by calling the `devstrategy` kernel service with the relevant `sc_buf` structure as a parameter.
2. The `sc_strategy` entry point initially checks the `sc_buf` structure for validity. These checks include validating the `devno` field, matching the SCSI ID/LUN to internal tables for configuration purposes, and validating the request size.
3. Although the SCSI adapter device driver cannot reorder transactions, it does perform queue chaining. If no other transactions are pending for the requested device, the `sc_strategy` routine immediately calls the `sc_start` routine with the new transaction. If there are other transactions pending, the new transaction is added to the tail of the device chain.
4. At each interrupt, the `sc_intr` interrupt handler verifies the current status. The SCSI adapter device driver fills in the `sc_buf` `status_validity` field, updating the `scsi_status` and `general_card_status` fields as required. The SCSI adapter device driver also fills in the `bufstruct.b_resid` field with the number of bytes not transferred from the request. If all the data was transferred, the `b_resid` field is set to a value of 0. When a transaction completes, the `sc_intr` routine causes the `sc_buf` entry to be removed from the device queue and calls the `iodone` kernel service, passing the just dequeued `sc_buf` structure for the device as the parameter. The `sc_start` routine is then called again to process the next

transaction on the device queue. The `iodone` kernel service calls the SCSI device driver `dd_iodone` entry point, signaling the SCSI device driver that the particular transaction has completed.

5. The SCSI device driver `dd_iodone` routine investigates the I/O completion codes in the `sc_buf` status entries and performs error recovery, if required. If the operation completed correctly, the SCSI device driver dequeues the original buffer structures. It calls the `iodone` kernel service with the original buffer pointers to notify the originator of the request.

Understanding SCSI Device Driver Internal Commands

During initialization, error recovery, and open or close operations, SCSI device drivers initiate some transactions not directly related to an operating system request. These transactions are called *internal commands* and are relatively simple to handle.

Internal commands differ from operating system-initiated transactions in several ways. The primary difference is that the SCSI device driver is required to generate a `struct buf` that is not related to a specific request. Also, the actual SCSI commands are typically more control-oriented than data transfer-related.

The only special requirement for commands with short data-phase transfers (less than or equal to 256 bytes) is that the SCSI device driver must have pinned the memory being transferred into or out of system memory pages. However, due to system hardware considerations, additional precautions must be taken for data transfers into system memory pages when the transfers are larger than 256 bytes. The problem is that any system memory area with a DMA data operation in progress causes the entire memory page that contains it to become inaccessible.

As a result, a SCSI device driver that initiates an internal command with more than 256 bytes must have preallocated and pinned an area of some multiple whose size is the system page size. The driver must not place in this area any other data areas that it may need to access while I/O is being performed into or out of that page. Memory pages so allocated must be avoided by the device driver from the moment the transaction is passed to the adapter device driver until the device driver `iodone` routine is called for the transaction (and for any other transactions to those pages).

Understanding the Execution of Initiator I/O Requests

During normal processing, many transactions are queued in the SCSI device driver. As the SCSI device driver processes these transactions and passes them to the SCSI adapter device driver, the SCSI device driver moves them to the in-process queue. When the SCSI adapter device driver returns through the `iodone` service with one of these transactions, the SCSI device driver either recovers any errors on the transaction or returns using the `iodone` kernel service to the calling level.

The SCSI device driver can send only one `sc_buf` structure per call to the SCSI adapter device driver. Thus, the `sc_buf.bufstruct.av_forw` pointer should be null when given to the SCSI adapter device driver, which indicates that this is the only request. The SCSI device driver can queue multiple `sc_buf` requests by making multiple calls to the SCSI adapter device driver strategy routine.

Spanned (Consolidated) Commands

Some kernel operations may be composed of sequential operations to a device. For example, if consecutive blocks are written to disk, blocks may or may not be in physically consecutive buffer pool blocks.

To enhance SCSI bus performance, the SCSI device driver should consolidate multiple queued requests when possible into a single SCSI command. To allow the SCSI adapter device driver the ability to handle the scatter and gather operations required, the `sc_buf.bp` should always point to the first `buf` structure entry for the spanned transaction. A null-terminated list of additional `struct buf` entries should be chained from the first field through the `buf.av_forw` field to give the SCSI adapter device driver enough information to perform the DMA scatter and gather operations required. This information must include at least the buffer's starting address, length, and cross-memory descriptor.

The spanned requests should always be for requests in either the read or write direction but not both, since the SCSI adapter device driver must be given a single SCSI command to handle the requests. The spanned request should always consist of complete I/O requests (including the additional `struct buf` entries). The SCSI device driver should not attempt to use partial requests to reach the maximum transfer size.

The maximum transfer size is actually adapter-dependent. The `IOCINFO ioctl` operation can be used to discover the SCSI adapter device driver's maximum allowable transfer size. To ease the design, implementation, and testing of components that may need to interact with multiple SCSI-adapter device drivers, a required minimum size has been established that all SCSI adapter device drivers must be capable of supporting. The value of this minimum/maximum transfer size is defined as the following value in the `/usr/include/sys/scsi.h` file:

```
SC_MAXREQUEST      /* maximum transfer request for a single */  
                   /* SCSI command (in bytes) */
```

If a transfer size larger than the supported maximum is attempted, the SCSI adapter device driver returns a value of `EINVAL` in the `sc_buf.bufstruct.b_error` field.

Due to system hardware requirements, the SCSI device driver must consolidate only commands that are memory page-aligned at both their starting and ending addresses. Specifically, this applies to the consolidation of *inner* memory buffers. The ending address of the first buffer and the starting address of all subsequent buffers should be memory page-aligned. However, the starting address of the first memory buffer and the ending address of the last do not need to be aligned so.

The purpose of consolidating transactions is to decrease the number of SCSI commands and bus phases required to perform the required operation. The time required to maintain the simple chain of `buf` structure entries is significantly less than the overhead of multiple (even two) SCSI bus transactions.

Fragmented Commands

Single I/O requests larger than the maximum transfer size must be divided into smaller requests by the SCSI device driver. For calls to a SCSI device driver's character I/O (read/write) entry points, the `uphysio` kernel service can be used to break up these requests. For a *fragmented command* such as this, the `sc_buf.bp` field

should be null so that the SCSI adapter device driver uses only the information in the `sc_buf` structure to prepare for the DMA operation.

Gathered Write Commands

The gathered write commands facilitate communications applications that are required to send header and trailer messages with data buffers. These headers and trailers are typically the same or similar for each transfer. Therefore, there may be a single copy of these messages but multiple data buffers.

The gathered write commands, accessed through the `sc_buf.resvd1` field, differ from the spanned commands, accessed through the `sc_buf.bp` field, in several ways:

- Gathered write commands can transfer data regardless of address alignment, while spanned commands must be memory page-aligned in address and length, making small transfers difficult.
- Gathered write commands can be implemented either in software (which requires the extra step of copying the data to temporary buffers) or hardware. Spanned commands can be implemented in system hardware due to address-alignment requirements. As a result, spanned commands are potentially faster to run.
- Gathered write commands are not able to handle read requests. Spanned commands can handle both read and write requests.
- Gathered write commands can be initiated only on the process level, but spanned commands can be initiated on either the process or interrupt level.

To execute a gathered write command, the SCSI device driver must:

- Fill in the `resvd1` field with a pointer to the `uio` struct.
- Call the SCSI adapter device driver on the same process level with the `sc_buf` structure in question.
- Be attempting a write.
- Not have put a non-null value in the `sc_buf.bp` field.

If any of these conditions are not met, the gather write commands do not succeed and the `sc_buf.bufstruct.b_error` is set to `EINVAL`.

This interface allows the SCSI adapter device driver to perform the gathered write commands in both software or hardware as long as the adapter supports this capability. Because the gathered write commands can be performed in software (by using such kernel services as `uiomove`), the contents of the `resvd1` field and the `uio` struct can be altered. Therefore, the caller must restore the contents of both the `resvd1` field and the `uio` struct before attempting a retry. Also, the retry must occur from the process level; it must not be performed from the caller's `iodone` subroutine.

To support SCSI adapter device drivers that perform the gathered write commands in software, additional return values in the `sc_buf.bufstruct.b_error` field are possible when gathered write commands are unsuccessful.

ENOMEM Error due to lack of system memory to perform copy.

EFAULT Error due to memory copy problem.

Note: The gathered write command facility is optional for both the SCSI device driver and the SCSI adapter device driver. Attempting a gathered write command to a SCSI adapter device driver that does not support gathered write can cause a system crash. Therefore, any SCSI device driver must issue a **SCIOGTHW** ioctl operation to the SCSI adapter device driver before using gathered writes. A SCSI adapter device driver that supports gathered writes must support the **SCIOGTHW** ioctl as well. The ioctl returns a successful return code if gathered writes are supported. If the ioctl fails, the SCSI device driver must not attempt a gathered write. Typically, a SCSI device driver places the **SCIOGTHW** call in its open routine for device instances that it will send gathered writes to.

SCSI Command Tag Queuing

Note: This operation is not supported by all SCSI I/O controllers.

SCSI command tag queuing refers to queuing multiple commands to a SCSI device. Queuing to the SCSI device can improve performance because the device itself determines the most efficient way to order and process commands. SCSI devices that support command tag queuing can be divided into two classes: those that clear their queues on error and those that do not. Devices that do not clear their queues on error resume processing of queued commands when the error condition is cleared (typically by receiving the next command). Devices that do clear their queues flush all commands currently outstanding.

Command tag queuing requires the SCSI adapter, the SCSI device, the SCSI device driver, and the SCSI adapter driver to support this capability. For a SCSI device driver to queue multiple commands to a SCSI device (that supports command tag queuing), it must be able to provide at least one of the following values in the `sc_buf.q_tag_msg`: **SC_SIMPLE_Q**, **SC_HEAD_OF_Q**, or **SC_ORDERED_Q**. The SCSI disk device driver and SCSI adapter driver do support this capability. This implementation provides some queuing-specific changeable attributes for disks that can queue commands. With this information, the disk device driver attempts to queue to the disk, first by queuing commands to the adapter driver. The SCSI adapter driver then queues these commands to the adapter, providing that the adapter supports command tag queuing. If the SCSI adapter does not support command tag queuing, then the SCSI adapter driver sends only one command at a time to the SCSI adapter and so multiple commands are not queued to the SCSI disk.

Understanding the `sc_buf` Structure

The `sc_buf` structure is used for communication between the SCSI device driver and the SCSI adapter device driver during an initiator I/O request. This structure is passed to and from the strategy routine in the same way a standard driver uses a **struct buf** structure.

Fields in the `sc_buf` Structure

The `sc_buf` structure contains certain fields used to pass a SCSI command and associated parameters to the SCSI adapter device driver. Other fields within this structure are used to pass returned status back to the SCSI device driver. The `sc_buf` structure is defined in the `/usr/include/sys/scsi.h` file.

Fields in the `sc_buf` structure are used as follows:

1. Reserved fields should be set to a value of 0, except where noted.
2. The `bufstruct` field contains a copy of the standard `buf` buffer structure that documents the I/O request. Included in this structure, for example, are the buffer address, byte count, and transfer direction. The `b_work` field in the `buf` structure is reserved for use by the SCSI adapter device driver. The current definition of the `buf` structure is in the `/usr/include/sys/buf.h` include file.
3. The `bp` field points to the original buffer structure received by the SCSI Device Driver from the caller, if any. This can be a chain of entries in the case of spanned transfers (SCSI commands that transfer data from or to more than one system-memory buffer). A null pointer indicates a nonspanned transfer. The null value specifically tells the SCSI adapter device driver that all the information needed to perform the DMA data transfer is contained in the `bufstruct` fields of the `sc_buf` structure. If the `bp` field is set to a non-null value, the `sc_buf.resvd1` field must have a value of null, or else the operation is not allowed.
4. The `scsi_command` field, defined as a `scsi` structure, contains, for example, the SCSI ID, SCSI command length, SCSI command, and a flag variable:
 - a. The `scsi_length` field is the number of bytes in the actual SCSI command. This is normally 6, 10, or 12 (decimal).
 - b. The `scsi_id` field is the SCSI physical unit ID.
 - c. The `scsi_flags` field contains the following bit flags:

SC_NODISC Do not allow the target to disconnect during this command.

SC_ASYNC Do not allow the adapter to negotiate for synchronous transfer to the SCSI device.

During normal use, the `SC_NODISC` bit should not be set. Setting this bit allows a device executing commands to monopolize the SCSI bus. Sometimes it is desirable for a particular device to maintain control of the bus once it has successfully arbitrated for it; for instance, when this is the only device on the SCSI bus or the only device that will be in use. For performance reasons, it may not be desirable to go through SCSI selections again to save SCSI bus overhead on each command.

Also during normal use, the `SC_ASYNC` bit must not be set. It should be set only in cases where a previous command to the device ended in an unexpected SCSI bus free condition. This condition is noted as **SC_SCSI_BUS_FAULT** in the `general_card_status` field of the `sc_cmd` structure. Since other errors may also result in the **SC_SCSI_BUS_FAULT** flag being set, the `SC_ASYNC` bit should only be set on the last retry of the failed command.

- d. The `sc_cmd` structure contains the physical SCSI command block. The 6 to 12 bytes of a single SCSI command are stored in consecutive bytes, with the op code and logical unit identified individually. The `sc_cmd` structure contains the following fields:
 - The `scsi_op_code` field specifies the standard SCSI op code for this command.
 - The `lun` field specifies the standard SCSI logical unit for the physical SCSI device controller. Typically, there will be one LUN per controller (LUN=0, for example) for devices with imbedded controllers. Only the upper 3 bits of this field contain the actual LUN ID. If addressing LUN's 0 - 7, this `lun` field should always be filled in with the LUN value. When addressing LUN's 8 - 31, this `lun` field should be set to 0 and the LUN value should be placed into the `sc_buf.lun` field described in this section.

- The `scsi_bytes` field contains the remaining command-unique bytes of the SCSI command block. The actual number of bytes depends on the value in the `scsi_op_code` field.
- The `resvd1` field is set to a non-null value to indicate a request for a gathered write. A gathered write means the SCSI command conducts a system-to-device data transfer where multiple, noncontiguous system buffers contain the write data. This data is transferred in order as a single data transfer for the SCSI command in this `sc_buf` structure.

The contents of the `resvd1` field, if non-null, must be a pointer to the `uio` structure that is passed to the SCSI device driver. The SCSI adapter device driver treats the `resvd1` field as a pointer to a `uio` structure that accesses the `iovec` structures containing pointers to the data. There are no address-alignment restrictions on the data in the `iovec` structures. The only restriction is that the total transfer length of all the data must not exceed the maximum transfer length for the adapter device driver. The `sc_buf.bufstruct.b_un.b_addr` field, which normally contains the starting system-buffer address, is ignored and can be altered by the SCSI adapter device driver when the `sc_buf` is returned. The `sc_buf.bufstruct.b_bcount` field should be set by the caller to the total transfer length for the data.

5. The `timeout_value` field specifies the time-out limit (in seconds) to be used for completion of this command. A time-out value of 0 means no time-out is applied to this I/O request.
6. The `status_validity` field contains an output parameter that can have one of the following bit flags as a value:

`SC_SCSI_ERROR` The `scsi_status` field is valid.
`SC_ADAPTER_ERROR` The `general_card_status` field is valid.

7. The `scsi_status` field in the `sc_buf` structure is an output parameter that provides valid SCSI command completion status when its `status_validity` bit is nonzero. The `sc_buf.bufstruct.b_error` field should be set to `EIO` anytime the `scsi_status` field is valid. Typical status values include:

`SC_GOOD_STATUS` The target successfully completed the command.
`SC_CHECK_CONDITION` The target is reporting an error, exception, or other conditions.
`SC_BUSY_STATUS` The target is currently busy and cannot accept a command now.
`SC_RESERVATION_CONFLICT` The target is reserved by another initiator and cannot be accessed.
`SC_COMMAND_TERMINATED` The target terminated this command after receiving a terminate I/O process message from the SCSI adapter.
`SC_QUEUE_FULL` The target's command queue is full, so this command is returned.

8. The `general_card_status` field is an output parameter that is valid when its `status_validity` bit is nonzero. The `sc_buf.bufstruct.b_error` field should be set to `EIO` anytime the `general_card_status` field is valid. This field contains generic SCSI adapter card status. It is intentionally general in coverage so that it can report error status from any typical SCSI adapter.

If an error is detected during execution of a SCSI command, and the error prevented the SCSI command from actually being sent to the SCSI bus by the adapter, then the error should be processed or recovered, or both, by the SCSI adapter device driver.

If it is recovered successfully by the SCSI adapter device driver, the error is logged, as appropriate, but is not reflected in the **general_card_status** byte. If the error cannot be recovered by the SCSI adapter device driver, the appropriate **general_card_status** bit is set and the **sc_buf** structure is returned to the SCSI device driver for further processing.

If an error is detected after the command was actually sent to the SCSI device, then it should be processed or recovered, or both, by the SCSI device driver.

For error logging, the SCSI adapter device driver logs SCSI bus- and adapter-related conditions, while the SCSI device driver logs SCSI device-related errors. In the following description, a capital letter "A" after the error name indicates that the SCSI adapter device driver handles error logging. A capital letter "H" indicates that the SCSI device driver handles error logging.

Some of the following error conditions indicate a SCSI device failure. Others are SCSI bus- or adapter-related.

SC_HOST_IO_BUS_ERR (A)	The system I/O bus generated or detected an error during a DMA or Programmed I/O (PIO) transfer.
SC SCSI_BUS_FAULT (H)	The SCSI bus protocol or hardware was unsuccessful.
SC_CMD_TIMEOUT (H)	The command timed out before completion.
SC_NO_DEVICE_RESPONSE (H)	The target device did not respond to selection phase.
SC_ADAPTER_HDW_FAILURE (A)	The adapter indicated an onboard hardware failure.
SC_ADAPTER_SFW_FAILURE (A)	The adapter indicated microcode failure.
SC_FUSE_OR_TERMINAL_PWR (A)	The adapter indicated a blown terminator fuse or bad termination.
SC SCSI_BUS_RESET (A)	The adapter indicated the SCSI bus has been reset.

9. When the SCSI device driver queues multiple transactions to a device, the **adap_q_status** field indicates whether or not the SCSI adapter driver has cleared its queue for this device after an error has occurred (see "SCSI Command Tag Queuing" on page 218). The flag of **SC_DID_NOT_CLEAR_Q** indicates that the SCSI adapter driver has not cleared its queue for this device and that it is in a halted state (so none of the pending queued transactions are sent to the device).

10. The **lun** field provides addressability of up to 32 logical units (LUNs). This field specifies the standard SCSI LUN for the physical SCSI device controller. If addressing LUN's 0 - 7, both this **lun** field (**sc_buf.lun**) and the **lun** field located in the **scsi_command** structure (**sc_buf.scsi_command.scsi_cmd.lun**) should be set to the LUN value. If addressing LUN's 8 - 31, this **lun** field (**sc_buf.lun**) should be set to the LUN value and the **lun** field located in the **scsi_command** structure (**sc_buf.scsi_command.scsi_cmd.lun**) should be set to 0.

Logical Unit Numbers (LUNs)		
lun Fields	LUN 0 - 7	LUN 8 - 31
sc_buf.lun	<i>LUN Value</i>	<i>LUN Value</i>
sc_buf.scsi_command.scsi_cmd.lun	<i>LUN Value</i>	0

Note: *LUN value* is the current value of LUN.

11. The `q_tag_msg` field indicates if the SCSI adapter can attempt to queue this transaction to the device (see "SCSI Command Tag Queuing" on page 218). This information causes the SCSI adapter to fill in the Queue Tag Message Code of the queue tag message for a SCSI command. The following values are valid for this field:

SC_NO_Q	Specifies that the SCSI adapter does not send a queue tag message for this command, and so the device does not allow more than one SCSI command on its command queue. This value must be used for all commands sent to SCSI devices that do not support command tag queuing.
SC_SIMPLE_Q	Specifies placing this command in the device's command queue. The device determines the order that it executes commands in its queue. The SCSI-2 specification calls this value the "Simple Queue Tag Message."
SC_HEAD_OF_Q	Specifies placing this command first in the device's command queue. This command does not preempt an active command at the device, but it is executed before all other commands in the command queue. The SCSI-2 specification calls this value the "Head of Queue Tag Message."
SC_ORDERED_Q	Specifies placing this command in the device's command queue. The device processes these commands in the order that they are received. The SCSI-2 specification calls this value the "Ordered Queue Tag Message."

Note: Commands with the value of **SC_NO_Q** for the `q_tag_msg` field (except for request sense commands) should not be queued to a device whose queue contains a command with another value for `q_tag_msg`. If commands with the **SC_NO_Q** value (except for request sense) are sent to the device, then the SCSI device driver must make sure that no active commands are using different values for `q_tag_msg`. Similarly, the SCSI device driver must also make sure that a command with a `q_tag_msg` value of **SC_ORDERED_Q**, **SC_HEAD_Q**, or **SC_SIMPLE_Q** is not sent to a device that has a command with the `q_tag_msg` field of **SC_NO_Q**.

12. The `flags` field contains bit flags sent from the SCSI device driver to the SCSI adapter device driver. The following flags are defined:

SC_RESUME	When set, means the SCSI adapter device driver should resume transaction queuing for this ID/LUN. Error recovery is complete after a SCIOHALT operation, check condition, or severe SCSI bus error. This flag is used to restart the SCSI adapter device driver following a reported error.
SC_DELAY_CMD	When set, means the SCSI adapter device driver should delay sending this command (following a SCSI reset or BDR to this device) by at least the number of seconds specified to the SCSI adapter device driver in its configuration information. For SCSI devices that do not require this function, this flag should not be set.

SC_Q_CLR	<p>When set, means the SCSI adapter driver should clear its transaction queue for this ID/LUN. The transaction containing this flag setting does not require an actual SCSI command in the <code>sc_buf</code> because it is flushed back to the SCSI device driver with the rest of the transactions for this ID/LUN. However, this transaction must have the SCSI ID field (<code>sc_buf.scsi_command.scsi_id</code>) and the LUN fields (<code>sc_buf.scsi_command.scsi_cmd.lun</code> and <code>sc_buf.lun</code>) filled in with the device's SCSI ID and logical unit number (LUN). This flag is valid only during error recovery of a check condition or command terminated at a command tag queuing device when the SC_DID_NOT_CLR_Q flag is set in the <code>sc_buf.adap_q_status</code> field (see "SCSI Command Tag Queuing" on page 218).</p> <p>Note: When addressing LUN's 8 - 31, be sure to see the description of the <code>sc_buf.lun</code> field within the <code>sc_buf</code> structure.</p>
SC_Q_RESUME	<p>When set, means that the SCSI adapter driver should resume its halted transaction queue for this ID/LUN. The transaction containing this flag setting does not require an actual SCSI command to be sent to the SCSI adapter driver. However, this transaction must have the <code>sc_buf.scsi_command.scsi_id</code> and <code>sc_buf.scsi_command.scsi_cmd.lun</code> fields filled in with the device's SCSI ID and logical unit number. If the transaction containing this flag setting is the first issued by the SCSI device driver after it receives an error (indicating that the adapter driver's queue is halted), then the SC_RESUME flag must be set also.</p> <p>Note: When addressing LUN's 8 - 31, be sure to see the description of the <code>sc_buf.lun</code> field within the <code>sc_buf</code> structure.</p>

Other SCSI Design Considerations

Responsibilities of the SCSI Device Driver

SCSI device drivers are responsible for the following actions:

- Interfacing with block I/O and logical-volume device-driver code in the operating system.
- Translating I/O requests from the operating system into SCSI commands suitable for the particular SCSI device. These commands are then given to the SCSI adapter device driver for execution.
- Issuing any and all SCSI commands to the attached device. The SCSI adapter device driver sends no SCSI commands except those it is directed to send by the calling SCSI device driver.
- Managing SCSI device reservations and releases. In the operating system, it is assumed that other SCSI initiators may be active on the SCSI bus. Usually, the SCSI device driver reserves the SCSI device at open time and releases it at close time (except when told to do otherwise through parameters in the SCSI device driver interface). Once the device is reserved, the SCSI device driver must be prepared to reserve the SCSI device again whenever a Unit Attention condition is reported through the SCSI request-sense data.

SCSI Options to the `openx` Subroutine

SCSI device drivers in the operating system must support eight defined extended options in their open routine (that is, an **openx** subroutine). Additional extended options to the open are also allowed, but they must not conflict with predefined open options. The defined extended options are bit flags in the `ext` open parameter. These options can be specified singly or in combination with each other. The required `ext` options are defined in the `/usr/include/sys/scsi.h` header file and can

have one of the following values:

<code>SC_FORCED_OPEN</code>	Do not honor device reservation-conflict status.
<code>SC_RETAIN_RESERVATION</code>	Do not release SCSI device on close.
<code>SC_DIAGNOSTIC</code>	Enter diagnostic mode for this device.
<code>SC_NO_RESERVE</code>	Prevents the reservation of the device during an openx subroutine call to that device. Allows multiple hosts to share a device.
<code>SC_SINGLE</code>	Places the selected device in Exclusive Access mode.
<code>SC_RESV_05</code>	Reserved for future expansion.
<code>SC_RESV_07</code>	Reserved for future expansion.
<code>SC_RESV_08</code>	Reserved for future expansion.

Using the `SC_FORCED_OPEN` Option

The `SC_FORCED_OPEN` option causes the SCSI device driver to call the SCSI adapter device driver's Bus Device Reset ioctl (`SCIORESET`) operation on the first open. This forces the device to release another initiator's reservation. After the `SCIORESET` command is completed, other SCSI commands are sent as in a normal open. If any of the SCSI commands fail due to a reservation conflict, the open registers the failure as an **EBUSY** status. This is also the result if a reservation conflict occurs during a normal open. The SCSI device driver should require the caller to have appropriate authority to request the `SC_FORCED_OPEN` option since this request can force a device to drop a SCSI reservation. If the caller attempts to execute this system call without the proper authority, the SCSI device driver should return a value of -1, with the `errno` global variable set to a value of `EPERM`.

Using the `SC_RETAIN_RESERVATION` Option

The `SC_RETAIN_RESERVATION` option causes the SCSI device driver not to issue the SCSI release command during the close of the device. This guarantees a calling program control of the device (using SCSI reservation) through open and close cycles. For shared devices (for example, disk or CD-ROM), the SCSI device driver must OR together this option for all opens to a given device. If any caller requests this option, the **close** routine does not issue the release even if other opens to the device do not set `SC_RETAIN_RESERVATION`. The SCSI device driver should require the caller to have appropriate authority to request the `SC_RETAIN_RESERVATION` option since this request can allow a program to monopolize a device (for example, if this is a nonshared device). If the caller attempts to execute this system call without the proper authority, the SCSI device driver should return a value of -1, with the `errno` global variable set to a value of `EPERM`.

Using the `SC_DIAGNOSTIC` Option

The `SC_DIAGNOSTIC` option causes the SCSI device driver to enter Diagnostic mode for the given device. This option directs the SCSI device driver to perform only minimal operations to open a logical path to the device. No SCSI commands should be sent to the device in the **open** or **close** routine when the device is in Diagnostic mode. One or more ioctl operations should be provided by the SCSI device driver to allow the caller to issue SCSI commands to the attached device for diagnostic purposes.

The `SC_DIAGNOSTIC` option gives the caller an exclusive open to the selected device. This option requires appropriate authority to execute. If the caller attempts to execute this system call without the proper authority, the SCSI device driver

should return a value of -1, with the **errno** global variable set to a value of **EPERM**. The **SC_DIAGNOSTIC** option may be executed only if the device is not already opened for normal operation. If this **ioctl** operation is attempted when the device is already opened, or if an **openx** call with the **SC_DIAGNOSTIC** option is already in progress, a return value of -1 should be passed, with the **errno** global variable set to a value of **EACCES**. Once successfully opened with the **SC_DIAGNOSTIC** flag, the SCSI device driver is placed in Diagnostic mode for the selected device.

Using the **SC_NO_RESERVE** Option

The **SC_NO_RESERVE** option causes the SCSI device driver not to issue the SCSI reserve command during the opening of the device and not to issue the SCSI release command during the close of the device. This allows multiple hosts to share the device. The SCSI device driver should require the caller to have appropriate authority to request the **SC_NO_RESERVE** option, since this request allows other hosts to modify data on the device. If a caller does this kind of request then the caller must ensure data integrity between multiple hosts. If the caller attempts to execute this system call without the proper authority, the SCSI device driver should return a value of -1, with the **errno** global variable set to a value of **EPERM**.

Using the **SC_SINGLE** Option

The **SC_SINGLE** option causes the SCSI device driver to issue a normal open, but does not allow another caller to issue another open until the first caller has closed the device. This request gives the caller an exclusive open to the selected device. If this **openx** is attempted when the device is already open, a return value of -1 is passed, with the **errno** global variable set to a value of **EBUSY**.

Once successfully opened, the device is placed in Exclusive Access mode. If another caller tries to do any type of **open**, a return value of -1 is passed, with the **errno** global variable set to a value of **EACCES**.

The remaining options for the *ext* parameter are reserved for future requirements.

Implementation note: The following table shows how the various combinations of *ext* options should be handled in the SCSI device driver.

EXT OPTIONS	Device Driver Action	
	Open	Close
openx ext option		
none	normal	normal
diag	no SCSI commands	no SCSI commands
diag + force	issue SCIORESET otherwise, no SCSI commands issued	no SCSI commands
diag + force + no_reserve	issue SCIORESET; otherwise, no SCSI commands issued	no SCSI commands
diag + force + no_reserve + single	issue SCIORESET; otherwise, no SCSI commands issued	no SCSI commands
diag + force +retain	issue SCIORESET; otherwise, no SCSI commands issued	no SCSI commands
diag + force +retain + no_reserve	issue SCIORESET; otherwise, no SCSI commands issued	no SCSI commands

EXT OPTIONS	Device Driver Action	
	Open	Close
openx ext option		
diag + force +retain + no_reserve + single	issue SCIORESET; otherwise, no SCSI commands issued	no SCSI commands
diag + force +retain + single	issue SCIORESET; otherwise, no SCSI commands issued	no SCSI commands
diag + force + single	issue SCIORESET; otherwise, no SCSI commands issued	no SCSI commands
diag+no_reserve	no SCSI commands	no SCSI commands
diag + retain	no SCSI commands	no SCSI commands
diag + retain + no_reserve	no SCSI commands	no SCSI commands
diag + retain + no_reserve + single	no SCSI commands	no SCSI commands
diag + retain + single	no SCSI commands	no SCSI commands
diag + single	no SCSI commands	no SCSI commands
diag + single + no_reserve	no SCSI commands	no SCSI commands
force	normal, except SCIORESET issued prior to any SCSI commands	normal
force + no_reserve	normal except SCIORESET issued prior to any SCSI commands. No RESERVE command issued.	normal except no RELEASE
force + retain	normal, except SCIORESET issued prior to any SCSI commands	no RELEASE
force + retain + no_reserve	normal except SCIORESET issued prior to any SCSI commands. No RESERVE command issued.	no RELEASE
force + retain + no_reserve + single	normal except SCIORESET issued prior to any SCSI commands. No RESERVE command issued.	no RELEASE
force + retain + single	normal except SCIORESET issued prior to any SCSI commands	no RELEASE
force + single	normal except SCIORESET issued prior to any SCSI commands	normal
force + single + no_reserve	normal except SCIORESET issued prior to any SCSI commands. No RESERVE command issued.	no RELEASE
no_reserve	no RESERVE	no RELEASE
retain	normal	no RELEASE
retain + no_reserve	no RESERVE	no RELEASE
retain + single	normal	no RELEASE

EXT OPTIONS	Device Driver Action	
openx <i>ext</i> option	Open	Close
retain + single + no_reserve	normal except no RESERVE command issued	no RELEASE
single	normal	normal
single + no_reserve	no RESERVE	no RELEASE

Closing the SCSI Device

When a SCSI device driver is preparing to close a device through the SCSI adapter device driver, it must ensure that all transactions are complete. When the SCSI adapter device driver receives a **SCIOSTOP** ioctl operation and there are pending I/O requests, the ioctl operation does not return until all have completed. New requests received during this time are rejected from the adapter device driver's **ddstrategy** routine.

When the SCSI adapter device driver receives an **SCIOSTOPTGT** ioctl operation, it must forcibly free any receive data buffers that have been queued to the SCSI device driver for this device and have not been returned to the SCSI adapter device driver through the buffer free routine. The SCSI device driver is responsible for making sure all the receive data buffers are freed before calling the **SCIOSTOPTGT** ioctl operation. However, the SCSI adapter device driver must check that this is done, and, if necessary, forcibly free the buffers. The buffers must be freed because those not freed result in memory areas being permanently lost to the system (until the next boot).

To allow the SCSI adapter device driver to free buffers that are sent to the SCSI device driver but never returned, it must track which **tm_bufs** are currently queued to the SCSI device driver. Tracking **tm_bufs** requires the SCSI adapter device driver to violate the general SCSI rule, which states the SCSI adapter device driver should not modify the **tm_bufs** structure while it is queued to the SCSI device driver. This exception to the rule is necessary since it is never acceptable not to free memory allocated from the system.

SCSI Error Processing

It is the responsibility of the SCSI device driver to process SCSI check conditions and other returned errors properly. The SCSI adapter device driver only passes SCSI commands without otherwise processing them and is not responsible for device error recovery.

Device Driver and Adapter Device Driver Interfaces

The SCSI device drivers can have both character (raw) and block special files in the **/dev** directory. The SCSI adapter device driver has only character (raw) special files in the **/dev** directory and has only the **ddconfig**, **ddopen**, **ddclose**, **dddump**, and **ddioctl** entry points available to operating system programs. The **ddread** and **ddwrite** entry points are not implemented.

Internally, the **devsw** table has entry points for the **ddconfig**, **ddopen**, **ddclose**, **dddump**, **ddioctl**, and **ddstrategy** routines. The SCSI device drivers pass their SCSI commands to the SCSI adapter device driver by calling the SCSI adapter device

driver **ddstrategy** routine. (This routine is unavailable to other operating system programs due to the lack of a block-device special file.)

Access to the SCSI adapter device driver's **ddconfig**, **ddopen**, **ddclose**, **dddump**, **ddioctl**, and **ddstrategy** entry points by the SCSI device drivers is performed through the kernel services provided. These include such services as **fp_opendev**, **fp_close**, **fp_ioctl**, **devdump**, and **devstrategy**.

Performing SCSI Dumps

A SCSI adapter device driver must have a **dddump** entry point if it is used to access a system dump device. A SCSI device driver must have a **dddump** entry point if it drives a dump device. Examples of dump devices are disks and tapes.

Note: System services providing interrupt and timer services are unavailable for use in the dump routine. Kernel DMA services are assumed to be available for use by the dump routine. The SCSI adapter device driver should be designed to ignore extra **DUMPINIT** and **DUMPSTART** commands to the **dddump** entry point.

The **DUMPQUERY** option should return a minimum transfer size of 0 bytes, and a maximum transfer size equal to the maximum transfer size supported by the SCSI adapter device driver.

Calls to the SCSI adapter device driver **DUMPWRITE** option should use the *arg* parameter as a pointer to the **sc_buf** structure to be processed. Using this interface, a SCSI **write** command can be executed on a previously started (opened) target device. The *uiop* parameter is ignored by the SCSI adapter device driver during the **DUMPWRITE** command. Spanned, or consolidated, commands are not supported using the **DUMPWRITE** option. Gathered **write** commands are also not supported using the **DUMPWRITE** option. No queuing of **sc_buf** structures is supported during dump processing since the dump routine runs essentially as a subroutine call from the caller's dump routine. Control is returned when the entire **sc_buf** structure has been processed.

Note: No error recovery is employed during a **DUMPWRITE** operation because any error that occurs during the operation is a problem. Return values from the call to the **dddump** routine indicate the specific nature of the failure.

Successful completion of the selected operation is indicated by a 0 return value to the subroutine. Unsuccessful completion is indicated by a return code set to one of the following values for the **errno** global variable. The various **sc_buf** status fields, including the **b_error** field, are not set by the SCSI adapter device driver at completion of the **DUMPWRITE** command. Error logging is, of necessity, not supported during the dump.

- An **errno** value of **EINVAL** indicates that a request that was not valid passed to the SCSI adapter device driver, such as to attempt a **DUMPSTART** command before successfully executing a **DUMPINIT** command.
- An **errno** value of **EIO** indicates that the SCSI adapter device driver was unable to complete the command due to a lack of required resources or an I/O error.
- An **errno** value of **ETIMEDOUT** indicates that the adapter did not respond with completion status before the passed command time-out value expired.

SCSI Target-Mode Overview

Note: This operation is not supported by all SCSI I/O controllers.

The SCSI target-mode interface is intended to be used with the SCSI initiator-mode interface to provide the equivalent of a full-duplex communications path between processor type devices. Both communicating devices must support target-mode and initiator-mode. To work with the SCSI subsystem in this manner, an attached device's target-mode and initiator-mode interfaces must meet certain minimum requirements:

- The device's target-mode interface must be capable of receiving and processing at least the following SCSI commands:
 - **send**
 - **request sense**
 - **inquiry**

The data returned by the **inquiry** command must set the peripheral device type field to processor device. The device should support the vendor and product identification fields. Additional functional SCSI requirements, such as SCSI message support, must be addressed by examining the detailed functional specification of the SCSI initiator that the target-mode device is attached to.

- The attached device's initiator mode interface must be capable of sending the following SCSI commands:
 - **send**
 - **request sense**

In addition, the **inquiry** command should be supported by the attached initiator if it needs to identify SCSI target devices. Additional functional SCSI requirements, such as SCSI message support, must be addressed by examining the detailed functional specification of the SCSI target that the initiator-mode device is attached to.

Configuring and Using SCSI Target Mode

The adapter, acting as either a target or initiator device, requires its own SCSI ID. This ID, as well as the IDs of all attached devices on this SCSI bus, must be unique and between 0 and 7, inclusive. Since each device on the bus must be at a unique ID, the user must complete any installation and configuration of the SCSI devices required to set the correct IDs before physically cabling the devices together. Failure to do so will produce unpredictable results.

SCSI target mode in the SCSI subsystem does not attempt to implement any receive-data protocol, with the exception of actions taken to prevent an application from excessive receive-data-buffer usage. Any protocol required to maintain or otherwise manage the communications of data must be implemented in user-supplied programs. The only delays in receiving data are those inherent in the SCSI subsystem and the hardware environment in which it operates.

The SCSI target mode is capable of simultaneously receiving data from all attached SCSI IDs using SCSI **send** commands. In target-mode, the host adapter is assumed to act as a single SCSI Logical Unit Number (LUN) at its assigned SCSI ID. Therefore, only one logical connection is possible between each attached SCSI

initiator on the SCSI Bus and the host adapter. The SCSI subsystem is designed to be fully capable of simultaneously sending SCSI commands in initiator-mode while receiving data in target-mode.

Managing Receive-Data Buffers

In the SCSI subsystem target-mode interface, the SCSI adapter device driver is responsible for managing the receive-data buffers versus the SCSI device driver because the buffering is dependent upon how the adapter works. It is not possible for the SCSI device driver to run a single approach that is capable of making full use of the performance advantages of various adapter's buffering schemes. With the SCSI adapter device driver layer performing the buffer management, the SCSI device driver can be interfaced to a variety of adapter types and can potentially get the best possible performance out of each adapter. This approach also allows multiple SCSI target-mode device drivers to be run on top of adapters that use a shared-pool buffer management scheme. This would not be possible if the target-mode device drivers managed the buffers.

Understanding Target-Mode Data Pacing

Because it is possible for the attached initiator device to send data faster than the host operating system and associated application can process it, eventually the situation arises in which all buffers for this device instance are in use at the same time. There are two possible scenarios:

- The previous **send** command has been received by the adapter, but there is no space for the next **send** command.
- The **send** command is not yet completed, and there is no space for the remaining data.

In both cases, the combination of the SCSI adapter device driver and the SCSI adapter must be capable of stopping the flow of data from the initiator device.

SCSI Adapter Device Driver

The adapter can handle both cases described previously by simply accepting the **send** command (if newly received) and then disconnecting during the data phase. When buffer space becomes available, the SCSI adapter reconnects and continues the data transfer. As an alternative, when handling a newly received command, a check condition can be given back to the initiator to indicate a lack of resources. The implementation of this alternative is adapter-dependent. The technique of accepting the command and then disconnecting until buffer space is available should result in better throughput, as it avoids both a **request sense** command and the retry of the **send** command.

For adapters allowing a shared pool of buffers to be used for all attached initiators' data transfers, an additional problem can result. If any single initiator instance is allowed to transfer data continually, the entire shared pool of buffers can fill up. These filled-up buffers prevent other initiator instances from transferring data. To solve this problem, the combination of the SCSI adapter device driver and the host SCSI adapter must stop the flow of data from a particular initiator ID on the bus. This could include disconnecting during the data phase for a particular ID but allowing other IDs to continue data transfer. This could begin when the number of **tm_buf** structures on a target-mode instance's **tm_buf** queue equals the number of buffers allocated for this device. When a threshold percentage of the number of buffers is processed and returned to the SCSI adapter device driver's buffer-free routine, the ID can be enabled again for the continuation of data transfer.

SCSI Device Driver

The SCSI device driver can optionally be informed by the SCSI adapter device driver whenever all buffers for this device are in use. This is known as a maximum-buffer-usage event. To pass this information, the SCSI device driver must be registered for notification of asynchronous event status from the SCSI adapter device driver. Registration is done by calling the SCSI adapter device-driver ioctl entry point with the **SCIOEVENT** operation. If registering for event notification, the SCSI device driver receives notification of all asynchronous events, not just the maximum buffer usage event.

Understanding the SCSI Target Mode Device Driver Receive Buffer Routine

The SCSI target-mode device-driver **receive buffer** routine must be a pinned routine that the SCSI adapter device driver can directly address. This routine is called directly from the SCSI adapter device driver hardware interrupt handling routine. The SCSI device driver writer must be aware of how this routine affects the design of the SCSI device driver.

First, since the **receive buffer** routine is running on the hardware interrupt level, the SCSI device driver must limit operations in order to limit routine processing time. In particular, the data copy, which occurs because the data is queued ahead of the user read request, must not occur in the **receive buffer** routine. Data copying in this routine will adversely affect system response time. Data copy is best performed in a process level SCSI device-driver routine. This routine sleeps, waiting for data, and is awakened by the **receive buffer** routine. Typically, this process level routine is the SCSI device driver's **read** routine.

Second, the **receive buffer** routine is called at the SCSI adapter device driver hardware interrupt level, so care must be taken when disabling interrupts. They must be disabled to the correct level in places in the SCSI device driver's lower execution priority routines which manipulate variables also modified in the **receive buffer** routine. To allow the SCSI device driver to disable to the correct level, the SCSI adapter device-driver writer must provide a configuration database attribute, named **intr_priority**, that defines the interrupt class, or priority, the adapter runs on. The SCSI device-driver configuration method should pass this attribute to the SCSI device driver along with other configuration data for the device instance.

Third, the SCSI device-driver writer must follow any other general system rules for writing a routine that must execute in an interrupt environment. For example, the routine must not attempt to sleep or wait on I/O operations. It can perform wake-up calls to allow the process level to handle those operations.

Duties of the SCSI device driver **receive buffer** routine include:

- Matching the data with the appropriate target-mode instance.
- Queuing the **tm_buf** structures to the appropriate target-mode instance.
- Waking up the process-level routine for further processing of the received data.

After the **tm_buf** structure has been passed to the SCSI device driver **receive buffer** routine, the SCSI device driver is considered to be responsible for it. Responsibilities include processing the data and any error conditions and also maintaining the next pointer for chained **tm_buf** structures. The SCSI device driver's responsibilities for the **tm_buf** structures end when it passes the structure back to the SCSI adapter device driver.

Until the **tm_buf** structure is again passed to the SCSI device driver **receive buffer** routine, the SCSI adapter device driver is considered responsible for it. The SCSI adapter device-driver writer must be aware that during the time the SCSI device driver is responsible for the **tm_buf** structure, it is still possible for the SCSI adapter device driver to access the structure's contents. Access is possible because only one copy of the structure is in memory, and only a pointer to the structure is passed to the SCSI device driver.

Note: Under no circumstances should the SCSI adapter device driver access the structure or modify its contents while the SCSI device driver is responsible for it, or the other way around.

It is recommended that the SCSI device-driver writer implement a threshold level to wake up the process level with available **tm_buf** structures. This way, processing for some of the buffers, including copying the data to the user buffer, can be overlapped with time spent waiting for more data. It is also recommended the writer implement a threshold level for these buffers to handle cases where the **send** command data length exceeds the aggregate receive-data buffer space. A suggested threshold level is 25% of the device's total buffers. That is, when 25% or more of the number of buffers allocated for this device is queued and no end to the **send** command is encountered, the SCSI device driver receive buffer routine should wake the process level to process these buffers.

Understanding the **tm_buf** Structure

The **tm_buf** structure is used for communication between the SCSI device driver and the SCSI adapter device driver for a target-mode received-data buffer. The **tm_buf** structure is passed by pointer directly to routines whose entry points have been registered through the **SCIOSTARTTGT** ioctl operation of the SCSI adapter device driver. The SCSI device driver is required to call this ioctl operation when opening a target-mode device instance.

Fields in the **tm_buf** Structure

The **tm_buf** structure contains certain fields used to pass a received data buffer from the SCSI adapter device driver to the SCSI device driver. Other fields are used to pass returned status back to the SCSI device driver. After processing the data, the **tm_buf** structure is passed back from the SCSI device driver to the SCSI adapter device driver to allow the buffer to be reused. The **tm_buf** structure is defined in the `/usr/include/sys/scsi.h` file and contains the following fields:

Note: Reserved fields must not be modified by the SCSI device driver, unless noted otherwise. Nonreserved fields can be modified, except where noted otherwise.

1. The **tm_correlator** field is an optional field for the SCSI device driver. This field is a copy of the field with the same name that was passed by the SCSI device driver in the **SCIOSTARTTGT** ioctl. The SCSI device driver should use this field to speed the search for the target-mode device instance the **tm_buf** structure is associated with. Alternatively, the SCSI device driver can combine the **tm_buf.user_id** and **tm_buf.adap_devno** fields to find the associated device.
2. The **adap_devno** field is the device major and minor numbers of the adapter instance on which this target mode device is defined. This field may be used to find the particular target-mode instance the **tm_buf** structure is associated with.

Note: The SCSI device driver must not modify this field.

3. The `data_addr` field is the kernel space address where the data begins for this buffer.
4. The `data_len` field is the length of valid data in the buffer starting at the `tm_buf.data_addr` location in memory.
5. The `user_flag` field is a set of bit flags that can be set to communicate information about this data buffer to the SCSI device driver. Except where noted, one or more of the following flags can be set:

TM_HASDATA Set to indicate a valid `tm_buf` structure

TM_MORE_DATA Set if more data is coming (that is, more `tm_buf` structures) for a particular `send` command. This is only possible for adapters that support spanning the `send` command data across multiple receive buffers. This flag cannot be used with the **TM_ERROR** flag.

TM_ERROR Set if any error occurred on a particular `send` command. This flag cannot be used with the **TM_MORE_DATA** flag.

6. The `user_id` field is set to the SCSI ID of the initiator that sent the data to this target mode instance. If more than one adapter is used for target mode in this system, this ID may not be unique. Therefore, this field must be used in combination with the `tm_buf.adap_devno` field to find the target-mode instance this ID is associated with.

Note: The SCSI device driver must not modify this field.

7. The `status_validity` field contains the following bit flag:

SC_ADAPTER_ERROR Indicates the `tm_buf.general_card_status` is valid.

8. The `general_card_status` field is a returned status field that gives a broad indication of the class of error encountered by the adapter. This field is valid when its status-validity bit is set in the `tm_buf.status_validity` field. The definition of this field is the same as that found in the structure definition (see “Understanding the `sc_buf` Structure” on page 218), except the **SC_CMD_TIMEOUT** value is not possible and is never returned for a target-mode transfer.
9. The `next` field is a `tm_buf` pointer that is either null, meaning this is the only or last `tm_buf` structure, or else contains a non-null pointer to the next `tm_buf` structure.

Understanding the Execution of SCSI Target-Mode Requests

The target-mode interface provided by the SCSI subsystem is designed to handle data reception from SCSI `send` commands. The host SCSI adapter acts as a secondary device that waits for an attached initiator device to issue a SCSI `send` command. The SCSI `send` command data is received by buffers managed by the SCSI adapter device driver. The `tm_buf` structure is used to manage individual buffers. For each buffer of data received from an attached initiator, the SCSI adapter device driver passes a `tm_buf` structure to the SCSI device driver for processing. Multiple `tm_buf` structures can be linked together and passed to the SCSI device driver at one time. When the SCSI device driver has processed one or more `tm_buf` structures, it passes the `tm_buf` structures back to the SCSI adapter device driver so they can be reused.

Detailed Execution of Target-Mode Requests

When a `send` command is received by the host SCSI adapter, data is placed in one or more receive-data buffers. These buffers are made available to the adapter by the SCSI adapter device driver. The procedure by which the data gets from the

SCSI bus to the system-memory buffer is adapter-dependent. The SCSI adapter device driver takes the received data and updates the information in one or more **tm_buf** structures in order to identify the data to the SCSI device driver. This process includes filling the `tm_correlator`, `adap_devno`, `data_addr`, `data_len`, `user_flag`, and `user_id` fields. Error status information is put in the `status_validity` and `general_card_status` fields. The next field is set to null to indicate this is the only element, or set to non-null to link multiple **tm_buf** structures. If there are multiple **tm_buf** structures, the final `tm_buf.next` field is set to null to terminate the chain. If there are multiple **tm_buf** structures and they are linked, they must all be from the same initiator SCSI ID. The `tm_buf.tm_correlator` field, in this case, has the same value as it does in the **SCIOSTARTTGT** ioctl operation to the SCSI adapter device driver. The SCSI device driver should use this field to speed the search for the target-mode instance designated by this **tm_buf** structure. For example, when using the value of `tm_buf.tm_correlator` as a pointer to the device-information structure associated with this target-mode instance.

Each **send** command, no matter how short its data length, requires its own **tm_buf** structure. For host SCSI adapters capable of spanning multiple receive-data buffers with data from a single **send** command, the SCSI adapter device driver must set the **TM_MORE_DATA** flag in the `tm_buf.user_flag` fields of all but the final **tm_buf** structure holding data for the **send** command. The SCSI device driver must be designed to support the **TM_MORE_DATA** flag. Using this flag, the target-mode SCSI device driver can associate multiple buffers with the single transfer they represent. The end of a **send** command will be the boundary used by the SCSI device driver to satisfy a user read request.

The SCSI adapter device driver is responsible for sending the **tm_buf** structures for a particular initiator SCSI ID to the SCSI device driver in the order they were received. The SCSI device driver is responsible for processing these **tm_buf** structures in the order they were received. There is no particular ordering implied in the processing of simultaneous **send** commands from different SCSI IDs, as long as the data from an individual SCSI ID's **send** command is processed in the order it was received.

The pointer to the **tm_buf** structure chain is passed by the SCSI adapter device driver to the SCSI device driver's receive buffer routine. The address of this routine is registered with the SCSI adapter device driver by the SCSI device driver using the **SCIOSTARTTGT** ioctl. The duties of the receive buffer routine include queuing the **tm_buf** structures and waking up a process-level routine (typically the SCSI device driver's **read** routine) to process the received data.

When the process-level SCSI device driver routine finishes processing one or more **tm_buf** structures, it passes them to the SCSI adapter device driver's buffer-free routine. The address of this routine is registered with the SCSI device driver in an output field in the structure passed to the SCSI adapter device driver **SCIOSTARTTGT** ioctl operation. The buffer-free routine must be a pinned routine the SCSI device driver can directly access. The buffer-free routine is typically called directly from the SCSI device driver buffer-handling routine. The SCSI device driver chains one or more **tm_buf** structures by using the next field (a null value for the last `tm_buf` next field ends the chain). It then passes a pointer, which points to the head of the chain, to the SCSI adapter device driver buffer-free routine. These **tm_buf** structures must all be for the same target-mode instance. Also, the SCSI device driver must not modify the `tm_buf.user_id` or `tm_buf.adap_devno` field.

The SCSI adapter device driver takes the **tm_buf** structures passed to its buffer-free routine and attempts to make the described receive buffers available to the adapter for future data transfers. Since it is desirable to keep as many buffers as possible available to the adapter, the SCSI device driver should pass processed **tm_buf** structures to the SCSI-adapter device driver's buffer-free routine as quickly as possible. The writer of a SCSI device driver should avoid requiring the last buffer of a **send** command to be received before processing buffers, as this could cause a situation where all buffers are in use and the **send** command has not completed. It is recommended that the writer therefore place a threshold of 25% on the free buffers. That is, when 25% or more of the number of buffers allocated for this device have been processed and the **send** command is not completed, the SCSI device driver should free the processed buffers by passing them to the SCSI adapter device driver's buffer-free routine.

Required SCSI Adapter Device Driver ioctl Commands

Various ioctl operations must be performed for proper operation of the SCSI adapter device driver. The ioctl operations described here are the minimum set of commands the SCSI adapter device driver must implement to support SCSI device drivers. Other operations may be required in the SCSI adapter device driver to support, for example, system management facilities and diagnostics. SCSI device driver writers also need to understand these ioctl operations.

Every SCSI adapter device driver must support the **IOCINFO** ioctl operation. The structure to be returned to the caller is the **devinfo** structure, including the **scsi** union definition for the SCSI adapter, which can be found in the **/usr/include/sys/devinfo.h** file. The SCSI device driver should request the **IOCINFO** ioctl operation (probably during its open routine) to get the maximum transfer size of the adapter.

Note: The SCSI adapter device driver ioctl operations can only be called from the process level. They cannot be executed from a call on any more favored priority levels. Attempting to call them from a more favored priority level can result in a system crash.

Initiator-Mode ioctl Commands

The following **SCIOSTART** and **SCIOSTOP** operations must be sent by the SCSI device driver (for the open and close routines, respectively) for each device. They cause the SCSI adapter device driver to allocate and initialize internal resources. The **SCIOHALT** ioctl operation is used to abort pending or running commands, usually after signal processing by the SCSI device driver. This might be used by a SCSI device driver to end an operation instead of waiting for completion or a time out. The **SCIORESET** operation is provided for clearing device hard errors and competing initiator reservations during open processing by the SCSI device driver. The **SCIOGTHW** operation is supported by SCSI adapter device drivers that support gathered write commands to target devices.

Except where noted otherwise, the *arg* parameter for each of the ioctl operations described here must contain a long integer. In this field, the least significant byte is the SCSI LUN and the next least significant byte is the SCSI ID value. (The upper two bytes are reserved and should be set to 0.) This provides the information required to allocate or deallocate resources and perform SCSI bus operations for the ioctl operation requested.

The following information is provided on the various ioctl operations:

SCIOSTART This operation allocates and initializes SCSI device-dependent information local to the SCSI adapter device driver. Run this operation only on the first open of an ID/LUN device. Subsequent **SCIOSTART** commands to the same ID/LUN fail unless an intervening **SCIOSTOP** command is issued.

The following values for the **errno** global variable are supported:

0 Indicates successful completion.

EIO Indicates lack of resources or other error-preventing device allocation.

EINVAL Indicates that the selected SCSI ID and LUN are already in use, or the SCSI ID matches the adapter ID.

ETIMEDOUT Indicates that the command did not complete.

SCIOSTOP This operation deallocates resources local to the SCSI adapter device driver for this SCSI device. This should be run on the last close of an ID/LUN device. If an **SCIOSTART** operation has not been previously issued, this command is unsuccessful.

The following values for the **errno** global variable should be supported:

0 Indicates successful completion.

EIO Indicates error preventing device deallocation.

EINVAL Indicates that the selected SCSI ID and LUN have not been started.

ETIMEDOUT Indicates that the command did not complete.

SCIOHALT This operation halts outstanding transactions to this ID/LUN device and causes the SCSI adapter device driver to stop accepting transactions for this device. This situation remains in effect until the SCSI device driver sends another transaction with the **SC_RESUME** flag set (in the **sc_buf.flags** field) for this ID/LUN combination. The **SCIOHALT** ioctl operation causes the SCSI adapter device driver to fail the command in progress, if any, as well as all queued commands for the device with a return value of **ENXIO** in the **sc_buf.bufstruct.b_error** field. If an **SCIOSTART** operation has not been previously issued, this command fails.

The following values for the **errno** global variable are supported:

0 Indicates successful completion.

EIO Indicates an unrecovered I/O error occurred.

EINVAL Indicates that the selected SCSI ID and LUN have not been started.

ETIMEDOUT Indicates that the command did not complete.

SCIORESET This operation causes the SCSI adapter device driver to send a SCSI Bus Device Reset (BDR) message to the selected SCSI ID. For this operation, the SCSI device driver should set the LUN in the *arg* parameter to the LUN ID of a LUN on this SCSI ID, which has been successfully started using the **SCIOSTART** operation.

The SCSI device driver should use this command only when directed to do a *forced open*. This occurs in two possible situations: one, when it is desirable to force the device to drop a SCSI reservation; two, when the device needs to be reset to clear an error condition (for example, when running diagnostics on this device).

Note: In normal system operation, this command should not be issued, as it would force the device to drop a SCSI reservation another initiator (and, hence, another system) may have. If an **SCIOSTART** operation has not been previously issued, this command is unsuccessful.

The following values for the **errno** global variable are supported:

0 Indicates successful completion.

EIO Indicates an unrecovered I/O error occurred.

EINVAL

Indicates that the selected SCSI ID and LUN have not been started.

ETIMEDOUT

Indicates that the command did not complete.

SCIOGTHW This operation is only supported by SCSI adapter device drivers that support gathered write commands. The purpose of the operation is to indicate support for gathered writes to SCSI device drivers that intend to use this facility. If the SCSI adapter device driver does not support gathered write commands, it must fail the operation. The SCSI device driver should call this operation from its open routine for a particular device instance. If the operation is unsuccessful, the SCSI device driver should not attempt to run a gathered write command.

The *arg* parameter to the **SCIOGTHW** is set to null by the caller to indicate that no input parameter is passed:

The following values for the **errno** global variable are supported:

0 Indicates successful completion and in particular that the adapter driver supports gathered writes.

EINVAL

Indicates that the SCSI adapter device driver does not support gathered writes.

Target-Mode ioctl Commands

The following **SCIOSTARTTGT** and **SCIOSTOPTGT** operations must be sent by the SCSI device driver (for the open and close routines, respectively) for each target-mode device instance. This causes the SCSI adapter device driver to allocate and initialize internal resources, and, if necessary, prepare the hardware for operation.

Target-mode support in the SCSI device driver and SCSI adapter device driver is optional. A failing return code from these commands, in the absence of any

programming error, indicates target mode is not supported. If the SCSI device driver requires target mode, it must check the return code to verify the SCSI adapter device driver supports it.

Only a kernel process or device driver can call these ioctls. If attempted by a user process, the ioctl will fail, and the **errno** global variable will be set to **EPERM**.

The following information is provided on the various target-mode ioctl operations:

SCIOSTARTTGT

This operation opens a logical path to a SCSI initiator device. It allocates and initializes SCSI device-dependent information local to the SCSI adapter device driver. This is run by the SCSI device driver in its open routine. Subsequent **SCIOSTARTTGT** commands to the same ID (LUN is always 0) are unsuccessful unless an intervening **SCIOSTOPTGT** is issued. This command also causes the SCSI adapter device driver to allocate system buffer areas to hold data received from the initiator, and makes the adapter ready to receive data from the selected initiator.

The *arg* parameter to the **SCIOSTARTTGT** should be set to the address of an **sc_strt_tgt** structure, which is defined in the **/usr/include/sys/scsi.h** file. The following parameters are supported:

- id* The caller fills in the SCSI ID of the attached SCSI initiator.
- lun* The caller sets the LUN to 0, as the initiator LUN is ignored for received data.
- buf_size* The caller specifies size in bytes to be used for each receive buffer allocated for this host target instance.
- num_bufs* The caller specifies how many buffers to allocate for this target instance.
- tm_correlator* The caller optionally places a value in this field to be passed back in each **tm_buf** for this target instance.
- recv_func* The caller places in this field the address of a pinned routine the SCSI adapter device driver should call to pass **tm_bufs** received for this target instance.
- free_func* This is an output parameter the SCSI adapter device driver fills with the address of a pinned routine which the SCSI device driver calls to pass **tm_bufs** after they have been processed. The SCSI adapter device driver ignores the value passed as input.

Note: All reserved fields should be set to 0 by the caller.

The following values for the **errno** global variable are supported:

- 0** Indicates successful completion.

EINVAL

An **SCIOSTARTTGT** command has already been issued to this SCSI ID.

The passed SCSI ID is the same as that of the adapter.

The LUN ID field is not set to zero.

The `buf_size` is not valid. This is an adapter dependent value.

The `num_bufs` is not valid. This is an adapter dependent value.

The `recv_func` value, which cannot be null, is not valid.

EPERM

Indicates the caller is not running in kernel mode, which is the only mode allowed to execute this operation.

ENOMEM

Indicates that a memory allocation failure has occurred.

EIO Indicates an I/O error occurred, preventing the device driver from completing **SCIOSTARTTGT** processing.

SCIOSTOPTGT

This operation closes a logical path to a SCSI initiator device. It causes the SCSI adapter device driver to deallocate device dependent information areas allocated in response to a **SCIOSTARTTGT** operation. It also causes the SCSI adapter device driver to deallocate system buffer areas used to hold data received from the initiator, and to disable the host adapter's ability to receive data from the selected initiator.

The *arg* parameter to the **SCIOSTOPTGT** ioctl should be set to the address of an **sc_stop_tgt** structure, which is defined in the `/usr/include/sys/scsi.h` file. The caller fills in the **id** field with the SCSI ID of the SCSI initiator, and sets the **lun** field to 0 as the initiator LUN is ignored for received data. Reserved fields should be set to 0 by the caller.

The following values for the **errno** global variable should be supported:

0 Indicates successful completion.

EINVAL

An **SCIOSTARTTGT** command has not been previously issued to this SCSI ID.

EPERM

Indicates the caller is not running in kernel mode, which is the only mode allowed to execute this operation.

Target- and Initiator-Mode ioctl Commands

For either target or initiator mode, the SCSI device driver may issue an **SCIOEVENT** ioctl operation to register for receiving asynchronous event status from the SCSI adapter device driver for a particular device instance. This is an optional call for the SCSI device driver, and is optionally supported for the SCSI adapter device driver. A failing return code from this command, in the absence of any programming error, indicates it is not supported. If the SCSI device driver requires this function, it must check the return code to verify the SCSI adapter device driver supports it.

Only a kernel process or device driver can invoke these ioctls. If attempted by a user process, the ioctl will fail, and the **errno** global variable will be set to **EPERM**.

The event registration performed by this ioctl operation is allowed once per device session. Only the first **SCIOEVENT** ioctl operation is accepted after the device session is opened. Succeeding **SCIOEVENT** ioctl operations will fail, and the **errno** global variable will be set to **EINVAL**. The event registration is canceled automatically when the device session is closed.

The *arg* parameter to the **SCIOEVENT** ioctl operation should be set to the address of an **sc_event_struct** structure, which is defined in the **/usr/include/sys/scsi.h** file. The following parameters are supported:

<i>id</i>	The caller sets <i>id</i> to the SCSI ID of the attached SCSI target device for initiator-mode. For target-mode, the caller sets the <i>id</i> to the SCSI ID of the attached SCSI initiator device.
<i>lun</i>	The caller sets the <i>lun</i> field to the SCSI LUN of the attached SCSI target device for initiator-mode. For target-mode, the caller sets the <i>lun</i> field to 0.
<i>mode</i>	Identifies whether the initiator- or target-mode device is being registered. These values are possible: SC_IM_MODE This is an initiator mode device. SC_TM_MODE This is a target mode device.
<i>async_correlator</i>	The caller places a value in this optional field which is saved by the SCSI adapter device driver and returned when an event occurs in this field in the sc_event_info structure. This structure is defined in the /user/include/sys/scsi.h file.
<i>async_func</i>	The caller fills in the address of a pinned routine which the SCSI adapter device driver calls whenever asynchronous event status is available. The SCSI adapter device driver passes a pointer to a sc_event_info structure to the caller's async_func routine.

Note: All reserved fields should be set to 0 by the caller.

The following values for the **errno** global variable are supported:

0	Indicates successful completion.
EINVAL	Either an SCIOSTART or SCIOSTARTTGT has not been issued to this device instance, or this device is already registered for async events.
EPERM	Indicates the caller is not running in kernel mode, which is the only mode allowed to execute this operation.

Chapter 13. Fibre Channel Protocol for SCSI Subsystem

This overview describes the interface between a Fibre Channel Protocol for SCSI (FCP) device driver and a FCP adapter device driver. The term FC SCSI is also used to refer to FCP devices. It is directed toward those wishing to design and write a FCP device driver that interfaces with an existing FCP adapter device driver. It is also meant for those wishing to design and write a FCP adapter device driver that interfaces with existing FCP device drivers.

FCP Subsystem Overview

The main topics covered in this overview are:

- “Responsibilities of the FCP Adapter Device Driver”
- “Responsibilities of the FCP Device Driver”
- “Communication between FCP Devices” on page 242
- “Initiator-Mode Support” on page 242

This section frequently refers to both a **FCP device driver** and a **FCP adapter device driver**. These two distinct device drivers work together in a layered approach to support attachment of a range of FCP devices. The FCP adapter device driver is the *lower* device driver of the pair, and the FCP device driver is the *upper* device driver.

Responsibilities of the FCP Adapter Device Driver

The FCP adapter device driver (the lower layer) is the software interface to the system hardware. This hardware includes the FCP transport layer hardware plus any other system I/O hardware required to run an I/O request. The FCP adapter device driver hides the details of the I/O hardware from the FCP device driver. The design of the software interface allows a user with limited knowledge of the system hardware to write the upper device driver.

The FCP adapter device driver manages the FCP transport layer but not the FCP devices. It can send and receive FCP commands, but it cannot interpret the contents of the command. The lower driver also provides recovery and logging for errors related to the FCP transport layer and system I/O hardware. Management of the device specifics is left to the FCP device driver. The interface of the two drivers allows the upper driver to communicate with different FCP transport layer adapters without requiring special code paths for each adapter.

Responsibilities of the FCP Device Driver

The FCP device driver (the upper layer) provides the rest of the operating system with the software interface to a given FCP device or device class. The upper layer recognizes which FCP commands are required to control a particular FCP device or device class. The FCP device driver builds I/O requests containing device FCP commands and sends them to the FCP adapter device driver in the sequence needed to operate the device successfully. The FCP device driver cannot manage adapter resources or give the FCP command to the adapter. Specifics about the adapter and system hardware are left to the lower layer.

The FCP device driver also provides recovery and logging for errors related to the FCP device it controls.

The operating system provides several kernel services allowing the FCP device driver to communicate with FCP adapter device driver entry points without having the actual name or address of those entry points. The description contained in Logical File System Kernel Services can provide more information.

Communication between FCP Devices

When two FCP devices communicate, one assumes the initiator-mode role, and the other assumes the target-mode role. The initiator-mode device generates the FCP command, which requests an operation, and the target-mode device receives the FCP command and acts. It is possible for a FCP device to perform both roles simultaneously.

When writing a new FCP adapter device driver, the writer must know which mode or modes must be supported to meet the requirements of the FCP adapter and any interfaced FCP device drivers.

Initiator-Mode Support

The interface between the FCP device driver and the FCP adapter device driver for initiator-mode support (that is, the attached device acts as a target) is accessed through calls to the FCP adapter device driver **open** (“open” on page 276), **close** (“close” on page 276), **ioctl** (“ioctl” on page 277), and **strategy** (“strategy” on page 277) routines. I/O requests are queued to the FCP adapter device driver through calls to its strategy entry point.

Communication between the FCP device driver and the FCP adapter device driver for a particular initiator I/O request is made through the **scsi_buf** structure, which is passed to and from the strategy routine in the same way a standard driver uses a **struct buf** structure.

Understanding FCP Asynchronous Event Handling

Note: This operation is not supported by all FCP I/O controllers.

A FCP device driver can register a particular device instance for receiving asynchronous event status by calling the **SCIOLEVENT** ioctl operation for the FCP-adapter device driver (see “SCIOLEVENT” on page 266). When an event covered by the **SCIOLEVENT** ioctl operation is detected by the FCP adapter device driver, it builds an **scsi_event_info** structure and passes a pointer to the structure and to the asynchronous event-handler routine entry point, which was previously registered. The fields in the structure are filled in by the FCP adapter device driver as follows:

scsi_id

For initiator mode, this is set to the SCSI ID of the attached FCP target device. For target mode, this is set to the SCSI ID of the attached FCP initiator device.

lun_id For initiator mode, this is set to the SCSI LUN of the attached FCP target device. For target mode, this is set to 0).

mode Identifies whether the initiator or target mode device is being reported. The following values are possible:

SCSI_IM_MODE

An initiator mode device is being reported.

SCSI_TM_MODE

A target mode device is being reported.

events This field is set to indicate what event or events are being reported. The following values are possible, as defined in the `/usr/include/sys/scsi.h` file:

SCSI_FATAL_HDW_ERR

A fatal adapter hardware error occurred.

SCSI_ADAP_CMD_FAILED

An unrecoverable adapter command failure occurred.

SCSI_RESET_EVENT

A FCP transport layer reset was detected.

SCSI_BUFS_EXHAUSTED

In target-mode, a maximum buffer usage event has occurred.

adap_devno

This field is set to indicate the device major and minor numbers of the adapter on which the device is located.

async_correlator

This field is set to the value passed to the FCP adapter device driver in the `scsi_event_struct` structure. The FCP device driver may optionally use this field to provide an efficient means of associating event status with the device instance it goes with. Alternatively, the FCP device driver would use the combination of the `id`, `lun`, `mode`, and `adap_devno` fields to identify the device instance.

The information reported in the `scsi_event_info.events` field does not queue to the FCP device driver, but is instead reported as one or more flags as they occur. Since the data does not queue, the FCP adapter device driver writer can use a single `scsi_event_info` structure and pass it one at a time, by pointer, to each asynchronous event handler routine for the appropriate device instance. After determining for which device the events are being reported, the FCP device driver must copy the `scsi_event_info.events` field into local space and must not modify the contents of the rest of the `scsi_event_info` structure.

Since the event status is optional, the FCP device driver writer determines what action is necessary to take upon receiving event status. The writer may decide to save the status and report it back to the calling application, or the FCP device driver or application level program can take error recovery actions.

Defined Events and Recovery Actions

The adapter fatal hardware failure event is intended to indicate that no further commands to or from this FCP device are likely to succeed, since the adapter it is attached to has failed. It is recommended that the application end the session with the device.

The unrecoverable adapter command failure event is not necessarily a fatal condition, but it can indicate that the adapter is not functioning properly. Possible actions by the application program include:

- Ending of the session with the device in the near future.
- Ending of the session after multiple (two or more) such events.
- Attempt to continue the session indefinitely.

The SCSI Reset detection event is mainly intended as information only, but may be used by the application to perform further actions, if necessary.

The maximum buffer usage detected event only applies to a given target-mode device; it will not be reported for an initiator-mode device. This event indicates to the application that this particular target-mode device instance has filled its maximum allotted buffer space. The application should perform **read** system calls fast enough to prevent this condition. If this event occurs, data is not lost, but it is delayed to prevent further buffer usage. Data reception will be restored when the application empties enough buffers to continue reasonable operations. The **num_bufs** attribute may need to be increased to help minimize this problem. Also, it is possible that regardless of the number of buffers, the application simply is not processing received data fast enough. This may require some fine tuning of the application's data processing routines.

Asynchronous Event-Handling Routine

The FCP-device driver asynchronous event-handling routine is typically called directly from the hardware interrupt-handling routine for the FCP adapter device driver. The FCP device driver writer must be aware of how this affects the design of the FCP device driver.

Since the event handling routine is running on the hardware interrupt level, the FCP device driver must be careful to limit operations in that routine. Processing should be kept to a minimum. In particular, if any error recovery actions are performed, it is recommended that the event-handling routine set state or status flags only and allow a process level routine to perform the actual operations.

The FCP device driver must be careful to disable interrupts at the correct level in places where the FCP device driver's lower execution priority routines manipulate variables that are also modified by the event-handling routine. To allow the FCP device driver to disable at the correct level, the FCP adapter device driver writer must provide a configuration database attribute that defines the interrupt class, or priority, it runs on. This attribute must be named **intr_priority** so that the FCP device driver configuration method knows which attribute of the parent adapter to query. The FCP device driver configuration method should then pass this interrupt priority value to the FCP device driver along with other configuration data for the device instance.

The FCP device driver writer must follow any other general system rules for writing a routine which must execute in an interrupt environment. For example, the routine must not attempt to sleep or wait on I/O operations. It can perform wakeups to allow the process level to handle those operations.

Since the FCP device driver copies the information from the **scsi_event_info.events** field on each call to its asynchronous event-handling routine, there is no resource to free or any information which must be passed back later to the FCP adapter device driver.

FCP Error Recovery

If the device is in initiator mode, the error-recovery process varies depending on whether or not the device is supporting command queuing. Also some devices may support NACA=1 error recovery. Thus FCP error recovery needs to deal with the two following concepts.

autosense data

When an FCP device returns a check condition or command terminated (the `scsi_buf.scsi_status` will have the value of `SC_CHECK_CONDITION` or `SC_COMMAND_TERMINATED`, respectively), it will also return the request sense data.

Note: Subsequent commands to the FCP device will clear the request sense data.

If the FCP device driver has specified a valid autosense buffer (`scsi_buf.autosense_length > 0` and the `scsi_buf.autosense_buffer_ptr` field is not NULL), then the FCP adapter device driver will copy the returned autosense data into the buffer referenced by `scsi_buf.autosense_buffer_ptr`. When this occurs the FCP adapter device driver will set the `SC_AUTOSENSE_DATA_VALID` flag in the `scsi_buf.adap_set_flags`.

When the FCP device driver receives the SCSI status of check condition or command terminated (the `scsi_buf.scsi_status` will have the value of `SC_CHECK_CONDITION` or `SC_COMMAND_TERMINATED`, respectively), it should then determine if the `SC_AUTOSENSE_DATA_VALID` flag is set in the `scsi_buf.adap_set_flags`. If so then it should process the autosense data and not send a SCSI request sense command.

NACA=1 error recovery

Some FCP devices support setting the NACA (Normal Auto Contingent Allegiance) bit to a value of one (NACA=1) in the control byte of the SCSI command. If an FCP device returns a check condition or command terminated (the `scsi_buf.scsi_status` will have the value of `SC_CHECK_CONDITION` or `SC_COMMAND_TERMINATED`, respectively) for a command with NACA=1 set, then the FCP device will require a Clear ACA task management request to clear the error condition on the drive. The FCP device driver can issue a Clear ACA task management request by sending a transaction with the `SC_CLEAR_ACA` flag in the `sc_buf.flags` field. The `SC_CLEAR_ACA` flag can be used in conjunction with the `SC_Q_CLR` and `SC_Q_RESUME` flag in the `sc_buf.flags` to clear or resume the queue of transactions for this device, respectively. (See “FCP Initiator-Mode Recovery During Command Tag Queuing” on page 246.)

FCP Initiator-Mode Recovery When Not Command Tag Queuing

If an error such as a check condition or hardware failure occurs, the transaction active during the error is returned with the `scsi_buf.bufstruct.b_error` field set to `EIO`. Other transactions in the queue may be returned with the `scsi_buf.bufstruct.b_error` field set to `ENXIO`. If the FCP adapter driver decides not return other outstanding commands it has queued to it, then the failed transaction will be returned to the FCP device driver with an indication that the queue for this device is not cleared by setting the `SC_DID_NOT_CLEAR_Q` flag in the `scsi_buf.adap_q_status` field. The FCP device driver should process or recover the condition, rerunning any mode selects or device reservations to recover from this condition properly. After this recovery, it should reschedule the transaction that had the error. In many cases, the FCP device driver only needs to retry the unsuccessful operation.

The FCP adapter device driver should never retry a SCSI command on error after the command has successfully been given to the adapter. The consequences for retrying a FCP command at this point range from minimal to catastrophic, depending upon the type of device. Commands for certain devices cannot be

retried immediately after a failure (for example, tapes and other sequential access devices). If such an error occurs, the failed command returns an appropriate error status with an **iodone** call to the FCP device driver for error recovery. Only the FCP device driver that originally issued the command knows if the command can be retried on the device. The FCP adapter device driver must only retry commands that were never successfully transferred to the adapter. In this case, if retries are successful, the **scsi_buf** status should not reflect an error. However, the FCP adapter device driver should perform error logging on the retried condition.

The first transaction passed to the FCP adapter device driver during error recovery must include a special flag. This **SC_RESUME** flag in the **scsi_buf.flags** field must be set to inform the FCP adapter device driver that the FCP device driver has recognized the fatal error and is beginning recovery operations. Any transactions passed to the FCP adapter device driver, after the fatal error occurs and before the **SC_RESUME** transaction is issued, should be flushed; that is, returned with an error type of **ENXIO** through an **iodone** call.

Note: If a FCP device driver continues to pass transactions to the FCP adapter device driver after the FCP adapter device driver has flushed the queue, these transactions are also flushed with an error return of **ENXIO** through the **iodone** service. This gives the FCP device driver a positive indication of all transactions flushed.

FCP Initiator-Mode Recovery During Command Tag Queuing

If the FCP device driver is queuing multiple transactions to the device and either a check condition error or a command terminated error occurs, the FCP adapter driver does not clear all transactions in its queues for the device. It returns the failed transaction to the FCP device driver with an indication that the queue for this device is not cleared by setting the **SC_DID_NOT_CLEAR_Q** flag in the **scsi_buf.adap_q_status** field. The FCP adapter driver halts the queue for this device awaiting error recovery notification from the FCP device driver. The FCP device driver then has three options to recover from this error:

- Send one error recovery command (request sense) to the device.
- Clear the FCP adapter driver's queue for this device.
- Resume the FCP adapter driver's queue for this device.

When the FCP adapter driver's queue is halted, the FCP device driver can get sense data from a device by setting the **SC_RESUME** flag in the **scsi_buf.flags** field and the **SC_NO_Q** flag in **scsi_buf.q_tag_msg** field of the request-sense **scsi_buf**. This action notifies the FCP adapter driver that this is an error-recovery transaction and should be sent to the device while the remainder of the queue for the device remains halted. When the request sense completes, the FCP device driver needs to either clear or resume the FCP adapter driver's queue for this device.

The FCP device driver can notify the FCP adapter driver to clear its halted queue by sending a transaction with the **SC_Q_CLR** flag in the **scsi_buf.flags** field. This transaction must not contain a FCP command because it is cleared from the FCP adapter driver's queue without being sent to the adapter. However, this transaction must have the SCSI ID field (**scsi_buf.scsi_id**) and the LUN field (**scsi_buf.lun_id**) filled in with the device's SCSI ID and logical unit number (LUN), respectively. Upon receiving an **SC_Q_CLR** transaction, the FCP adapter driver flushes all transactions for this device and sets their **scsi_buf.bufstruct.b_error** fields to **ENXIO**. The FCP device driver must wait until the **scsi_buf** with the **SC_Q_CLR** flag set is returned before it resumes issuing transactions. The first transaction sent by the

FCP device driver after it receives the returned **SC_Q_CLR** transaction must have the **SC_RESUME** flag set in the **scsi_buf.flags** fields.

If the FCP device driver wants the FCP adapter driver to resume its halted queue, it must send a transaction with the **SC_Q_RESUME** flag set in the **scsi_buf.flags** field. This transaction can contain an actual FCP command, but it is not required. However, this transaction must have the SCSI ID field (**scsi_buf.scsi_id**) and the LUN field (**scsi_buf.lun_id**) filled in with the device's SCSI ID and logical unit number (LUN). If this is the first transaction issued by the FCP device driver after receiving the error (indicating that the adapter driver's queue is halted), then the **SC_RESUME** flag must be set as well as the **SC_Q_RESUME** flag.

Analyzing Returned Status

The following order of precedence should be followed by FCP device drivers when analyzing the returned status:

1. If the **scsi_buf.bufstruct.b_flags** field has the **B_ERROR** flag set, then an error has occurred and the **scsi_buf.bufstruct.b_error** field contains a valid **errno** value.

If the **b_error** field contains the **ENXIO** value, either the command needs to be restarted or it was canceled at the request of the FCP device driver.

If the **b_error** field contains the **EIO** value, then either one or no flag is set in the **scsi_buf.status_validity** field. If a flag is set, an error in either the **scsi_status** or **adapter_status** field is the cause.

If the **status_validity** field is 0, then the **scsi_buf.bufstruct.b_resid** field should be examined to see if the FCP command issued was in error. The **b_resid** field can have a value without an error having occurred. To decide whether an error has occurred, the FCP device driver must evaluate this field with regard to the FCP command being sent and the FCP device being driven.

If the **SC_CHECK_CONDITION** or **SC_COMMAND_TERMINATED** is set in **scsi_status**, then a FCP device driver must analyze the value of **sc_buf.scsi_fields.adap_set_flags** (i.e. **sc_buf.scsi_fields** must point to a valid **scsi3_fields** structure) to determine if autosense data was returned from the FCP device.

If the **SC_AUTOSENSE_DATA_VALID** flag is set in the **sc_buf.scsi_fields.adap_set_flags** field for a FCP device, then the FCP device returned autosense data in the buffer referenced by **sc_buf.scsi_fields.autosense_buffer_ptr**. In this situation the FCP device driver does not need to issue a SCSI request sense to determine the appropriate error recovery for the FCP devices.

If the FCP device driver is queuing multiple transactions to the device and if either **SC_CHECK_CONDITION** or **SC_COMMAND_TERMINATED** is set in **scsi_status**, then the value of **scsi_buf.adap_q_status** must be analyzed to determine if the adapter driver has cleared its queue for this device. If the FCP adapter driver has not cleared its queue after an error, then it holds that queue in a halted state.

If **scsi_buf.adap_q_status** is set to 0, the FCP adapter driver has cleared its queue for this device and any transactions outstanding are flushed back to the FCP device driver with an error of **ENXIO**.

If the **SC_DID_NOT_CLEAR_Q** flag is set in the **scsi_buf.adap_q_status** field, the adapter driver has not cleared its queue for this device. When this condition occurs, the FCP adapter driver allows the FCP device driver to send one error recovery transaction (request sense) that has the field **scsi_buf.q_tag_msg** set to **SC_NO_Q** and the field **scsi_buf.flags** set to **SC_RESUME**.

The FCP device driver can then notify the FCP adapter driver to clear or resume its queue for the device by sending a **SC_Q CLR** or **SC_Q RESUME** transaction.

If the FCP device driver does not queue multiple transactions to the device (that is, the **SC_NO_Q** is set in **scsi_buf.q_tag_msg**), then the FCP adapter clears its queue on error and sets **scsi_buf.adap_q_status** to 0.

2. If the **scsi_buf.bufstruct.b_flags** field does not have the **B_ERROR** flag set, then no error is being reported. However, the FCP device driver should examine the **b_resid** field to check for cases where less data was transferred than expected. For some FCP commands, this occurrence may not represent an error. The FCP device driver must determine if an error has occurred.

If a nonzero **b_resid** field does represent an error condition, then the device queue is not halted by the FCP adapter device driver. It is possible for one or more succeeding queued commands to be sent to the adapter (and possibly the device). Recovering from this situation is the responsibility of the FCP device driver.

3. In any of the above cases, if **scsi_buf.bufstruct.b_flags** field has the **B_ERROR** flag set, then the queue of the device in question has been halted. The first **scsi_buf** structure sent to recover the error (or continue operations) must have the **SC_RESUME** bit set in the **scsi_buf.flags** field.

A Typical Initiator-Mode FCP Driver Transaction Sequence

A simplified sequence of events for a transaction between a FCP device driver and a FCP adapter device driver follows. In this sequence, routine names preceded by a **dd_** are part of the FCP device driver, while those preceded by a **scsi_** are part of the FCP adapter device driver.

1. The FCP device driver receives a call to its **dd_strategy** routine; any required internal queuing occurs in this routine. The **dd_strategy** entry point then triggers the operation by calling the **dd_start** entry point. The **dd_start** routine invokes the **scsi_strategy** entry point by calling the **devstrategy** kernel service with the relevant **scsi_buf** structure as a parameter.
2. The **scsi_strategy** entry point initially checks the **scsi_buf** structure for validity. These checks include validating the **devno** field, matching the SCSI ID/LUN to internal tables for configuration purposes, and validating the request size.
3. Although the FCP adapter device driver cannot reorder transactions, it does perform queue chaining. If no other transactions are pending for the requested device, the **scsi_strategy** routine immediately calls the **scsi_start** routine with the new transaction. If there are other transactions pending, the new transaction is added to the tail of the device chain.
4. At each interrupt, the **scsi_intr** interrupt handler verifies the current status. The FCP adapter device driver fills in the **scsi_buf status_validity** field, updating the **scsi_status** and **adapter_status** fields as required. The FCP adapter device driver also fills in the **bufstruct.b_resid** field with the number of bytes not transferred from the request. If all the data was transferred, the **b_resid** field is set to a value of 0. If the SCSI adapter driver is a FCP adapter driver and autosense data is returned from the FCP device, then the adapter driver will also fill in the **adap_set_flags** and **autosense_buffer_ptr** fields of the **scsi_buf** structure. When a transaction completes, the **scsi_intr** routine causes the **scsi_buf** entry to be removed from the device queue and calls the **iodone** kernel service, passing the just dequeued **scsi_buf** structure for the device as the parameter. The **scsi_start** routine is then called again to process the next

transaction on the device queue. The **iodone** kernel service calls the FCP device driver **dd_iodone** entry point, signaling the FCP device driver that the particular transaction has completed.

5. The FCP device driver **dd_iodone** routine investigates the I/O completion codes in the **scsi_buf** status entries and performs error recovery, if required. If the operation completed correctly, the FCP device driver dequeues the original buffer structures. It calls the **iodone** kernel service with the original buffer pointers to notify the originator of the request.

Understanding FCP Device Driver Internal Commands

During initialization, error recovery, and open or close operations, FCP device drivers initiate some transactions not directly related to an operating system request. These transactions are called *internal commands* and are relatively simple to handle.

Internal commands differ from operating system-initiated transactions in several ways. The primary difference is that the FCP device driver is required to generate a **struct buf** that is not related to a specific request. Also, the actual FCP commands are typically more control-oriented than data transfer-related.

The only special requirement for commands with short data-phase transfers (less than or equal to 256 bytes) is that the FCP device driver must have pinned the memory being transferred into or out of system memory pages. However, due to system hardware considerations, additional precautions must be taken for data transfers into system memory pages when the transfers are larger than 256 bytes. The problem is that any system memory area with a DMA data operation in progress causes the entire memory page that contains it to become inaccessible.

As a result, a FCP device driver that initiates an internal command with more than 256 bytes must have preallocated and pinned an area of some multiple whose size is the system page size. The driver must not place in this area any other data areas that it may need to access while I/O is being performed into or out of that page. Memory pages so allocated must be avoided by the device driver from the moment the transaction is passed to the adapter device driver until the device driver **iodone** routine is called for the transaction (and for any other transactions to those pages).

Understanding the Execution of Initiator I/O Requests

During normal processing, many transactions are queued in the FCP device driver. As the FCP device driver processes these transactions and passes them to the FCP adapter device driver, the FCP device driver moves them to the in-process queue. When the FCP adapter device driver returns through the **iodone** service with one of these transactions, the FCP device driver either recovers any errors on the transaction or returns using the **iodone** kernel service to the calling level.

The FCP device driver can send only one **scsi_buf** structure per call to the FCP adapter device driver. Thus, the **scsi_buf.bufstruct.av_forw** pointer should be `null` when given to the FCP adapter device driver, which indicates that this is the only request. The FCP device driver can queue multiple **scsi_buf** requests by making multiple calls to the FCP adapter device driver strategy routine.

Spanned (Consolidated) Commands

Some kernel operations may be composed of sequential operations to a device. For example, if consecutive blocks are written to disk, blocks may or may not be in physically consecutive buffer pool blocks.

To enhance FCP transport layer performance, the FCP device driver should consolidate multiple queued requests when possible into a single FCP command. To allow the FCP adapter device driver the ability to handle the scatter and gather operations required, the `scsi_buf.bp` should always point to the first `buf` structure entry for the spanned transaction. A null-terminated list of additional `struct buf` entries should be chained from the first field through the `buf.av_forw` field to give the FCP adapter device driver enough information to perform the DMA scatter and gather operations required. This information must include at least the buffer's starting address, length, and cross-memory descriptor.

The spanned requests should always be for requests in either the read or write direction but not both, since the FCP adapter device driver must be given a single FCP command to handle the requests. The spanned request should always consist of complete I/O requests (including the additional `struct buf` entries). The FCP device driver should not attempt to use partial requests to reach the maximum transfer size.

The maximum transfer size is actually adapter-dependent. The `IOCINFO` ioctl operation can be used to discover the FCP adapter device driver's maximum allowable transfer size. To ease the design, implementation, and testing of components that may need to interact with multiple FCP-adapter device drivers, a required minimum size has been established that all FCP adapter device drivers must be capable of supporting. The value of this minimum/maximum transfer size is defined as the following value in the `/usr/include/sys/scsi_buf.h` file:

```
SC_MAXREQUEST /* maximum transfer request for a single */
               /* FCP command (in bytes)                */
```

If a transfer size larger than the supported maximum is attempted, the FCP adapter device driver returns a value of `EINVAL` in the `scsi_buf.bufstruct.b_error` field.

Due to system hardware requirements, the FCP device driver must consolidate only commands that are memory page-aligned at both their starting and ending addresses. Specifically, this applies to the consolidation of *inner* memory buffers. The ending address of the first buffer and the starting address of all subsequent buffers should be memory page-aligned. However, the starting address of the first memory buffer and the ending address of the last do not need to be aligned so.

The purpose of consolidating transactions is to decrease the number of FCP commands and transport layer phases required to perform the required operation. The time required to maintain the simple chain of `buf` structure entries is significantly less than the overhead of multiple (even two) FCP transport layer transactions.

Fragmented Commands

Single I/O requests larger than the maximum transfer size must be divided into smaller requests by the FCP device driver. For calls to a FCP device driver's character I/O (read/write) entry points, the `uphysio` kernel service can be used to break up these requests. For a *fragmented command* such as this, the `scsi_buf.bp`

field should be null so that the FCP adapter device driver uses only the information in the `scsi_buf` structure to prepare for the DMA operation.

FCP Command Tag Queuing

Note: This operation is not supported by all FCP I/O controllers.

FCP command tag queuing refers to queuing multiple commands to a FCP device. Queuing to the FCP device can improve performance because the device itself determines the most efficient way to order and process commands. FCP devices that support command tag queuing can be divided into two classes: those that clear their queues on error and those that do not. Devices that do not clear their queues on error resume processing of queued commands when the error condition is cleared (typically by receiving the next command). Devices that do clear their queues flush all commands currently outstanding.

Command tag queuing requires the FCP adapter, the FCP device, the FCP device driver, and the FCP adapter driver to support this capability. For a FCP device driver to queue multiple commands to a FCP device (that supports command tag queuing), it must be able to provide at least one of the following values in the `scsi_buf.q_tag_msg`: `SC_SIMPLE_Q`, `SC_HEAD_OF_Q`, or `SC_ORDERED_Q`. The FCP disk device driver and FCP adapter driver do support this capability. This implementation provides some queuing-specific changeable attributes for disks that can queue commands. With this information, the disk device driver attempts to queue to the disk, first by queuing commands to the adapter driver. The FCP adapter driver then queues these commands to the adapter, providing that the adapter supports command tag queuing. If the FCP adapter does not support command tag queuing, then the FCP adapter driver sends only one command at a time to the FCP adapter and so multiple commands are not queued to the FCP disk.

Understanding the `scsi_buf` Structure

The `scsi_buf` structure is used for communication between the FCP device driver and the FCP adapter device driver during an initiator I/O request. This structure is passed to and from the strategy routine in the same way a standard driver uses a `struct buf` structure.

Fields in the `scsi_buf` Structure

The `scsi_buf` structure contains certain fields used to pass a FCP command and associated parameters to the FCP adapter device driver. Other fields within this structure are used to pass returned status back to the FCP device driver. The `scsi_buf` structure is defined in the `/usr/include/sys/scsi_buf.h` file.

Fields in the `scsi_buf` structure are used as follows:

1. Reserved fields should be set to a value of 0, except where noted.
2. The `bufstruct` field contains a copy of the standard `buf` buffer structure that documents the I/O request. Included in this structure, for example, are the buffer address, byte count, and transfer direction. The `b_work` field in the `buf` structure is reserved for use by the FCP adapter device driver. The current definition of the `buf` structure is in the `/usr/include/sys/buf.h` include file.
3. The `bp` field points to the original buffer structure received by the FCP Device Driver from the caller, if any. This can be a chain of entries in the case of spanned transfers (FCP commands that transfer data from or to more than one

system-memory buffer). A null pointer indicates a nonspanned transfer. The null value specifically tells the FCP adapter device driver that all the information needed to perform the DMA data transfer is contained in the **bufstruct** fields of the **scsi_buf** structure.

4. The **scsi_command** field, defined as a **scsi_cmd** structure, contains, for example, the SCSI command length, SCSI command, and a flag variable:
 - a. The **scsi_length** field is the number of bytes in the actual SCSI command. This is normally 6, 10, 12, or 16 (decimal).
 - b. The **FCP_flags** field contains the following bit flags:

SC_NODISC

Do not allow the target to disconnect during this command.

SC_ASYNC

Do not allow the adapter to negotiate for synchronous transfer to the FCP device.

During normal use, the **SC_NODISC** bit should not be set. Setting this bit allows a device executing commands to monopolize the FCP transport layer. Sometimes it is desirable for a particular device to maintain control of the transport layer once it has successfully arbitrated for it; for instance, when this is the only device on the FCP transport layer or the only device that will be in use. For performance reasons, it may not be desirable to go through FCP selections again to save FCP transport layer overhead on each command.

Also during normal use, the **SC_ASYNC** bit must not be set. It should be set only in cases where a previous command to the device ended in an unexpected FCP transport free condition. This condition is noted as **SCSI_TRANSPORT_FAULT** in the **adapter_status** field of the **scsi_cmd** structure. Since other errors may also result in the **SCSI_TRANSPORT_FAULT** flag being set, the **SC_ASYNC** bit should only be set on the last retry of the failed command.

- c. The **scsi_cdb** structure contains the physical SCSI command block. The 6 to 16 bytes of a single SCSI command are stored in consecutive bytes, with the op code identified individually. The **scsi_cdb** structure contains the following fields:

scsi_op_code

This field specifies the standard FCP op code for this command.

scsi_bytes

This field contains the remaining command-unique bytes of the FCP command block. The actual number of bytes depends on the value in the **scsi_op_code** field.

5. The **timeout_value** field specifies the time-out limit (in seconds) to be used for completion of this command. A time-out value of 0 means no time-out is applied to this I/O request.
6. The **status_validity** field contains an output parameter that can have one of the following bit flags as a value:

SC_SCSI_ERROR

The **scsi_status** field is valid.

SC_ADAPTER_ERROR

The **adapter_status** field is valid.

7. The **scsi_status** field in the **scsi_buf** structure is an output parameter that provides valid FCP command completion status when its **status_validity** bit is nonzero. The **scsi_buf.bufstruct.b_error** field should be set to **EIO** any time the **scsi_status** field is valid. Typical status values include:

SC_GOOD_STATUS

The target successfully completed the command.

SC_CHECK_CONDITION

The target is reporting an error, exception, or other conditions.

SC_BUSY_STATUS

The target is currently transporting and cannot accept a command now.

SC_RESERVATION_CONFLICT

The target is reserved by another initiator and cannot be accessed.

SC_COMMAND_TERMINATED

The target terminated this command after receiving a terminate I/O process message from the FCP adapter.

SC_QUEUE_FULL

The target's command queue is full, so this command is returned.

SC_ACA_ACTIVE

The FCP device has an ACA (auto contingent allegiance) condition that requires a Clear ACA to request to clear it.

8. The **adapter_status** field is an output parameter that is valid when its **status_validity** bit is nonzero. The **scsi_buf.bufstruct.b_error** field should be set to **EIO** any time the **adapter_status** field is valid. This field contains generic FCP adapter card status. It is intentionally general in coverage so that it can report error status from any typical FCP adapter.

If an error is detected during execution of a FCP command, and the error prevented the FCP command from actually being sent to the FCP transport layer by the adapter, then the error should be processed or recovered, or both, by the FCP adapter device driver.

If it is recovered successfully by the FCP adapter device driver, the error is logged, as appropriate, but is not reflected in the **adapter_status** byte. If the error cannot be recovered by the FCP adapter device driver, the appropriate **adapter_status** bit is set and the **scsi_buf** structure is returned to the FCP device driver for further processing.

If an error is detected after the command was actually sent to the FCP device, then it should be processed or recovered, or both, by the FCP device driver.

For error logging, the FCP adapter device driver logs FCP transport layer and adapter-related conditions, while the FCP device driver logs FCP device-related errors. In the following description, a capital letter (A) after the error name indicates that the FCP adapter device driver handles error logging. A capital letter (H) indicates that the FCP device driver handles error logging.

Some of the following error conditions indicate a FCP device failure. Others are FCP transport layer or adapter-related.

SCSI_HOST_IO_BUS_ERR (A)

The system I/O transport layer generated or detected an error during a DMA or Programmed I/O (PIO) transfer.

SCSI_TRANSPORT_FAULT (H)

The FCP transport protocol or hardware was unsuccessful.

SCSI_CMD_TIMEOUT (H)

The command timed out before completion.

SCSI_NO_DEVICE_RESPONSE (H)

The target device did not respond to selection phase.

SCSI_ADAPTER_HDW_FAILURE (A)

The adapter indicated an onboard hardware failure.

SCSI_ADAPTER_SFW_FAILURE (A)

The adapter indicated microcode failure.

SCSI_FUSE_OR_TERMINAL_PWR (A)

The adapter indicated a blown terminator fuse or bad termination.

SCSI_TRANSPORT_RESET (A)

The adapter indicated the FCP transport layer has been reset.

SCSI_WW_NAME_CHANGE (A)

The adapter indicated the device at this SCSI ID has a new FCS world wide name.

9. The **add_status** field contains additional device status. For FCP devices, this field contains the FCP Response code returned.
10. When the FCP device driver queues multiple transactions to a device, the **adap_q_status** field indicates whether or not the FCP adapter driver has cleared its queue for this device after an error has occurred. The flag of **SC_DID_NOT_CLEAR_Q** indicates that the FCP adapter driver has not cleared its queue for this device and that it is in a halted state (so none of the pending queued transactions are sent to the device).
11. The **q_tag_msg** field indicates if the FCP adapter can attempt to queue this transaction to the device. This information causes the FCP adapter to fill in the Queue Tag Message Code of the queue tag message for a FCP command. The following values are valid for this field:

SC_NO_Q

Specifies that the FCP adapter does not send a queue tag message for this command, and so the device does not allow more than one FCP command on its command queue. This value must be used for all commands sent to FCP devices that do not support command tag queuing.

SC_SIMPLE_Q

Specifies placing this command in the device's command queue. The device determines the order that it executes commands in its queue. The SCSI-2 specification calls this value the "Simple Queue Tag Message".

SC_HEAD_OF_Q

Specifies placing this command first in the device's command queue. This command does not preempt an active command at the device, but it is executed before all other commands in the command queue. The SCSI-2 specification calls this value the "Head of Queue Tag Message".

SC_ORDERED_Q

Specifies placing this command in the device's command queue. The device processes these commands in the order that they are received. The SCSI-2 specification calls this value the "Ordered Queue Tag Message".

SC_ACA_Q

Specifies placing this command in the device's command queue, when the device has an ACA (Auto Contingent Allegiance) condition. The SCSI-3 Architecture Model calls this value the "ACA task attribute".

Note: Commands with the value of SC_NO_Q for the **q_tag_msg** field (except for request sense commands) should not be queued to a device whose queue contains a command with another value for **q_tag_msg**. If commands with the SC_NO_Q value (except for request sense) are sent to the device, then the FCP device driver must make sure that no active commands are using different values for **q_tag_msg**. Similarly, the FCP device driver must also make sure that a command with a **q_tag_msg** value of SC_ORDERED_Q, SC_HEAD_Q, or SC_SIMPLE_Q is not sent to a device that has a command with the **q_tag_msg** field of SC_NO_Q.

12. The flags field contains bit flags sent from the FCP device driver to the FCP adapter device driver. The following flags are defined:

SC_RESUME

When set, means the FCP adapter device driver should resume transaction queuing for this ID/LUN. Error recovery is complete after a **SCIOHALT** operation, check condition, or severe FCP transport error (see "SCIOHALT" on page 282). This flag is used to restart the FCP adapter device driver following a reported error.

SC_DELAY_CMD

When set, means the FCP adapter device driver should delay sending this command (following a FCP reset or BDR to this device) by at least the number of seconds specified to the FCP adapter device driver in its configuration information. For FCP devices that do not require this function, this flag should not be set.

SC_Q_CLR

When set, means the FCP adapter driver should clear its transaction queue for this ID/LUN. The transaction containing this flag setting does not require an actual FCP command in the **scsi_buf** because it is flushed back to the FCP device driver with the rest of the transactions for this ID/LUN. However, this transaction must have the SCSI ID field (**scsi_buf.scsi_id**) and the LUN field (**scsi_buf.lun_id**) filled in with the device's SCSI ID and logical unit number (LUN). This flag is valid only during error recovery of a check condition or command terminated at a command tag queuing device when the **SC_DID_NOT_CLR_Q** flag is set in the **scsi_buf.adap_q_status** field.

SC_Q_RESUME

When set, means that the FCP adapter driver should resume its halted transaction queue for this ID/LUN. The transaction containing this flag setting does not require an actual FCP command to be sent to the FCP adapter driver. However, this transaction must have the SCSI ID field (**scsi_buf.scsi_id**) and the LUN field (**scsi_buf.lun_id**) filled in with the device's SCSI ID and logical unit number (LUN). If the transaction containing this flag setting is the first issued by the FCP device driver after it receives an error (indicating that the adapter driver's queue is halted), then the **SC_RESUME** flag must be set also.

SC_CLEAR_ACA

When set, means the SCSI adapter driver should issue a Clear ACA task management request for this ID/LUN. This flag should be used in conjunction with either the **SC_Q_CLEAR** or **SC_Q_RESUME** flags to clear

or resume the SCSI adapter driver's queue for this device. If neither of these flags is used, then this transaction is treated as if the `SC_Q_RESUME` flag is also set. The transaction containing the `SC_CLEAR_ACA` flag setting does not require an actual SCSI command in the `sc_buf`. If this transaction contains a SCSI command then it will be processed depending on whether `SC_Q_CLR` or `SC_Q_RESUME` is set.

This transaction must have the SCSI ID field (`scsi_buf.scsi_id`) and the LUN field (`scsi_buf.lun_id`) filled in with the device's SCSI ID and logical unit number (LUN). This flag is valid only during error recovery of a check condition or command terminated at a command tag queuing.

13. The `dev_flags` field contains additional values sent from the FCP device driver to the FCP adapter device driver. The following values are defined:

FC_CLASS1

When set, this tells the SCSI adapter driver that it should issue this request as a Fibre Channel Class 1 request. If the SCSI adapter driver does not support this class, then it will fail the `scsi_buf` with an error of `EINVAL`. If no Fibre Channel Class is specified in the `scsi_buf` then the SCSI adapter will default to a Fibre Channel Class.

FC_CLASS2

When set, this tells the SCSI adapter driver that it should issue this request as a Fibre Channel Class 2 request. If the SCSI adapter driver does not support this class, then it will fail the `scsi_buf` with an error of `EINVAL`. If no Fibre Channel Class is specified in the `scsi_buf` then the SCSI adapter will default to a Fibre Channel Class.

FC_CLASS3

When set, this tells the SCSI adapter driver that it should issue this request as a Fibre Channel Class 3 request. If the SCSI adapter driver does not support this class, then it will fail the `scsi_buf` with an error of `EINVAL`. If no Fibre Channel Class is specified in the `scsi_buf` then the SCSI adapter will default to a Fibre Channel Class.

FC_CLASS4

When set, this tells the SCSI adapter driver that it should issue this request as a Fibre Channel Class 4 request. If the SCSI adapter driver does not support this class, then it will fail the `scsi_buf` with an error of `EINVAL`. If no Fibre Channel Class is specified in the `scsi_buf` then the SCSI adapter will default to a Fibre Channel Class.

14. The `add_work` field is reserved for use by the FCP adapter device driver.
15. The `adap_set_flags` field contains an output parameter that can have one of the following bit flags as a value:

SC_AUTOSENSE_DATA_VALID

Autosense data was placed in the autosense buffer referenced by the `autosense_buffer_ptr` field.

16. The `autosense_length` field contains the length in bytes of the SCSI device driver's sense buffer, which is referenced via the `autosense_buffer_ptr` field. For FCP devices this field must be non-zero, otherwise the autosense data will be lost.
17. The `autosense_buffer_ptr` field contains the address of the SCSI devices driver's autosense buffer for this command. For FCP devices this field must be non-NULL, otherwise the autosense data will be lost.

18. The **dev_burst_len** field contains the burst size if this write operation in bytes. This should only be set by the FCP device driver if it has negotiated with the device and it allows burst of write data without transfer ready. For most operations, this should be set to 0.
19. The **scsi_id** field contains the 64-bit SCSI ID for this device. This field must be set for FCP devices.
20. The **lun_id** field contains the 64-bit lun ID for this device. This field must be set for FCP devices.

Other FCP Design Considerations

Responsibilities of the FCP Device Driver

FCP device drivers are responsible for the following actions:

- Interfacing with block I/O and logical-volume device-driver code in the operating system.
- Translating I/O requests from the operating system into FCP commands suitable for the particular FCP device. These commands are then given to the FCP adapter device driver for execution.
- Issuing any and all FCP commands to the attached device. The FCP adapter device driver sends no FCP commands except those it is directed to send by the calling FCP device driver.
- Managing FCP device reservations and releases. In the operating system, it is assumed that other FCP initiators may be active on the FCP transport layer. Usually, the FCP device driver reserves the FCP device at open time and releases it at close time (except when told to do otherwise through parameters in the FCP device driver interface). Once the device is reserved, the FCP device driver must be prepared to reserve the FCP device again whenever a Unit Attention condition is reported through the FCP request-sense data.

FCP Options to the `openx` Subroutine

FCP device drivers in the operating system must support eight defined extended options in their open routine (see “`openx`” on page 276). Additional extended options to the open are also allowed, but they must not conflict with predefined open options. The defined extended options are bit flags in the `ext` open parameter. These options can be specified singly or in combination with each other. The required `ext` options are defined in the `/usr/include/sys/scsi.h` header file and can have one of the following values:

SC_FORCED_OPEN

Do not honor device reservation-conflict status. (See “Using the `SC_FORCED_OPEN` Option” on page 258.)

SC_RETAIN_RESERVATION

Do not release FCP device on close. (See “Using the `SC_RETAIN_RESERVATION` Option” on page 258.)

SC_DIAGNOSTIC

Enter diagnostic mode for this device. (See “Using the `SC_DIAGNOSTIC` Option” on page 258.)

SC_NO_RESERVE

Prevents the reservation of the device during an `openx` subroutine call to that device. Allows multiple hosts to share a device. (See “Using the `SC_NO_RESERVE` Option” on page 259.)

SC_SINGLE

Places the selected device in Exclusive Access mode. (See “Using the SC_SINGLE Option” on page 259.)

SC_RESV_04

Reserved for future expansion.

SC_RESV_05

Reserved for future expansion.

SC_RESV_06

Reserved for future expansion.

SC_RESV_07

Reserved for future expansion.

SC_RESV_08

Reserved for future expansion.

Using the SC_FORCED_OPEN Option

The **SC_FORCED_OPEN** option causes the FCP device driver to call the FCP adapter device driver’s transport Device Reset ioctl operation on the first open (see “SCIORESET” on page 265). This forces the device to release another initiator’s reservation. After the **SCIORESET** command is completed, other FCP commands are sent as in a normal open. If any of the FCP commands fail due to a reservation conflict, the open registers the failure as an **EBUSY** status. This is also the result if a reservation conflict occurs during a normal open. The FCP device driver should require the caller to have appropriate authority to request the **SC_FORCED_OPEN** option since this request can force a device to drop a FCP reservation. If the caller attempts to execute this system call without the proper authority, the FCP device driver should return a value of -1, with the **errno** global variable set to a value of **EPERM**.

Using the SC_RETAIN_RESERVATION Option

The **SC_RETAIN_RESERVATION** option causes the FCP device driver not to issue the FCP release command during the close of the device. This guarantees a calling program control of the device (using FCP reservation) through open and close cycles. For shared devices (for example, disk or CD-ROM), the FCP device driver must OR together this option for all opens to a given device. If any caller requests this option, the **close** routine does not issue the release even if other opens to the device do not set **SC_RETAIN_RESERVATION** (see “close” on page 276). The FCP device driver should require the caller to have appropriate authority to request the **SC_RETAIN_RESERVATION** option since this request can allow a program to monopolize a device (for example, if this is a nonshared device). If the caller attempts to execute this system call without the proper authority, the FCP device driver should return a value of -1, with the **errno** global variable set to a value of **EPERM**.

Using the SC_DIAGNOSTIC Option

The **SC_DIAGNOSTIC** option causes the FCP device driver to enter Diagnostic mode for the given device. This option directs the FCP device driver to perform only minimal operations to open a logical path to the device. No FCP commands should be sent to the device in the **open** (see “open” on page 276) or **close** (see “close” on page 276) routine when the device is in Diagnostic mode. One or more ioctl operations should be provided by the FCP device driver to allow the caller to issue FCP commands to the attached device for diagnostic purposes.

The **SC_DIAGNOSTIC** option gives the caller an exclusive open to the selected device. This option requires appropriate authority to execute. If the caller attempts to execute this system call without the proper authority, the FCP device driver should return a value of -1, with the **errno** global variable set to a value of **EPERM**. The **SC_DIAGNOSTIC** option may be executed only if the device is not already opened for normal operation. If this ioctl operation is attempted when the device is already opened, or if an **openx** (see “openx” on page 276) call with the **SC_DIAGNOSTIC** option is already in progress, a return value of -1 should be passed, with the **errno** global variable set to a value of **EACCES**. Once successfully opened with the **SC_DIAGNOSTIC** flag, the FCP device driver is placed in Diagnostic mode for the selected device.

Using the **SC_NO_RESERVE** Option

The **SC_NO_RESERVE** option causes the FCP device driver not to issue the FCP reserve command during the opening of the device and not to issue the FCP release command during the close of the device. This allows multiple hosts to share the device. The FCP device driver should require the caller to have appropriate authority to request the **SC_NO_RESERVE** option, since this request allows other hosts to modify data on the device. If a caller does this kind of request then the caller must ensure data integrity between multiple hosts. If the caller attempts to execute this system call without the proper authority, the FCP device driver should return a value of -1, with the **errno** global variable set to a value of **EPERM**.

Using the **SC_SINGLE** Option

The **SC_SINGLE** option causes the FCP device driver to issue a normal open, but does not allow another caller to issue another open until the first caller has closed the device. This request gives the caller an exclusive open to the selected device. If this **openx** routine is attempted when the device is already open, a return value of -1 is passed, with the **errno** global variable set to a value of **EBUSY** (see “openx” on page 276).

Once successfully opened, the device is placed in Exclusive Access mode. If another caller tries to do any type of **open** routine (see “open” on page 276), a return value of -1 is passed, with the **errno** global variable set to a value of **EACCES**.

The remaining options for the *ext* parameter are reserved for future requirements.

Note: The following table shows how the various combinations of *ext* options should be handled in the FCP device driver.

EXT OPTIONS	Device Driver Action	
openx ext option	Open	Close
none	normal	normal
diag	no FCP commands	no FCP commands
diag + force	issue SCIORESET; otherwise, no FCP commands issued	no FCP commands
diag + force + no_reserve	issue SCIORESET; otherwise, no FCP commands issued	no FCP commands

EXT OPTIONS	Device Driver Action	
openx <i>ext</i> option	Open	Close
diag + force + no_reserve + single	issue SCIORESET; otherwise, no FCP commands issued.	no FCP commands
diag + force + retain	issue SCIORESET; otherwise, no FCP commands issued	no FCP commands
diag + force + retain + no_reserve	issue SCIORESET; otherwise, no FCP commands issued	no FCP commands
diag + force + retain + no_reserve + single	issue SCIORESET; otherwise, no FCP commands issued	no FCP commands
diag + force + retain + single	issue SCIORESET; otherwise, no FCP commands issued	no FCP commands
diag + force + single	issue SCIORESET; otherwise, no FCP commands issued	no FCP commands
diag + no_reserve	no FCP commands	no FCP commands
diag + retain	no FCP commands	no FCP commands
diag + retain + no_reserve	no FCP commands	no FCP commands
diag + retain + no_reserve + single	no FCP commands	no FCP commands
diag + retain + single	no FCP commands	no FCP commands
diag + single	no FCP commands	no FCP commands
diag + single + no_reserve	no FCP commands	no FCP commands
force	normal, except SCIORESET issued prior to any FCP commands.	normal
force + no_reserve	normal, except SCIORESET issued prior to any FCP commands. No RESERVE command issued	normal except no RELEASE
force + retain	normal, except SCIORESET issued prior to any FCP commands	no RELEASE
force + retain + no_reserve	normal except SCIORESET issued prior to any FCP commands. No RESERVE command issued.	no RELEASE
force + retain + no_reserve + single	normal, except SCIORESET issued prior to any FCP commands. No RESERVE command issued.	no RELEASE
force + retain + single	normal, except SCIORESET issued prior to any FCP commands.	no RELEASE

EXT OPTIONS	Device Driver Action	
openx <i>ext</i> option	Open	Close
force + single	normal, except SCIORESET issued prior to any FCP commands.	normal
force + single + no_reserve	normal, except SCIORESET issued prior to any FCP commands. No RESERVE command issued	no RELEASE
no_reserve	no RESERVE	no RELEASE
retain	normal	no RELEASE
retain + no_reserve	no RESERVE	no RELEASE
retain + single	normal	no RELEASE
retain + single + no_reserve	normal, except no RESERVE command issued	no RELEASE
single	normal	normal
single + no_reserve	no RESERVE	no RELEASE

Closing the FCP Device

When a FCP device driver is preparing to close a device through the FCP adapter device driver, it must ensure that all transactions are complete. When the FCP adapter device driver receives a **SCIOLSTOP** ioctl operation and there are pending I/O requests, the ioctl operation does not return until all have completed (see “SCIOLSTOP” on page 264). New requests received during this time are rejected from the adapter device driver’s **ddstrategy** routine.

FCP Error Processing

It is the responsibility of the FCP device driver to process FCP check conditions and other returned errors properly. The FCP adapter device driver only passes FCP commands without otherwise processing them and is not responsible for device error recovery.

Length of Data Transfer for FCP Commands

Commands initiated by the FCP device driver internally or as subordinates to a transaction from above must have data phase transfers of 256 bytes or less to prevent DMA/CPU memory conflicts. The length indicates to the FCP adapter device driver that data phase transfers are to be handled internally in its address space. This is required to prevent DMA/CPU memory conflicts for the FCP device driver. The FCP adapter device driver specifically interprets a byte count of 256 or less as an indication that it can not perform data-phase DMA transfers directly to or from the destination buffer.

The actual DMA transfer goes to a dummy buffer inside the FCP adapter device driver and then is block-copied to the destination buffer. Internal FCP device driver operations that typically have small data-transfer phases are FCP control-type commands, such as Mode select, Mode sense, and Request sense. However, this discussion applies to any command received by the FCP adapter device driver that has a data-phase size of 256 bytes or less.

Internal commands with data phases larger than 256 bytes require the FCP device driver to allocate specifically the required memory on the process level. The memory pages containing this memory cannot be accessed in any way by the CPU (that is, the FCP device driver) from the time the transaction is passed to the FCP adapter device driver until the FCP device driver receives the `iodone` call for the transaction.

Device Driver and Adapter Device Driver Interfaces

The FCP device drivers can have both character (raw) and block special files in the `/dev` directory. The FCP adapter device driver has only character (raw) special files in the `/dev` directory and has only the `ddconfig`, `ddopen`, `ddclose`, `dddump`, and `ddioctl` entry points available to operating system programs. The `ddread` and `ddwrite` entry points are not implemented.

Internally, the `devsw` table has entry points for the `ddconfig`, `ddopen`, `ddclose`, `dddump`, `ddioctl`, and `ddstrat` routines. The FCP device drivers pass their FCP commands to the FCP adapter device driver by calling the FCP adapter device driver `ddstrat` routine. (This routine is unavailable to other operating system programs due to the lack of a block-device special file.)

Access to the FCP adapter device driver's `ddconfig`, `ddopen`, `ddclose`, `dddump`, `ddioctl`, and `ddstrat` entry points by the FCP device drivers is performed through the kernel services provided. These include such services as `fp_opendev`, `fp_close`, `fp_ioctl`, `devdump`, and `devstrat`.

Performing FCP Dumps

A FCP adapter device driver must have a `dddump` entry point if it is used to access a system dump device. A FCP device driver must have a `dddump` entry point if it drives a dump device. Examples of dump devices are disks and tapes.

Note: FCP adapter-device-driver writers should be aware that system services providing interrupt and timer services are unavailable for use in the dump routine. Kernel DMA services are assumed to be available for use by the dump routine. The FCP adapter device driver should be designed to ignore extra `DUMPINIT` and `DUMPSTART` commands to the `dddump` entry point.

The `DUMPQUERY` option should return a minimum transfer size of 0 bytes, and a maximum transfer size equal to the maximum transfer size supported by the FCP adapter device driver.

Calls to the FCP adapter device driver `DUMPWRITE` option should use the `arg` parameter as a pointer to the `scsi_buf` structure to be processed. Using this interface, a FCP `write` command can be executed on a previously started (opened) target device. The `uiop` parameter is ignored by the FCP adapter device driver during the `DUMPWRITE` command. Spanned, or consolidated, commands are not supported using the `DUMPWRITE` option. Gathered `write` commands are also not supported using the `DUMPWRITE` option. No queuing of `scsi_buf` structures is supported during dump processing since the dump routine runs essentially as a subroutine call from the caller's dump routine. Control is returned when the entire `scsi_buf` structure has been processed.

Warning: Also, both adapter-device-driver and device-driver writers should be aware that any error occurring during the `DUMPWRITE` option is

considered unsuccessful. Therefore, no error recovery is employed during the **DUMPWRITE**. Return values from the call to the **dddump** routine indicate the specific nature of the failure.

Successful completion of the selected operation is indicated by a 0 return value to the subroutine. Unsuccessful completion is indicated by a return code set to one of the following values for the **errno** global variable. The various **scsi_buf** status fields, including the **b_error** field, are not set by the FCP adapter device driver at completion of the **DUMPWRITE** command. Error logging is, of necessity, not supported during the dump.

- An **errno** value of **EINVAL** indicates that a request that was not valid passed to the FCP adapter device driver, such as to attempt a **DUMPSTART** command before successfully executing a **DUMPINIT** command.
- An **errno** value of **EIO** indicates that the FCP adapter device driver was unable to complete the command due to a lack of required resources or an I/O error.
- An **errno** value of **ETIMEDOUT** indicates that the adapter did not respond with completion status before the passed command time-out value expired.

Required FCP Adapter Device Driver ioctl Commands

Description

Various ioctl operations must be performed for proper operation of the FCP adapter device driver. The ioctl operations described here are the minimum set of commands the FCP adapter device driver must implement to support FCP device drivers. Other operations may be required in the FCP adapter device driver to support, for example, system management facilities and diagnostics. FCP device driver writers also need to understand these ioctl operations.

Every FCP adapter device driver must support the **IOCINFO** ioctl operation. The structure to be returned to the caller is the **devinfo** structure, including the FCP union definition for the FCP adapter, which can be found in the **/usr/include/sys/devinfo.h** file. The FCP device driver should request the **IOCINFO** ioctl operation (probably during its open routine) to get the maximum transfer size of the adapter.

Note: The FCP adapter device driver ioctl operations can only be called from the process level. They cannot be executed from a call on any more favored priority levels. Attempting to call them from a more favored priority level can result in a system crash.

Initiator-Mode ioctl Commands

The following **SCIOLSTART** and **SCIOLSTOP** operations must be sent by the FCP device driver (for the open and close routines, respectively) for each device (see “**SCIOLSTART**” on page 264 or “**SCIOLSTOP**” on page 264). They cause the FCP adapter device driver to allocate and initialize internal resources. The **SCIOLHALT** ioctl operation is used to abort pending or running commands, usually after signal processing by the FCP device driver (see “**SCIOLHALT**” on page 282). This might be used by a FCP device driver to end an operation instead of waiting for completion or a time out. The **SCIOLRESET** operation is provided for clearing device hard errors and competing initiator reservations during open processing by the FCP device driver. (See “**SCIOLRESET**” on page 265.)

Except where noted otherwise, the *arg* parameter for each of the ioctl operations described here must contain a long integer. In this field, the least significant byte is

the FCP LUN and the next least significant byte is the FCP ID value. (The upper two bytes are reserved and should be set to 0.) This provides the information required to allocate or deallocate resources and perform FCP transport layer operations for the `ioctl` operation requested.

The following information is provided on the various `ioctl` operations:

SCIOLSTART

This operation allocates and initializes FCP device-dependent information local to the FCP adapter device driver. Run this operation only on the first open of an ID/LUN device. Subsequent **SCIOLSTART** commands to the same ID/LUN fail unless an intervening **SCIOLSTOP** command is issued.

For this operation an `scsi_sciolst` structure (The `scsi_sciolst` structure is defined in the `/usr/include/sys/scsi_buf.h` file.) must be used to specify the FCP device's SCSI id and LUN id. In addition, the `scsi_sciolst` structure can be used to specify an explicit FCP process login for this operation.

The following values for the `errno` global variable are supported:

0	Indicates successful completion.
EIO	Indicates lack of resources or other error-preventing device allocation.
EINVAL	Indicates that the selected SCSI ID and LUN are already in use, or the SCSI ID matches the adapter ID.
ENOMEM	Indicates that system resources are not available to start this device.
ETIMEDOUT	Indicates that the command did not complete.
ENODEV	Indicates that no FCP device responded to the explicit process login at this SCSI ID.
ECONNREFUSED	Indicates that the FCP device at this SCSI ID rejected explicit process login. This could be due to the device rejecting the security password or the device does not support FCP.

SCIOLSTOP

This operation deallocates resources local to the FCP adapter device driver for this FCP device. This should be run on the last close of an ID/LUN device. If an **SCIOLSTART** operation has not been previously issued, this command is unsuccessful. For this operation an `scsi_sciolst` structure (The `scsi_sciolst` structure is defined in the `/usr/include/sys/scsi_buf.h` file.) must be used to specify the FCP device's SCSI id and LUN id. In addition the `scsi_sciolst` structure can be used to specify an explicit FCP process login for this operation.

The following values for the `errno` global variable should be supported:

0	Indicates successful completion.
EIO	Indicates error preventing device deallocation.
EINVAL	Indicates that the selected FCP ID and LUN have not been started.
ETIMEDOUT	Indicates that the command did not complete.

SCIOLHALT

This operation halts outstanding transactions to this ID/LUN device and causes the FCP adapter device driver to stop accepting transactions for this device. This situation remains in effect until the FCP device driver sends another transaction with the `SC_RESUME` flag set (in the `scsi_buf.flags` field) for this ID/LUN combination. The **SCIOLHALT** `ioctl` operation causes the FCP adapter device driver to fail the command in progress, if any, as well as all queued commands for the device with a return value of `ENXIO` in the `scsi_buf.bufstruct.b_error` field. If

an **SCIOSTART** operation has not been previously issued, this command fails. For this operation an `scsi_sciolst` structure (The `scsi_sciolst` structure is defined in the `/usr/include/sys/scsi_buf.h` file.) must be used to specify the FCP device's SCSI id and LUN id. In addition the `scsi_sciolst` structure can be used to specify an explicit FCP process login for this operation.

The following values for the `errno` global variable are supported:

0	Indicates successful completion.
EIO	Indicates an unrecovered I/O error occurred.
EINVAL	Indicates that the selected FCP ID and LUN have not been started.
ETIMEDOUT	Indicates that the command did not complete.

SCIORESET

This operation causes the FCP adapter device driver to send a FCP transport Device Reset (BDR) message to the selected FCP ID. For this operation, the FCP device driver should set the LUN in the `arg` parameter to the LUN ID of a LUN on this FCP ID, which has been successfully started using the **SCIOSTART** operation. For this operation an `scsi_sciolst` structure (The `scsi_sciolst` structure is defined in the `/usr/include/sys/scsi_buf.h` file.) must be used to specify the FCP device's SCSI id and LUN id. In addition the `scsi_sciolst` structure can be used to specify an explicit FCP process login for this operation.

The FCP device driver should use this command only when directed to do a *forced open*. This occurs in two possible situations: one, when it is desirable to force the device to drop a FCP reservation; two, when the device needs to be reset to clear an error condition (for example, when running diagnostics on this device).

Note: In normal system operation, this command should not be issued, as it would force the device to drop a FCP reservation another initiator (and, hence, another system) may have. If an **SCIOSTART** operation has not been previously issued, this command is unsuccessful.

The following values for the `errno` global variable are supported:

0	Indicates successful completion.
EIO	Indicates an unrecovered I/O error occurred.
EINVAL	Indicates that the selected FCP ID and LUN have not been started.
ETIMEDOUT	Indicates that the command did not complete.

SCIOLCMD

This operation provides the means for issuing any SCSI command to the specified device. The SCSI adapter driver performs no error recovery or logging on failures of this `ioctl` operation.

The following values for the `errno` global variable are supported:

0	Indicates successful completion.
---	----------------------------------

EIO	A system error has occurred. Retry the operation (about three times). If an EIO error occurs and the status_validity field is set to SC_SCSI_ERROR, then the scsi_status field has a valid value and should be inspected. If the status_validity field is zero and remains so on successive retries, then an unrecoverable error has occurred with the device. If the status_validity field is SC_SCSI_ERROR and the scsi_status field contains a Check Condition status, then a SCSI request sense should be issued via the SCIOLCMD ioctl to recover the the sense data.
EFAULT	A user process copy has failed.
EINVAL	The device is not opened.
EACCES	The adapter is in diagnostics mode.
ENOMEM	A memory request has failed.
ETIMEDOUT	The command has timed out. Retry the operation.
ENODEV	The device is not responding.
ETIMEDOUT	The operation did not complete before the time-out value was exceeded.

Initiator-Mode ioctl Command used by FCP Device Drivers

SCIOLEVENT

For initiator mode, the FCP device driver may issue an **SCIOLEVENT** ioctl operation to register for receiving asynchronous event status from the FCP adapter device driver for a particular device instance. This is an optional call for the FCP device driver, and is optionally supported for the FCP adapter device driver. A failing return code from this command, in the absence of any programming error, indicates it is not supported. If the FCP device driver requires this function, it must check the return code to verify the FCP adapter device driver supports it.

Only a kernel process or device driver can invoke these ioctls. If attempted by a user process, the ioctl will fail, and the **errno** global variable will be set to EPERM.

The event registration performed by this ioctl operation is allowed once per device session. Only the first **SCIOLEVENT** ioctl operation is accepted after the device session is opened. Succeeding **SCIOLEVENT** ioctl operations will fail, and the **errno** global variable will be set to EINVAL. The event registration is canceled automatically when the device session is closed.

The *arg* parameter to the **SCIOLEVENT** ioctl operation should be set to the address of an **scsi_event_struct** structure, which is defined in the **/usr/include/sys/scsi_buf.h** file. The following parameters are supported:

- id* The caller sets *id* to the FCP ID of the attached FCP target device for initiator-mode. For target-mode, the caller sets the *id* to the FCP ID of the attached FCP initiator device.
- lun* The caller sets the **lun** field to the FCP LUN of the attached FCP target device for initiator-mode. For target-mode, the caller sets the **lun** field to 0.
- mode* Identifies whether the initiator-mode or target-mode device is being registered. These values are possible:

SC_IM_MODE

This is an initiator-mode device.

SC_TM_MODE

This is a target-mode device.

async_correlator

The caller places a value in this optional field which is saved by the FCP adapter device driver and returned when an event occurs in this field in the **scsi_event_info** structure. This structure is defined in the **/user/include/sys/scsi_buf.h**.

async_func

The caller fills in the address of a pinned routine which the FCP adapter device driver calls whenever asynchronous event status is available. The FCP adapter device driver passes a pointer to a **scsi_event_info** structure to the caller's **async_func** routine.

Note: All reserved fields should be set to 0 by the caller.

The following values for the **errno** global variable are supported:

0	Indicates successful completion.
EINVAL	An SCIOSTART has not been issued to this device instance, or this device is already registered for async events.
EPERM	Indicates the caller is not running in kernel mode, which is the only mode allowed to execute this operation.

Chapter 14. FCP Device Drivers

Programming FCP Device Drivers

The AIX Fibre Channel Protocol for SCSI (FCP) subsystem has two parts:

- FCP Device Driver
- FCP Adapter Device Driver

The FCP adapter device driver is designed to shield you from having to communicate directly with the system I/O hardware. This gives you the ability to successfully write a FCP device driver without having a detailed knowledge of the system hardware. You can look at the FCP subsystem as a two-tiered structure in which the adapter device driver is the bottom or supporting layer. As a programmer, you need only worry about the upper layer. This chapter only discusses writing a FCP device driver, because the FCP adapter device driver is already provided in AIX.

The FCP adapter device driver, or lower layer, is responsible only for the communications to and from the FCP bus, and any error logging and recovery. The upper layer is responsible for all of the device-specific commands. The FCP device driver should handle all commands directed towards its specific device by building the necessary sequence of I/O requests to the FCP adapter device driver in order to properly communicate with the device.

These I/O requests contain the FCP commands that are needed by the FCP device. One important aspect to note is that the FCP device driver cannot access any of the adapter resources and should never try to pass the FCP commands directly to the adapter, since it has absolutely no knowledge of the meaning of those commands.

FCP Device Driver Overview

The role of the FCP device driver is to pass information between the operating system and the FCP adapter device driver by accepting I/O requests and passing these requests to the FCP adapter device driver. The device driver should accept either character or block I/O requests, build the necessary FCP commands, and then issue these commands to the device through the FCP adapter device driver.

The FCP device driver should also process the various required reservations and releases needed for the device. The device driver is notified through the **iodone** kernel service once the adapter has completed the processing of the command. The device driver should then notify its calling process that the request has completed processing through the **iodone** kernel service.

FCP Adapter Device Driver Overview

Unlike most other device drivers, the FCP adapter device driver does *not* support the **read** and **write** subroutines. It only supports the **open**, **close**, **ioctl**, **config**, and **strategy** subroutines. Included with the **open** subroutine call is the **openx** subroutine that allows FCP adapter diagnostics.

A FCP device driver does not need to access the FCP diagnostic commands. Commands received from the device driver through the **strategy** routine of the

adapter are processed from a queue. Once the command has completed, the device driver is notified through the **iodone** kernel service.

FCP Adapter/Device Interface

The AIX FCP adapter device driver does not contain the **ddread** and **ddwrite** entry points, but does contain the **ddconfig**, **ddopen**, **ddclose**, **dddump**, and **ddioctl** entry points.

Therefore, the adapter device driver's entry in the kernel devsw table contains only those entries plus an additional **ddstrategy** entry point. This **ddstrategy** routine is the path that the FCP device driver uses to pass commands to the device driver. Access to these entry points is possible through the following kernel services:

- **fp_open**
- **fp_close**
- **devdump**
- **fp_ioctl**
- **devstrat**

The FCP adapter is accessed by the device driver through the **/dev/fscsi#** special files, where # indicates ascending numbers 0, 1, 2, and so on. The adapter is designed so that multiple devices on the same adapter can be accessed at the same time.

For additional information on spanned and gathered write commands, see *Understanding the Execution of Initiator I/O Requests*.

scsi_buf Structure

The I/O requests made from the FCP device driver to the FCP adapter device driver are completed through the use of the **scsi_buf** structure, which is defined in the **/usr/include/sys/scsi_buf.h** header file. This structure, which is similar to the **buf** structure in other drivers, is passed between the two FCP subsystem drivers through the **strategy** routine (see "strategy" on page 277). The following is a brief description of the fields contained in the **scsi_buf** structure:

1. Reserved fields should be set to a value of 0, except where noted.
2. The **bufstruct** field contains a copy of the standard **buf** buffer structure that documents the I/O request. Included in this structure, for example, are the buffer address, byte count, and transfer direction. The **b_work** field in the **buf** structure is reserved for use by the FCP adapter device driver. The current definition of the **buf** structure is in the **/usr/include/sys/buf.h** include file.
3. The **bp** field points to the original buffer structure received by the FCP Device Driver from the caller, if any. This can be a chain of entries in the case of spanned transfers (FCP commands that transfer data from or to more than one system-memory buffer). A null pointer indicates a nonspanned transfer. The null value specifically tells the FCP adapter device driver that all the information needed to perform the DMA data transfer is contained in the **bufstruct** fields of the **scsi_buf** structure.
4. The **scsi_command** field, defined as a **scsi_cmd** structure, contains, for example, the SCSI command length, SCSI command, and a flag variable:
 - a. The **scsi_length** field is the number of bytes in the actual SCSI command. This is normally 6, 10, 12, or 16 (decimal).
 - b. The **FCP_flags** field contains the following bit flags:

SC_NODISC

Do not allow the target to disconnect during this command.

SC_ASYNC

Do not allow the adapter to negotiate for synchronous transfer to the FCP device.

During normal use, the **SC_NODISC** bit should not be set. Setting this bit allows a device executing commands to monopolize the FCP transport layer. Sometimes it is desirable for a particular device to maintain control of the transport layer once it has successfully arbitrated for it; for instance, when this is the only device on the FCP transport layer or the only device that will be in use. For performance reasons, it may not be desirable to go through FCP selections again to save FCP transport layer overhead on each command.

Also during normal use, the **SC_ASYNC** bit must not be set. It should be set only in cases where a previous command to the device ended in an unexpected FCP transport free condition. This condition is noted as **SCSI_TRANSPORT_FAULT** in the **adapter_status** field of the **scsi_cmd** structure. Since other errors may also result in the **SCSI_TRANSPORT_FAULT** flag being set, the **SC_ASYNC** bit should only be set on the last retry of the failed command.

- c. The **scsi_cdb** structure contains the physical SCSI command block. The 6 to 16 bytes of a single SCSI command are stored in consecutive bytes, with the op code identified individually. The **scsi_cdb** structure contains the following fields:
 - 1) The **scsi_op_code** field specifies the standard FCP op code for this command.
 - 2) The **scsi_bytes** field contains the remaining command-unique bytes of the FCP command block. The actual number of bytes depends on the value in the **scsi_op_code** field.
5. The **timeout_value** field specifies the time-out limit (in seconds) to be used for completion of this command. A time-out value of 0 means no time-out is applied to this I/O request.
6. The **status_validity** field contains an output parameter that can have one of the following bit flags as a value:

SC_SCSI_ERROR

The **scsi_status** field is valid.

SC_ADAPTER_ERROR

The **adapter_status** field is valid.

7. The **scsi_status** field in the **scsi_buf** structure is an output parameter that provides valid FCP command completion status when its **status_validity** bit is nonzero. The **scsi_buf.bufstruct.b_error** field should be set to EIO anytime the **scsi_status** field is valid. Typical status values include:

SC_GOOD_STATUS

The target successfully completed the command.

SC_CHECK_CONDITION

The target is reporting an error, exception, or other conditions.

SC_BUSY_STATUS

The target is currently transporting and cannot accept a command now.

SC_RESERVATION_CONFLICT

The target is reserved by another initiator and cannot be accessed.

SC_COMMAND_TERMINATED

The target terminated this command after receiving a terminate I/O process message from the FCP adapter.

SC_QUEUE_FULL

The target's command queue is full, so this command is returned.

SC_ACA_ACTIVE

The FCP device has an ACA (auto contingent allegiance) condition that requires a Clear ACA to request to clear it.

8. The **adapter_status** field is an output parameter that is valid when its **status_validity** bit is nonzero. The **scsi_buf.bufstruct.b_erro** field should be set to E10 anytime the **adapter_status** field is valid. This field contains generic FCP adapter card status. It is intentionally general in coverage so that it can report error status from any typical FCP adapter.

If an error is detected during execution of a FCP command, and the error prevented the FCP command from actually being sent to the FCP transport layer by the adapter, then the error should be processed or recovered, or both, by the FCP adapter device driver.

If it is recovered successfully by the FCP adapter device driver, the error is logged, as appropriate, but is not reflected in the **adapter_status** byte. If the error cannot be recovered by the FCP adapter device driver, the appropriate **adapter_status** bit is set and the **scsi_buf** structure is returned to the FCP device driver for further processing.

If an error is detected after the command was actually sent to the FCP device, then it should be processed or recovered, or both, by the FCP device driver.

For error logging, the FCP adapter device driver logs FCP transport layer and adapter-related conditions, while the FCP device driver logs FCP device-related errors. In the following description, a capital letter (A) after the error name indicates that the FCP adapter device driver handles error logging. A capital letter (H) indicates that the FCP device driver handles error logging.

Some of the following error conditions indicate a FCP device failure. Others are FCP transport layer or adapter-related.

SCSI_HOST_IO_BUS_ERR (A)

The system I/O transport layer generated or detected an error during a DMA or Programmed I/O (PIO) transfer.

SCSI_TRANSPORT_FAULT (H)

The FCP transport protocol or hardware was unsuccessful.

SCSI_CMD_TIMEOUT (H)

The command timed out before completion.

SCSI_NO_DEVICE_RESPONSE (H)

The target device did not respond to selection phase.

SCSI_ADAPTER_HDW_FAILURE (A)

The adapter indicated an onboard hardware failure.

SCSI_ADAPTER_SFW_FAILURE (A)

The adapter indicated microcode failure.

SCSI_FUSE_OR_TERMINAL_PWR (A)

The adapter indicated a blown terminator fuse or bad termination.

SCSI_TRANSPORT_RESET (A)

The adapter indicated the FCP transport layer has been reset.

SCSI_WW_NAME_CHANGE (A)

The adapter indicated the device at this SCSI ID has a new FCS world wide name.

9. The **add_status** field contains additional device status. For FCP devices, this field contains the FCP Response code returned.
10. When the FCP device driver queues multiple transactions to a device, the **adap_q_status** field indicates whether or not the FCP adapter driver has cleared its queue for this device after an error has occurred. The flag of **SC_DID_NOT_CLEAR_Q** indicates that the FCP adapter driver has not cleared its queue for this device and that it is in a halted state (so none of the pending queued transactions are sent to the device).
11. The **q_tag_msg** field indicates if the FCP adapter can attempt to queue this transaction to the device. This information causes the FCP adapter to fill in the Queue Tag Message Code of the queue tag message for a FCP command. The following values are valid for this field:

SC_NO_Q

Specifies that the FCP adapter does not send a queue tag message for this command, and so the device does not allow more than one FCP command on its command queue. This value must be used for all commands sent to FCP devices that do not support command tag queuing.

SC_SIMPLE_Q

Specifies placing this command in the device's command queue. The device determines the order that it executes commands in its queue. The SCSI-2 specification calls this value the Simple Queue Tag Message.

SC_HEAD_OF_Q

Specifies placing this command first in the device's command queue. This command does not preempt an active command at the device, but it is executed before all other commands in the command queue. The SCSI-2 specification calls this value the Head of Queue Tag Message.

SC_ORDERED_Q

Specifies placing this command in the device's command queue. The device processes these commands in the order that they are received. The SCSI-2 specification calls this value the Ordered Queue Tag Message.

SC_ACA_Q

Specifies placing this command in the device's command queue, when the device has an ACA (auto contingent allegiance) condition. The SCSI-3 Architecture Model calls this value the ACA task attribute.

Note: Commands with the value of **SC_NO_Q** for the **q_tag_msg** field (except for request sense commands) should not be queued to a device whose queue contains a command with another value for **q_tag_msg**. If commands with the **SC_NO_Q** value (except for request sense) are sent to the device, then the FCP device driver must make sure that no active commands are using different values for **q_tag_ms**. Similarly, the FCP device driver must also make sure that a command with a **q_tag_msg** value of **SC_ORDERED_Q**, **SC_HEAD_Q**, or **SC_SIMPLE_Q** is not sent to a device that has a command with the **q_tag_msg** field of **SC_NO_Q**.

12. The `flags` field contains bit flags sent from the FCP device driver to the FCP adapter device driver. The following flags are defined:

SC_RESUME

When set, means the FCP adapter device driver should resume transaction queuing for this ID/LUN. Error recovery is complete after a **SCIOHALT** operation, check condition, or severe FCP transport error. This flag is used to restart the FCP adapter device driver following a reported error.

SC_DELAY_CMD

When set, means the FCP adapter device driver should delay sending this command (following a FCP reset or BDR to this device) by at least the number of seconds specified to the FCP adapter device driver in its configuration information. For FCP devices that do not require this function, this flag should not be set.

SC_Q_CLR

When set, means the FCP adapter driver should clear its transaction queue for this ID/LUN. The transaction containing this flag setting does not require an actual FCP command in the `scsi_buf` because it is flushed back to the FCP device driver with the rest of the transactions for this ID/LUN. However, this transaction must have the SCSI ID field (`scsi_buf.scsi_id`) and the LUN field (`scsi_buf.lun_id`) filled in with the device's SCSI ID and logical unit number (LUN). This flag is valid only during error recovery of a check condition or command terminated at a command tag queuing device when the `SC_DID_NOT_CLR_Q` flag is set in the `scsi_buf.adap_q_status` field.

SC_Q_RESUME

When set, means that the FCP adapter driver should resume its halted transaction queue for this ID/LUN. The transaction containing this flag setting does not require an actual FCP command to be sent to the FCP adapter driver. However, this transaction must have the SCSI ID field (`scsi_buf.scsi_id`) and the LUN field (`scsi_buf.lun_id`) filled in with the device's SCSI ID and logical unit number (LUN). If the transaction containing this flag setting is the first issued by the FCP device driver after it receives an error (indicating that the adapter driver's queue is halted), then the `SC_RESUME` flag must be set also.

SC_CLEAR_ACA

When set, means the SCSI adapter driver should issue a Clear ACA task management request for this ID/LUN. This flag should be used in conjunction with either the `SC_Q_CLEAR` or `SC_Q_RESUME` flags to clear or resume the SCSI adapter driver's queue for this device. If neither of these flags is used, then this transaction is treated as if the `SC_Q_RESUME` flag is also set. The transaction containing the `SC_CLEAR_ACA` flag setting does not require an actual SCSI command in the `sc_buf`. If this transaction contains a SCSI command then it will be processed depending on whether `SC_Q_CLR` or `SC_Q_RESUME` is set. This transaction must have the SCSI ID field (`scsi_buf.scsi_id`) and the LUN field (`scsi_buf.lun_id`) filled in with the device's SCSI ID and logical unit number (LUN). This flag is valid only during error recovery of a check condition or command terminated at a command tag queuing.

13. The `dev_flags` field contains additional values sent from the FCP device driver to the FCP adapter device driver. The following values are defined:

FC_CLASS1

When set, this tells the SCSI adapter driver that it should issue this request as a Fibre Channel Class 1 request. If the SCSI adapter driver does not support this class, then it will fail the **scsi_buf** with an error of EINVAL. If no Fibre Channel Class is specified in the **scsi_buf** then the SCSI adapter will default to a Fibre Channel Class.

FC_CLASS2

When set, this tells the SCSI adapter driver that it should issue this request as a Fibre Channel Class 2 request. If the SCSI adapter driver does not support this class, then it will fail the **scsi_buf** with an error of EINVAL. If no Fibre Channel Class is specified in the **scsi_buf** then the SCSI adapter will default to a Fibre Channel Class.

FC_CLASS3

When set, this tells the SCSI adapter driver that it should issue this request as a Fibre Channel Class 3 request. If the SCSI adapter driver does not support this class, then it will fail the **scsi_buf** with an error of EINVAL. If no Fibre Channel Class is specified in the **scsi_buf** then the SCSI adapter will default to a Fibre Channel Class.

FC_CLASS4

When set, this tells the SCSI adapter driver that it should issue this request as a Fibre Channel Class 4 request. If the SCSI adapter driver does not support this class, then it will fail the **scsi_buf** with an error of EINVAL. If no Fibre Channel Class is specified in the **scsi_buf** then the SCSI adapter will default to a Fibre Channel Class.

14. The **add_work** field is reserved for use by the FCP adapter device driver.
15. The **adap_set_flags** field contains an output parameter that can have one of the following bit flags as a value:

SC_AUTOSENSE_DATA_VALID

Autosense data was placed in the autosense buffer referenced by the **autosense_buffer_ptr** field.

16. The **autosense_length** field contains the length in bytes of the SCSI device driver's sense buffer, which is referenced via the **autosense_buffer_ptr** field. For FCP devices this field must be non-zero, otherwise the autosense data will be lost.
17. The **autosense_buffer_ptr** field contains the address of the SCSI devices driver's autosense buffer for this command. For FCP devices this field must be non-NULL, otherwise the autosense data will be lost.
18. The **dev_burst_len** field contains the burst size if this write operation in bytes. This should only be set by the FCP device driver if it has negotiated with the device and it allows burst of write data without transfer ready's. For most operations, this should be set to 0.
19. The **scsi_id** field contains the 64-bit SCSI ID for this device. This field must be set for FCP devices.
20. The **lun_id** field contains the 64-bit lun ID for this device. This field must be set for FCP devices.

Adapter/Device Driver Intercommunication

In a typical request to the device driver, a call is first made to the device driver's **strategy** routine, which takes care of any necessary queuing. The device driver's **strategy** routine then calls the device driver's **start** routine, which fills in the **scsi_buf** structure and calls the adapter driver's **strategy** routine through the **devstrat** kernel service.

The adapter driver's **strategy** routine validates all of the information contained in the **scsi_buf** structure and also performs any necessary queuing of the transaction request. If no queuing is necessary, the adapter driver's **start** subroutine is called.

When an interrupt occurs, the FCP adapter **interrupt** routine fills in the **status_validity** field and the appropriate **scsi_status** or **adapter_status** field of the **scsi_buf** structure. The **bufstruct.b_resid** field is also filled in with the value of nontransferred bytes. The adapter driver's **interrupt** routine then passes this newly filled in **scsi_buf** structure to the **iodone** kernel service which then signals the FCP device driver's **iodone** subroutine. The adapter driver's **start** routine is also called from the **interrupt** routine to process any additional transactions on the queue.

The device driver's **iodone** routine should then process all of the applicable fields in the queued **scsi_buf** structure for any errors and attempt error recovery if necessary. The device driver should then dequeue the **scsi_buf** structure and then pass a pointer to the structure back to the **iodone** kernel service so that it can notify the originator of the request.

FCP Adapter Device Driver Routines

config

The **config** routine performs all of the processing needed to configure, unconfigure, and read Vital Product Data (VPD) for the FCP adapter. When this routine is called to configure an adapter, it performs the required checks and building of data structures needed to prepare the adapter for the processing of requests.

When asked to unconfigure or terminate an adapter, this routine deallocates any structures defined for the adapter and marks the adapter as unconfigured. This routine can also be called to return the Vital Product Data for the adapter, which contains information that is used to identify the serial number, change level, or part number of the adapter.

open

The **open** routine establishes a connection between a special file and a file descriptor. This file descriptor is the link to the special file that is the access point to a device and is used by all subsequent calls to perform I/O requests to the device. Interrupts are enabled and any data structures needed by the adapter driver are also initialized.

close

The **close** routine marks the adapter as closed and disables all future interrupts, which causes the driver to reject all future requests to this adapter.

openx

The **openx** routine allows a process with the proper authority to open the adapter in diagnostic mode. If the adapter is already open in either normal or diagnostic mode, the **openx** subroutine has a return value of -1. Improper authority results in an **errno** value of **EPERM**, while an already open error results in an **errno** value of **EACCES**. If the adapter is in diagnostic mode, only the **close** and **ioctl** routines are allowed. All other routines return a value of -1 and an **errno** value of **EACCES**.

While in diagnostics mode, the adapter can run diagnostics, run wrap tests, and download microcode. The **openx** routine is called with an *ext* parameter that contains the adapter mode and the SC_DIAGNOSTIC value, both of which are defined in the `sys/scsi.h` header file.

strategy

The **strategy** routine is the link between the device driver and the FCP adapter device driver for all normal I/O requests. Whenever the FCP device driver receives a call, it builds an **scsi_buf** structure with the correct parameters and then passes it to this routine, which in turn queues up the request if necessary. Each request on the pending queue is then processed by building the necessary FCP commands required to carry out the request. When the command has completed, the FCP device driver is notified through the **iodone** kernel service.

ioctl

The **ioctl** routine allows various diagnostic and nondiagnostic adapter operations. Operations include the following:

- **IOCINFO**
- **SCIOLSTART**
- **SCIOLSTOP**
- **SCIOLINQU**
- **SCIOLEVENT**
- **SCIOLSTUNIT**
- **SCIOLTUR**
- **SCIOLREAD**
- **SCIOLRESET**
- **SCIOLHALT**
- **SCIOLCMD**

start

The **start** routine is responsible for checking all pending queues and issuing commands to the adapter. When a command is issued to the adapter, the **scsi_buf** is converted into an adapter specific request needed for the **scsi_buf**. At this time, the **bufstruct.b_addr** for the **scsi_buf** will be mapped for DMA. When the adapter specific request is completed, the adapter will be notified of this request.

interrupt

The **interrupt** routine is called whenever the adapter posts an interrupt. When this occurs, the interrupt routine will find the **scsi_buf** corresponding to this interrupt. The buffer for the **scsi_buf** will be unmapped from DMA. If an error occurred, the **status_validity**, **scsi_status**, and **adapter_status** fields will be set accordingly. The **bufstruct.b_resid** field will be set with the number of nontransferred bytes. The interrupt handler will then **iodone** this **scsi_buf**, which will send the **scsi_buf** back to the device driver which originated it.

FCP Adapter ioctl Operations

IOCINFO

This operation allows a FCP device driver to obtain important information about a FCP adapter, including the adapter's SCSI ID, the maximum data transfer size in

bytes, and the FC topology to which the adapter is connected. By knowing the maximum data transfer size, a FCP device driver can control several different devices on several different adapters. This operation returns a **devinfo** structure as defined in the **sys/devinfo.h** header file with the device type **DD_BUS** and subtype **DS_FCP**. The following is an example of a call to obtain the information:

```
rc = fp_ioctl(fp, IOCINFO, &infostruct, NULL);
```

where *fp* is a pointer to a file structure and *infostruct* is a **devinfo** structure. A non-zero rc value indicates an error. Note that the **devinfo** structure is a union of several structures and that **fcp** is the structure that applies to the adapter.

For example, the maximum transfer size value is contained in the variable **infostruct.un.fcp.max_transfer** and the card ID is contained in **infostruct.un.fcp.scsi_id**.

SCIOLSTART

This operation opens a logical path to the FCP device and causes the FCP adapter device driver to allocate and initialize all of the data areas needed for the FCP device. The SCIOLSTOP operation should be issued when those data areas are no longer needed. This operation should be issued before any nondiagnostic operation except for IOCINFO. The following is a typical call:

```
rc = fp_ioctl(fp, SCIOLSTART, &sciolst);
```

where *fp* is a pointer to a file structure and *sciolst* is a **scsi_sciolst** structure (defined in **/usr/include/sys/scsi_buf.h**) that contains the SCSI and Logical Unit Number (LUN) ID values of the device to be started. In addition the **scsi_sciolst** structure can be used to specify an explicit FCP process login for this operation.

A nonzero return value indicates an error has occurred and all operations to this SCSI/LUN pair should cease since the device is either already started or failed the start operation. Possible **errno** values are

EIO	The command could not complete due to a system error.
EINVAL	Either the Logical Unit Number (LUN) ID or SCSI ID is invalid, or the adapter is already open.
ENOMEM	Indicates that system resources are not available to start this device.
ETIMEDOUT	Indicates that the command did not complete.
ENODEV	Indicates that no FCP device responded to the explicit process login at this SCSI ID.
ECONNREFUSED	Indicates that the FCP device at this SCSI ID rejected explicit process login. This could be due to the device rejecting the security password or the device does not support FCP.
EACCES	The adapter is not in normal mode.

SCIOLSTOP

This operation closes a logical path to the FCP device and causes the FCP adapter device driver to deallocate all data areas that were allocated by the SCIOLSTART operation. This operation should only be issued after a successful SCIOLSTART operation to a device. The following is a typical call:

```
rc = fp_ioctl(fp, SCIOLSTOP, &sciolst);
```

where *fp* is a pointer to a file structure and *sciolst* is a **scsi_sciolst** structure (defined in **/usr/include/sys/scsi_buf.h**) that contains the SCSI and Logical Unit Number (LUN) ID values of the device to be started.

A non-zero return value indicates an error has occurred. Possible **errno** values are:

EIO	An unrecoverable system error has occurred.
EINVAL	The adapter was not in open mode.

This operation requires **SCIOSTART** to be run first.

SCIOLEVENT

This operation allows a FCP device driver to register a particular device instance for receiving asynchronous event status by calling the **SCIOLEVENT** ioctl operation for the FCP-adapter device driver. When an event covered by the **SCIOLEVENT** ioctl operation is detected by the FCP adapter device driver, it builds an **scsi_event_info** structure and passes a pointer to the structure and to the asynchronous event-handler routine entry point, which was previously registered.

The information reported in the **scsi_event_info.events** field does not queue to the FCP device driver, but is instead reported as one or more flags as they occur. Since the data does not queue, the FCP adapter device driver writer can use a single **scsi_event_info** structure and pass it one at a time, by pointer, to each asynchronous event handler routine for the appropriate device instance. After determining for which device the events are being reported, the FCP device driver must copy the **scsi_event_info.events** field into local space and must not modify the contents of the rest of the **scsi_event_info** structure.

Since the event status is optional, the FCP device driver writer determines what action is necessary to take upon receiving event status. The writer may decide to save the status and report it back to the calling application, or the FCP device driver or application level program can take error recovery actions.

This operation should only be issued after a successful **SCIOSTART** operation to a device. The following is a typical call:

```
rc = fp_ioctl(fp, SCIOLEVENT, &scevent);
```

where *fp* is a pointer to a file structure and *scevent* is a **scsi_event_struct** structure (defined in **/usr/include/sys/scsi_buf.h**) that contains the SCSI and Logical Unit Number (LUN) ID values of the device to be started.

A non-zero return value indicates an error has occurred. Possible **errno** values are:

EIO	An unrecoverable system error has occurred.
EINVAL	The adapter was not in open mode.

This operation requires **SCIOSTART** to be run first.

SCIOLINQU

This operation issues an inquiry command to a FCP device and is used to aid in FCP device configuration. The following is a typical call:

```
rc = ioctl(adapter, SCIOLINQU, &inquiry_block);
```

where *adapter* is a file descriptor and *inquiry_block* is a **scsi_inquiry** structure as defined in the **/usr/include/sys/scsi_buf.h** header file. The FCP ID and LUN should be placed in the **scsi_inquiry** parameter block. The **SC_ASYNC** flag should not be set on the first call to this operation and should only be set if a bus fault has

occurred. Possible **errno** values are:

EIO	A system error has occurred. Retry the operation.
EFAULT	A user process copy has failed.
EINVAL	The device is not opened.
EACCES	The adapter is in diagnostics mode.
ENOMEM	A memory request has failed.
ETIMEDOUT	The command has timed out. Retry the operation.
ENODEV	The device is not responding. Possibly no LUNs exist on the present FCP ID.
ENOCONNECT	A bus fault has occurred and the operation should be retried with the SC_ASYNC flag set in the scsi_inquiry structure. In the case of multiple retries, this flag should be set only on the last retry.

This operation requires **SCIOLSTART** to be run first.

SCIOLSTUNIT

This operation issues a start unit command to a FCP device and is used to aid in FCP device configuration. The following is a typical call:

```
rc = ioctl(adapter, SCIOLSTUNIT, &start_block);
```

where *adapter* is a file descriptor and *start_block* is a **scsi_startunit** structure as defined in the **/usr/include/sys/scsi_buf.h** header file. The FCP ID and LUN should be placed in the **scsi_startunit** parameter block. The **start_flag** field designates the start option, which when set to true, makes the device available for use. When this field is set to false, the device is stopped.

The **SC_ASYNC** flag should not be set on the first call to this operation and should only be set if a bus fault has occurred. The **immed_flag** field allows overlapping start operations to several devices on the FCP bus. When this field is set to false, status is returned only when the operation has completed. When this field is set to true, status is returned as soon as the device receives the command. The **SCIOLTUR** operation can then be issued to check on completion of the operation on a particular device.

Note that when the FCP adapter is allowed to issue simultaneous start operations, it is important that a delay of 10 seconds be allowed between successive **SCIOLSTUNIT** operations to devices sharing a common power supply since damage to the system or devices can occur if this precaution is not followed. Possible **errno** values are:

EIO	A system error has occurred. Retry the operation.
EFAULT	A user process copy has failed.
EINVAL	The device is not opened.
EACCES	The adapter is in diagnostics mode.
ENOMEM	A memory request has failed.
ETIMEDOUT	The command has timed out. Retry the operation.
ENODEV	The device is not responding. Possibly no LUNs exist on the present FCP ID.
ENOCONNECT	A bus fault has occurred. Try the operation again with the SC_ASYNC flag set in the scsi_inquiry structure. In the case of multiple retries, this flag should be set only on the last retry.

This operation requires **SCIOLSTART** to be run first.

SCIOLTUR

This operation issues a FCP Test Unit Ready command to an adapter and aids in FCP device configuration. The following is a typical call:

```
rc = ioctl(adapter, SCIOLTUR, &ready_struct);
```

where *adapter* is a file descriptor and *ready_struct* is a **scsi_ready** structure as defined in the **/usr/include/sys/scsi_buf.h** header file. The FCP ID and LUN should be placed in the **scsi_ready** parameter block. The **SC_ASYNC** flag should not be set on the first call to this operation and should only be set if a bus fault has occurred. The status of the device can be determined by evaluating the two output fields: **status_validity** and **scsi_status**. Possible **errno** values are:

EIO	A system error has occurred. Retry the operation. If an EIO error occurs and the status_validity field is set to SC_FCP_ERROR , then the scsi_status field has a valid value and should be inspected. If the status_validity field is zero and remains so on successive retries, then an unrecoverable error has occurred with the device. If the status_validity field is SC_FCP_ERROR and the scsi_status field contains a Check Condition status, then the SCIOLTUR operation should be retried after several seconds. If after successive retries, the Check Condition status remains, the device should be considered inoperable.
EFAULT	A user process copy has failed.
EINVAL	The device is not opened.
EACCES	The adapter is in diagnostics mode.
ENOMEM	A memory request has failed.
ETIMEDOUT	The command has timed out. Retry the operation.
ENODEV	The device is not responding and possibly no LUNs exist on the present FCP ID.
ENOCONNECT	A bus fault has occurred and the operation should be retried with the SC_ASYNC flag set in the scsi_inquiry structure. In the case of multiple retries, this flag should be set only on the last retry.

This operation requires **SCIOLSTART** to be run first.

SCIOLREAD

This operation issues an read command to a FCP device and is used to aid in FCP device configuration. The following is a typical call:

```
rc = ioctl(adapter, SCIOLREAD, &readblk);
```

where *adapter* is a file descriptor and *readblk* is a **scsi_readblk** structure as defined in the **/usr/include/sys/scsi_buf.h** header file. The FCP ID and LUN should be placed in the **scsi_readblk** parameter block. The **SC_ASYNC** flag should not be set on the first call to this operation and should only be set if a bus fault has occurred. Possible **errno** values are:

EIO	A system error has occurred. Retry the operation.
EFAULT	A user process copy has failed.
EINVAL	The device is not opened.
EACCES	The adapter is in diagnostics mode.
ENOMEM	A memory request has failed.
ETIMEDOUT	The command has timed out. Retry the operation.

ENODEV	The device is not responding. Possibly no LUNs exist on the present FCP ID.
ENOCNECT	A bus fault has occurred and the operation should be retried with the SC_ASYNC flag set in the scsi_readblk structure. In the case of multiple retries, this flag should be set only on the last retry.

This operation requires **SCIOSTART** to be run first.

SCIORESET

This operation causes a FCP device to release all reservations, clear all current commands, and return to an initial state by issuing a Bus Device Reset (BDR) to all LUNs associated with the specified FCP ID. A FCP reserve command should be issued after the **SCIORESET** operation to prevent other initiators from claiming the device. Note that because a certain amount of time exists between a reset and reserve command, it is still possible for another initiator to successfully reserve a particular device. The following is a typical call:

```
rc = fp_ioctl(fp, SCIORESET, &sciolst);
```

where *fp* is a pointer to a file structure and *sciolst* is a **scsi_sciolst** structure (defined in **/usr/include/sys/scsi_buf.h**) that contains the SCSI and Logical Unit Number (LUN) ID values of the device to be started.

A nonzero return value indicates an error has occurred. Possible **errno** values are:

EIO	An unrecoverable system error has occurred.
EINVAL	The device is not opened.
EACCES	The adapter is in diagnostics mode.
ETIMEDOUT	The operation did not complete before the time-out value was exceeded.

This operation requires **SCIOSTART** to be run first.

SCIOHALT

This operation stops the current command of the selected device, clears the command queue of any pending commands, and brings the device to a halted state. The FCP adapter sends a FCP abort message to the device and is usually used by the FCP device driver to abort the current operation instead of allowing it to complete or time out.

After the **SCIOHALT** operation is sent, the device driver must set the **SC_RESUME** flag in the next **scsi_buf** structure sent to the adapter device driver, or all subsequent **scsi_buf** structures sent are ignored.

The FCP adapter also performs normal error recovery procedures during this command which include issuing a FCP bus reset in response to a FCP bus hang. The following is a typical call:

```
rc = fp_ioctl(fp, SCIOHALT, &sciolst);
```

where *fp* is a pointer to a file structure and *sciolst* is a **scsi_sciolst** structure (defined in **/usr/include/sys/scsi_buf.h**) that contains the SCSI and Logical Unit Number (LUN) ID values of the device to be started.

A nonzero return value indicates an error has occurred. Possible **errno** values are:

EIO	An unrecoverable system error has occurred.
EINVAL	The device is not opened.
EACCES	The adapter is in diagnostics mode.
ETIMEDOUT	The operation did not complete before the time-out value was exceeded.

This operation requires **SCIOSTART** to be run first.

SCIOLCMD

When the SCSI device has been successfully started (**SCIOSTART**), this operation provides the means for issuing any SCSI command to the specified device. The SCSI adapter driver performs no error recovery or logging on failures of this `ioctl` operation. The following is a typical call:

```
rc = ioctl(adapter, SCIOLCMD, &iocmd);
```

where *adapter* is a file descriptor and *iocmd* is a **scsi_iocmd** structure as defined in the `/usr/include/sys/scsi_buf.h` header file. The SCSI ID and LUN should be placed in the **scsi_iocmd** parameter block.

The SCSI status byte and the adapter status bytes are returned via the **scsi_iocmd** structure. If the **SCIOLCMD** operation returns a value of -1 and the **errno** global variable is set to a nonzero value, the requested operation has failed. In this case, the caller should evaluate the returned status bytes to determine why the operation failed and what recovery actions should be taken.

The **devinfo** structure defines the maximum transfer size for the command. If an attempt is made to transfer more than the maximum, a value of -1 is returned and the **errno** global variable set to a value of **EINVAL**. Refer to the *Small Computer System Interface (SCSI) Specification* for the applicable device to get request sense information.

Possible **errno** values are:

EIO	A system error has occurred. Retry the operation. If an EIO error occurs and the status_validity field is set to <code>SC_SCSI_ERROR</code> , then the scsi_status field has a valid value and should be inspected. If the status_validity field is zero and remains so on successive retries then an unrecoverable error has occurred with the device. If the status_validity field is <code>SC_SCSI_ERROR</code> and the scsi_status field contains a Check Condition status, then a SCSI request sense should be issued via the SCIOLCMD <code>ioctl</code> to recover the the sense data.
EFAULT	A user process copy has failed.
EINVAL	The device is not opened.
EACCES	The adapter is in diagnostics mode.
ENOMEM	A memory request has failed.
ETIMEDOUT	The command has timed out. Retry the operation.
ENODEV	The device is not responding.
ETIMEDOUT	The operation did not complete before the time-out value was exceeded.

This operation requires **SCIOLSTART** to be run first. (See “SCIOLSTART” on page 264 .)

Chapter 15. Integrated Device Electronics (IDE) Subsystem

This overview describes the interface between an Integrated Device Electronics (IDE) device driver and an IDE adapter device driver. It is directed toward those designing and writing an IDE device driver that interfaces with an existing IDE adapter device driver. It is also meant for those designing and writing an IDE adapter device driver that interfaces with existing IDE device drivers.

The main topics covered in this overview are:

- Responsibilities of the IDE Adapter Device Driver
- “Responsibilities of the IDE Device Driver”
- “Communication Between IDE Device Drivers and IDE Adapter Device Drivers” on page 286

This section frequently refers to both an IDE device driver and an IDE adapter device driver. These two distinct device drivers work together in a layered approach to support attachment of a range of IDE devices. The IDE adapter device driver is the lower device driver of the pair, and the IDE device driver is the upper device driver.

Responsibilities of the IDE Adapter Device Driver

The IDE adapter device driver (the lower layer) is the software interface to the system hardware. This hardware includes the IDE bus hardware plus any other system I/O hardware required to run an I/O request. The IDE adapter device driver hides the details of the I/O hardware from the IDE device driver. The design of the software interface allows a user with limited knowledge of the system hardware to write the upper device driver.

The IDE adapter device driver manages the IDE bus, but not the IDE devices. It can send and receive IDE commands, but it cannot interpret the contents of the command. The lower driver also provides recovery and logging for errors related to the IDE bus and system I/O hardware. Management of the device specifics is left to the IDE device driver. The interface of the two drivers allows the upper driver to communicate with different IDE bus adapters without requiring special code paths for each adapter.

Responsibilities of the IDE Device Driver

The IDE device driver (the upper layer) provides the rest of the operating system with the software interface to a given IDE device or device class. The upper layer recognizes which IDE commands are required to control a particular IDE device or device class. The IDE device driver builds I/O requests containing device IDE commands and sends them to the IDE adapter device driver in the sequence needed to operate the device successfully. The IDE device driver cannot manage adapter resources or give the IDE command to the adapter. Specifics about the adapter and system hardware are left to the lower layer.

The IDE device driver also provides recovery and logging for errors related to the IDE device it controls.

The operating system provides several kernel services allowing the IDE device driver to communicate with IDE adapter device driver entry points without having the actual name or address of those entry points. See “Logical File System Kernel Services” on page 51 for more information.

Communication Between IDE Device Drivers and IDE Adapter Device Drivers

The interface between the IDE device driver and the IDE adapter device driver is accessed through calls to the IDE adapter device driver **open**, **close**, **ioctl**, and **strategy** routines. I/O requests are queued to the IDE adapter device driver through calls to its strategy entry point.

Communication between the IDE device driver and the IDE adapter device driver for a particular I/O request is made through the **ataide_buf** structure, which is passed to and from the **strategy** routine in the same way a standard driver uses a **struct buf** structure. The **ataide_buf.ata** structure represents the ATA or ATAPI command that the adapter driver must send to the specified IDE device. The **ataide_buf.status_validity** field and the **ataide_buf.ata** structure contain completion status returned to the IDE device driver.

IDE Error Recovery

If an error, such as a check condition or hardware failure occurs, the transaction active during the error is returned with the **ataide_buf.bufstruct.b_error** field set to **EIO**. The IDE device driver should process or recover the condition, rerunning any mode selects to recover from this condition properly. After this recovery, it should reschedule the transaction that had the error. In many cases, the IDE device driver only needs to retry the unsuccessful operation.

The IDE adapter device driver should never retry an IDE command on error after the command has successfully been given to the adapter. The consequences for retrying an IDE command at this point range from minimal to catastrophic, depending upon the type of device. Commands for certain devices cannot be retried immediately after a failure (for example, tapes and other sequential access devices). If such an error occurs, the failed command returns an appropriate error status with an **iodone** call to the IDE device driver for error recovery. Only the IDE device driver that originally issued the command knows if the command can be retried on the device. The IDE adapter device driver must only retry commands that were never successfully transferred to the adapter. In this case, if retries are successful, the **ataide_buf** status should not reflect an error. However, the IDE adapter device driver should perform error logging on the retried condition.

Analyzing Returned Status

The following order of precedence should be followed by IDE device drivers when analyzing the returned status:

1. If the **ataide_buf.bufstruct.b_flags** field has the **B_ERROR** flag set, then an error has occurred and the **ataide_buf.bufstruct.b_error** field contains a valid **errno** value.

If the **b_error** field contains the **ENXIO** value, either the command needs to be restarted or it was canceled at the request of the IDE device driver.

If the `b_error` field contains the **EIO** value, then either one or no flag is set in the `ataide_buf.status_validity` field. If a flag is set, an error in either the `ata.status` or `ata.errval` field is the cause.

If the `status_validity` field is 0, then the `ataide_buf.bufstruct.b_resid` field should be examined to see if the IDE command issued was in error. The `b_resid` field can have a value without an error having occurred. To decide whether an error has occurred, the IDE device driver must evaluate this field with regard to the IDE command being sent and the IDE device being driven.

2. If the `ataide_buf.bufstruct.b_flags` field does not have the **B_ERROR** flag set, then no error is being reported. However, the IDE device driver should examine the `b_resid` field to check for cases where less data was transferred than expected. For some IDE commands, this occurrence may not represent an error. The IDE device driver must determine if an error has occurred.

There is a special case when `b_resid` will be nonzero. The DMA service routine may not be able to map all virtual to real memory pages for a single DMA transfer. This may occur when sending close to the maximum amount of data that the adapter driver supports. In this case, the adapter driver transfers as much of the data that can be mapped by the DMA service. The unmapped size is returned in the `b_resid` field, and the `status_validity` will have the `ATA_IDE_DMA_NORES` bit set. The IDE device driver is expected to send the data represented by the `b_resid` field in a separate request.

If a nonzero `b_resid` field does represent an error condition, then the device queue is not halted by the IDE adapter device driver. It is possible for one or more succeeding queued commands to be sent to the adapter (and possibly the device). Recovering from this situation is the responsibility of the IDE device driver.

A Typical IDE Driver Transaction Sequence

A simplified sequence of events for a transaction between an IDE device driver and an IDE adapter device driver follows. In this sequence, routine names preceded by a `dd_` are part of the IDE device driver, while those preceded by an `ide_` are part of the IDE adapter device driver.

1. The IDE device driver receives a call to its **dd_strategy** routine; any required internal queuing occurs in this routine. The **dd_strategy** entry point then triggers the operation by calling the **dd_start** entry point. The **dd_start** routine invokes the **ide_strategy** entry point by calling the **devstrat** kernel service with the relevant **ataide_buf** structure as a parameter.
2. The **ide_strategy** entry point initially checks the **ataide_buf** structure for validity. These checks include validating the `devno` field, matching the IDE device ID to internal tables for configuration purposes, and validating the request size.
3. The IDE adapter device driver does not queue transactions. Only a single transaction is accepted per device (one master, one slave). If no transaction is currently active, the **ide_strategy** routine immediately calls the **ide_start** routine with the new transaction. If there is a current transaction for the same device, the new transaction is returned with an error indicated in the **ataide_buf** structure. If there is a current transaction for the other device, the new transaction is queued to the inactive device.
4. At each interrupt, the **ide_intr interrupt** handler verifies the current status. The IDE adapter device driver fills in the `ataide_buf` `status_validity` field, updating the `ata.status` and `ata.errval` fields as required. The IDE adapter device driver also fills in the `bufstruct.b_resid` field with the number of bytes

not transferred from the request. If all the data was transferred, the `b_resid` field is set to a value of 0. When a transaction completes, the `ide_intr` routine causes the `ataide_buf` entry to be removed from the device queue and calls the `iodone` kernel service, passing the just dequeued `ataide_buf` structure for the device as the parameter. The `ide_start` routine is then called again to process the next transaction on the device queue. The `iodone` kernel service calls the IDE device driver `dd_iodone` entry point, signaling the IDE device driver that the particular transaction has completed.

5. The IDE device driver `dd_iodone` routine investigates the I/O completion codes in the `ataide_buf` status entries and performs error recovery, if required. If the operation completed correctly, the IDE device driver dequeues the original buffer structures. It calls the `iodone` kernel service with the original buffer pointers to notify the originator of the request.

IDE Device Driver Internal Commands

During initialization, error recovery, and open or close operations, IDE device drivers initiate some transactions not directly related to an operating system request. These transactions are called internal commands and are relatively simple to handle.

Internal commands differ from operating system-initiated transactions in several ways. The primary difference is that the IDE device driver is required to generate a **struct buf** that is not related to a specific request. Also, the actual IDE commands are typically more control oriented than data transfer-related.

The only special requirement for commands is that the IDE device driver must have pinned the memory transferred into or out of system memory pages. However, due to system hardware considerations, additional precautions must be taken for data transfers into system memory pages. The problem is that any system memory area with a DMA data operation in progress causes the entire memory page that contains it to become inaccessible.

As a result, an IDE device driver that initiates an internal command must have preallocated and pinned an area of some multiple whose size is the system page size. The driver must not place in this area any other data areas that it may need to access while I/O is being performed into or out of that page. Memory pages allocated must be avoided by the device driver from the moment the transaction is passed to the adapter device driver until the device driver `iodone` routine is called for the transaction (and for any other transactions to those pages).

Execution of I/O Requests

During normal processing, many transactions are queued in the IDE device driver. As the IDE device driver processes these transactions and passes them to the IDE adapter device driver, the IDE device driver moves them to the in-process queue. When the IDE adapter device driver returns through the `iodone` service with one of these transactions, the IDE device driver either recovers any errors on the transaction or returns using the `iodone` kernel service to the calling level.

The IDE device driver can send only one `ataide_buf` structure per call to the IDE adapter device driver. Thus, the `ataide_buf.bufstruct.av_forw` pointer should be

null when given to the IDE adapter device driver, which indicates that this is the only request. The IDE adapter driver does not support queuing multiple requests to the same device.

Spanned (Consolidated) Commands

Some kernel operations may be composed of sequential operations to a device. For example, if consecutive blocks are written to disk, blocks may or may not be in physically consecutive buffer pool blocks.

To enhance IDE bus performance, the IDE device driver should consolidate multiple queued requests when possible into a single IDE command. To allow the IDE adapter device driver the ability to handle the scatter and gather operations required, the `ataide_buf.bp` should always point to the first `buf` structure entry for the spanned transaction. A null-terminated list of additional `struct buf` entries should be chained from the first field through the `buf.av_forw` field to give the IDE adapter device driver enough information to perform the DMA scatter and gather operations required. This information must include at least the buffer's starting address, length, and cross-memory descriptor.

The spanned requests should always be for requests in either the read or write direction but not both, since the IDE adapter device driver must be given a single IDE command to handle the requests. The spanned request should always consist of complete I/O requests (including the additional `struct buf` entries). The IDE device driver should not attempt to use partial requests to reach the maximum transfer size.

The maximum transfer size is actually adapter-dependent. The `IOCINFO ioctl` operation can be used to discover the IDE adapter device driver's maximum allowable transfer size. To ease the design, implementation, and testing of components that may need to interact with multiple IDE-adapter device drivers, a required minimum size has been established that all IDE adapter device drivers must be capable of supporting. The value of this minimum/maximum transfer size is defined as the following value in the `/usr/include/sys/ide.h` file:

```
IDE_MAXREQUEST /* maximum transfer request for a single IDE command (in bytes) */
```

If a transfer size larger than the supported maximum is attempted, the IDE adapter device driver returns a value of `EINVAL` in the `ataide_buf.bufstruct.b_error` field.

Due to system hardware requirements, the IDE device driver must consolidate only commands that are memory page-aligned at both their starting and ending addresses. Specifically, this applies to the consolidation of inner memory buffers. The ending address of the first buffer and the starting address of all subsequent buffers should be memory page-aligned. However, the starting address of the first memory buffer and the ending address of the last do not need to be aligned.

The purpose of consolidating transactions is to decrease the number of IDE commands and bus phases required to perform the required operation. The time required to maintain the simple chain of `buf` structure entries is significantly less than the overhead of multiple (even two) IDE bus transactions.

Fragmented Commands

Single I/O requests larger than the maximum transfer size must be divided into smaller requests by the IDE device driver. For calls to an IDE device driver's character I/O (read/write) entry points, the **uphysio** kernel service can be used to break up these requests. For a fragmented command such as this, the `ataide_buf.bp` field should be NULL so that the IDE adapter device driver uses only the information in the `ataide_buf` structure to prepare for the DMA operation.

Gathered Write Commands

The gathered write commands facilitate communications applications that are required to send header and trailer messages with data buffers. These headers and trailers are typically the same or similar for each transfer. Therefore, there may be a single copy of these messages but multiple data buffers.

The gathered write commands, accessed through the `ataide_buf.sg_ptr` field, differ from the spanned commands, accessed through the `ataide_buf.bp` field, in several ways:

- Gathered write commands can transfer data regardless of address alignment, while spanned commands must be memory page-aligned in address and length, making small transfers difficult.
- Gathered write commands can be implemented either in software (which requires the extra step of copying the data to temporary buffers) or hardware. Spanned commands can be implemented in system hardware due to address-alignment requirements. As a result, spanned commands are potentially faster to run.
- Gathered write commands are not able to handle read requests. Spanned commands can handle both read and write requests.
- Gathered write commands can be initiated only on the process level, but spanned commands can be initiated on either the process or interrupt level.

To execute a gathered write command, the IDE device driver must:

- Fill in the `sg_ptr` field with a pointer to the `uio` struct.
- Call the IDE adapter device driver on the same process level with the `ataide_buf` structure in question.
- Be attempting a write.
- Not have put a non-null value in the `ataide_buf.bp` field.

If any of these conditions are not met, the gather write commands do not succeed and the `ataide_buf.bufstruct.b_error` is set to **EINVAL**.

This interface allows the IDE adapter device driver to perform the gathered write commands in both software or hardware as long as the adapter supports this capability. Because the gathered write commands can be performed in software (by using such kernel services as **uiomove**), the contents of the `sg_ptr` field and the `uio` struct can be altered. Therefore, the caller must restore the contents of both the `sg_ptr` field and the `uio` struct before attempting a retry. Also, the retry must occur from the process level; it must not be performed from the caller's **iodone** subroutine.

To support IDE adapter device drivers that perform the gathered write commands in software, additional return values in the `ataide_buf.bufstruct.b_error` field are

possible when gathered write commands are unsuccessful.

ENOMEM	Error due to lack of system memory to perform copy.
EFAULT	Error due to memory copy problem.

Note: The gathered write command facility is optional for both the IDE device driver and the IDE adapter device driver. Attempting a gathered write command to a IDE adapter device driver that does not support gathered write can cause a system crash. Therefore, any IDE device driver must issue an **IDEIOGTHW** ioctl operation to the IDE adapter device driver before using gathered writes. An IDE adapter device driver that supports gathered writes must support the **IDEIOGTHW** ioctl as well. The ioctl returns a successful return code if gathered writes are supported. If the ioctl fails, the IDE device driver must not attempt a gathered write. Typically, an IDE device driver places the **IDEIOGTHW** call in its **open** routine for device instances that it will send gathered writes to.

ataide_buf Structure

The **ataide_buf** structure is used for communication between the IDE device driver and the IDE adapter device driver during an initiator I/O request. This structure is passed to and from the **strategy** routine in the same way a standard driver uses a **struct buf** structure.

Fields in the ataide_buf Structure

The **ataide_buf** structure contains certain fields used to pass an IDE command and associated parameters to the IDE adapter device driver. Other fields within this structure are used to pass returned status back to the IDE device driver. The **ataide_buf** structure is defined in the **/usr/include/sys/ide.h** file.

Fields in the **ataide_buf** structure are used as follows:

1. Reserved fields should be set to a value of 0, except where noted.
2. The **bufstruct** field contains a copy of the standard **buf** buffer structure that documents the I/O request. Included in this structure, for example, are the buffer address, byte count, and transfer direction. The **b_work** field in the **buf** structure is reserved for use by the IDE adapter device driver. The current definition of the **buf** structure is in the **/usr/include/sys/buf.h** include file.
3. The **bp** field points to the original buffer structure received by the IDE device driver from the caller, if any. This can be a chain of entries in the case of spanned transfers (IDE commands that transfer data from or to more than one system-memory buffer). A null pointer indicates a nonspanned transfer. The null value specifically tells the IDE adapter device driver all the information needed to perform the DMA data transfer is contained in the **bufstruct** fields of the **ataide_buf** structure. If the **bp** field is set to a non-null value, the **ataide_buf.sg_ptr** field must have a value of null, or else the operation is not allowed.
4. The **ata** field, defined as an **ata_cmd** structure, contains the IDE command (ATA or ATAPI), status, error indicator, and a flag variable:
 - The **flags** field contains the following bit flags:

ATA_CHS_MODE	Execute the command in cylinder head sector mode.
ATA_LBA_MODE	Execute the command in logical block addressing mode.
ATA_BUS_RESET	Reset the ATA bus, ignore the current command.

- The `command` field is the IDE ATA command opcode. For ATAPI packet commands, this field must be set to `ATA_ATAPI_PACKET_COMMAND` (0xA1).
 - The `device` field is the IDE indicator for either the master (0) or slave (1) IDE device.
 - The `sector_cnt_cmd` field is the number of sectors affected by the command. A value of zero usually indicates 256 sectors.
 - The `startblk` field is the starting LBA or CHS sector.
 - The `feature` field is the ATA feature register.
 - The `status` field is an output parameter indicating the ending status for the command. This field is updated by the IDE adapter device driver upon completion of a command.
 - The `errval` field is the error type indicator when the `ATA_ERROR` bit is set in the `status` field. This field has slightly different interpretations for ATA and ATAPI commands.
 - The `sector_cnt_ret` field is the number of sectors not processed by the device.
 - The `endblk` field is the completion LBA or CHS sector.
 - The `atapi` field is defined as an `atapi_command` structure, which contains the IDE ATAPI command. The 12 or 16 bytes of a single IDE command are stored in consecutive bytes, with the opcode identified individually. The `atapi_command` structure contains the following fields:
 - The `length` field is the number of bytes in the actual IDE command. This is normally 12 or 16 (decimal).
 - The `packet.op_code` field specifies the standard IDE ATAPI opcode for this command.
 - The `packet.bytes` field contains the remaining command-unique bytes of the IDE ATAPI command block. The actual number of bytes depends on the value in the `length` field.
5. The `sg_ptr` field is set to a non-null value to indicate a request for a gathered write. A gathered write means the IDE command conducts a system-to-device data transfer where multiple, noncontiguous system buffers contain the write data. This data is transferred in order as a single data transfer for the IDE command in this `ataide_buf` structure.
- The contents of the `sg_ptr` field, if non-null, must be a pointer to the `uio` structure that is passed to the IDE device driver. The IDE adapter device driver treats the `sg_ptr` field as a pointer to a `uio` structure that accesses the `iovec` structures containing pointers to the data. There are no address-alignment restrictions on the data in the `iovec` structures. The only restriction is that the total transfer length of all the data must not exceed the maximum transfer length for the adapter device driver.
- The `ataide_buf.bufstruct.b_un.b_addr` field normally contains the starting system-buffer address and is ignored and can be altered by the IDE adapter device driver when the `ataide_buf` is returned. The `ataide_buf.bufstruct.b_bcount` field should be set by the caller to the total transfer length for the data.
- 6. The `timeout_value` field specifies the time-out limit (in seconds) to be used for completion of this command. A time-out value of 0 means no time-out is applied to this I/O request.
 - 7. The `status_validity` field contains an output parameter that can have the following bit flags as a value:

<code>ATA_IDE_STATUS</code>	The <code>ata.status</code> field is valid.
<code>ATA_ERROR_VALID</code>	The <code>ata.errval</code> field contains a valid error indicator.
<code>ATA_CMD_TIMEOUT</code>	The IDE adapter driver caused the command to time out.
<code>ATA_NO_DEVICE_RESPONSE</code>	The IDE device is not ready.
<code>ATA_IDE_DMA_ERROR</code>	The IDE adapter driver encountered a DMA error.
<code>ATA_IDE_DMA_NORES</code>	The IDE adapter driver was not able to transfer entire request. The <code>bufstruct.b_resid</code> contains the count not transferred.

If an error is detected during execution of an IDE command, and the error prevented the IDE command from actually being sent to the IDE bus by the adapter, then the error should be processed or recovered, or both, by the IDE adapter device driver.

If it is recovered successfully by the IDE adapter device driver, the error is logged, as appropriate, but is not reflected in the `ata.errval` byte. If the error cannot be recovered by the IDE adapter device driver, the appropriate `ata.errval` bit is set and the `ataide_buf` structure is returned to the IDE device driver for further processing.

If an error is detected after the command was actually sent to the IDE device, then it should be processed or recovered, or both, by the IDE device driver.

For error logging, the IDE adapter device driver logs IDE bus- and adapter-related conditions, while the IDE device driver logs IDE device-related errors. In the following description, a capital letter "A" after the error name indicates that the IDE adapter device driver handles error logging. A capital letter "H" indicates that the IDE device driver handles error logging.

Some of the following error conditions indicate an IDE device failure. Others are IDE bus- or adapter-related.

<code>ATA_IDE_DMA_ERROR (A)</code>	The system I/O bus generated or detected an error during a DMA transfer.
<code>ATA_ERROR_VALID (H)</code>	The request sent to the device failed.
<code>ATA_CMD_TIMEOUT (H)</code>	The command timed out before completion.
<code>ATA_NO_DEVICE_RESPONSE (A)</code>	The target device did not respond.
<code>ATA_IDE_BUS_RESET (A)</code>	The adapter indicated the IDE bus reset failed.

Other IDE Design Considerations

IDE Device Driver Tasks

IDE device drivers are responsible for the following actions:

- Interfacing with block I/O and logical volume device driver code in the operating system.
- Translating I/O requests from the operating system into IDE commands suitable for the particular IDE device. These commands are then given to the IDE adapter device driver for execution.
- Issuing any and all IDE commands to the attached device. The IDE adapter device driver sends no IDE commands except those it is directed to send by the calling IDE device driver.

Closing the IDE Device

When an IDE device driver is preparing to close a device through the IDE adapter device driver, it must ensure that all transactions are complete. When the IDE adapter device driver receives an **IDEIOSTOP** ioctl operation and there are pending I/O requests, the ioctl operation does not return until all have completed. New requests received during this time are rejected from the adapter device driver's **ddstrategy** routine.

IDE Error Processing

It is the responsibility of the IDE device driver to process IDE check conditions and other returned errors properly. The IDE adapter device driver only passes IDE commands without otherwise processing them and is not responsible for device error recovery.

Device Driver and Adapter Device Driver Interfaces

The IDE device drivers can have both character (raw) and block special files in the **/dev** directory. The IDE adapter device driver has only character (raw) special files in the **/dev** directory and has only the **ddconfig**, **ddopen**, **ddclose**, **dddump**, and **ddioctl** entry points available to operating system programs. The **ddread** and **ddwrite** entry points are not implemented.

Internally, the **devsw** table has entry points for the **ddconfig**, **ddopen**, **ddclose**, **dddump**, **ddioctl**, and **ddstrategy** routines. The IDE device drivers pass their IDE commands to the IDE adapter device driver by calling the IDE adapter device driver **ddstrategy** routine. (This routine is unavailable to other operating system programs due to the lack of a block-device special file.)

Access to the IDE adapter device driver's **ddconfig**, **ddopen**, **ddclose**, **dddump**, **ddioctl**, and **ddstrategy** entry points by the IDE device drivers is performed through the kernel services provided. These include such services as **fp_opendev**, **fp_close**, **fp_ioctl**, **devdump**, and **devstrat**.

Performing IDE Dumps

An IDE adapter device driver must have a **dddump** entry point if it is used to access a system dump device. An IDE device driver must have a **dddump** entry point if it drives a dump device. Examples of dump devices are disks and tapes.

Note: IDE adapter device driver writers should be aware that system services providing interrupt and timer services are unavailable for use in the **dump** routine. Kernel DMA services are assumed to be available for use by the **dump** routine. The IDE adapter device driver should be designed to ignore extra **DUMPINIT** and **DUMPSTART** commands to the **dddump** entry point.

The **DUMPQUERY** option should return a minimum transfer size of 0 bytes, and a maximum transfer size equal to the maximum transfer size supported by the IDE adapter device driver.

Calls to the IDE adapter device driver **DUMPWRITE** option should use the **arg** parameter as a pointer to the **ataide_buf** structure to be processed. Using this interface, an IDE write command can be executed on a previously started (opened) target device. The **uiop** parameter is ignored by the IDE adapter device driver

during the **DUMPWRITE** command. Spanned or consolidated commands are not supported using the **DUMPWRITE** option. Gathered write commands are also not supported using the **DUMPWRITE** option. No queuing of **ataide_buf** structures is supported during dump processing since the **dump** routine runs essentially as a subroutine call from the caller's dump routine. Control is returned when the entire **ataide_buf** structure has been processed.

Note: No error recovery techniques are used during the **DUMPWRITE** option because *any* error occurring during **DUMPWRITE** is a true problem. Return values from the call to the **dddump** routine indicate the specific nature of the failure.

Successful completion of the selected operation is indicated by a 0 return value to the subroutine. Unsuccessful completion is indicated by a return code set to one of the following values for the **errno** global variable. The various **ataide_buf** status fields, including the **b_error** field, are not set by the IDE adapter device driver at completion of the **DUMPWRITE** command. Error logging is, of necessity, not supported during the dump.

- An **errno** value of **EINVAL** indicates that a request that was not valid passed to the IDE adapter device driver, such as to attempt a **DUMPSTART** command before successfully executing a **DUMPINIT** command.
- An **errno** value of **EIO** indicates that the IDE adapter device driver was unable to complete the command due to a lack of required resources or an I/O error.
- An **errno** value of **ETIMEDOUT** indicates that the adapter did not respond with completion status before the passed command time-out value expired.

Required IDE Adapter Device Driver ioctl Commands

Various ioctl operations must be performed for proper operation of the IDE adapter device driver. The ioctl operations described here are the minimum set of commands the IDE adapter device driver must implement to support IDE device drivers. Other operations may be required in the IDE adapter device driver to support, for example, system management facilities. IDE device driver writers also need to understand these ioctl operations.

Every IDE adapter device driver must support the **IOCINFO** ioctl operation. The structure to be returned to the caller is the **devinfo** structure, including the **ide** union definition for the IDE adapter found in the **/usr/include/sys/devinfo.h** file. The IDE device driver should request the **IOCINFO** ioctl operation (probably during its open routine) to get the maximum transfer size of the adapter.

Note: The IDE adapter device driver ioctl operations can only be called from the process level. They cannot be executed from a call on any more favored priority levels. Attempting to call them from a more favored priority level can result in a system crash.

ioctl Commands

The following **IDEIOSTART** and **IDEIOSTOP** operations must be sent by the IDE device driver (for the open and close routines, respectively) for each device. They cause the IDE adapter device driver to allocate and initialize internal resources. The **IDEIORESET** operation is provided for clearing device hard errors. The **IDEIOGTHW** operation is supported by IDE adapter device drivers that support gathered write commands to target devices.

Except where noted otherwise, the **arg** parameter for each of the **ioctl** operations described here must contain a long integer. In this field, the least significant byte is the IDE device ID value. (The upper three bytes are reserved and should be set to 0.) This provides the information required to allocate or deallocate resources and perform IDE bus operations for the **ioctl** operation requested.

The following information is provided on the various **ioctl** operations:

IDEIOSTART This operation allocates and initializes IDE device-dependent information local to the IDE adapter device driver. Run this operation only on the first open of a device. Subsequent **IDEIOSTART** commands to the same device fail unless an intervening **IDEIOSTOP** command is issued.

The following values for the **errno** global variable are supported:

0 Indicates successful completion.

EIO Indicates lack of resources or other error-preventing device allocation.

EINVAL Indicates that the selected IDE device ID is already in use.

ETIMEDOUT Indicates that the command did not complete.

IDEIOSTOP This operation deallocates resources local to the IDE adapter device driver for this IDE device. This should be run on the last close of an IDE device. If an **IDEIOSTART** operation has not been previously issued, this command is unsuccessful.

The following values for the **errno** global variable should be supported:

0 Indicates successful completion.

EIO Indicates error preventing device deallocation.

EINVAL Indicates that the selected IDE device ID has not been started.

ETIMEDOUT Indicates that the command did not complete.

IDEIORESET This operation causes the IDE adapter device driver to send an ATAPI device reset to the specified IDE device ID.

The IDE device driver should use this command only when directed to do a forced open. This occurs in for the situation when the device needs to be reset to clear an error condition.

Note: In normal system operation, this command should not be issued, as it would reset all devices connected to the controller. If an **IDEIOSTART** operation has not been previously issued, this command is unsuccessful.

The following values for the errno global variable are supported:

0 Indicates successful completion.

EIO Indicates an unrecovered I/O error occurred.

EINVAL

Indicates that the selected IDE device ID has not been started.

ETIMEDOUT

Indicates that the command did not complete.

IDEIOGTHW This operation is only supported by IDE adapter device drivers that support gathered write commands. The purpose of the operation is to indicate support for gathered writes to IDE device drivers that intend to use this facility. If the IDE adapter device driver does not support gathered write commands, it must fail the operation. The IDE device driver should call this operation from its open routine for a particular device instance. If the operation is unsuccessful, the IDE device driver should not attempt to run a gathered write command.

The **arg** parameter to the **IDEIOGTHW** is set to NULL by the caller to indicate that no input parameter is passed:

The following values for the **errno** global variable are supported.

0 Indicates successful completion and in particular that the adapter driver supports gathered writes.

EINVAL

Indicates that the IDE adapter device driver does not support gathered writes.

Chapter 16. Serial Direct Access Storage Device Subsystem

With *sequential* access to a storage device, such as with tape, a system enters and retrieves data based on the location of the data, and on a reference to information previously accessed. The closer the physical location of information on the storage device, the quicker the information can be processed.

In contrast, with *direct* access, entering and retrieving information depends only on the location of the data and not on a reference to data previously accessed. Because of this, access time for information on direct access storage devices (DASDs) is effectively independent of the location of the data.

Direct access storage devices (DASDs) include both fixed and removable storage devices. Typically, these devices are hard disks. A *fixed* storage device is any storage device defined during system configuration to be an integral part of the system DASD. If a fixed storage device is not available at some time during normal operation, the operating system detects an error.

A *removable* storage device is any storage device you define during system configuration to be an optional part of the system DASD. Removable storage devices can be removed from the system at any time during normal operation. As long as the device is logically unmounted before you remove it, the operating system does not detect an error.

The following types of devices are not considered DASD and are not supported by the logical volume manager (LVM):

- Diskettes
- CD-ROM (compact disk read-only memory)
- WORM (write-once read-mostly)

DASD Device Block Level Description

The DASD *device block* (or *sector*) level is the level at which a processing unit can request low-level operations on a device block address basis. Typical low-level operations for DASD are read-sector, write-sector, read-track, write-track, and format-track.

By using direct access storage, you can quickly retrieve information from random addresses as a stream of one or more blocks. Many DASDs perform best when the blocks to be retrieved are close in physical address to each other.

A DASD consists of a set of flat, circular rotating platters. Each platter has one or two sides on which data is stored. Platters are read by a set of nonrotating, but positionable, read or read/write heads that move together as a unit.

The following terms are used when discussing DASD device block operations:

sector An addressable subdivision of a track used to record one block of a program or data. On a DASD, this is a contiguous, fixed-size block. Every sector of every DASD is exactly 512 bytes.

track	<p>A circular path on the surface of a disk on which information is recorded and from which recorded information is read; a contiguous set of sectors. A track corresponds to the surface area of a single platter swept out by a single head while the head remains stationary.</p> <p>A DASD contains at least 17 sectors per track. Otherwise, the number of sectors per track is not defined architecturally and is device-dependent. A typical DASD track can contain 17, 35, or 75 sectors.</p> <p>A DASD may contain 1024 tracks. The number of tracks per DASD is not defined architecturally and is device-dependent.</p>
head	<p>A head is a positionable entity that can read and write data from a given track located on one side of a platter. Usually a DASD has a small set of heads that move from track to track as a unit.</p> <p>There must be at least 43 heads on a DASD. Otherwise, the number is not defined architecturally and is device-dependent. A typical DASD has 8 heads.</p>
cylinder	<p>The tracks of a DASD that can be accessed without repositioning the heads. If a DASD has n number of vertically aligned heads, a cylinder has n number of vertically aligned tracks.</p>

Chapter 17. Debugging Tools

This chapter provides information about the available procedures for debugging a device driver which is under development. The procedures discussed include:

- Saving device driver information in a system dump.
- Using the **crash** command to interpret and format system structures.
- Using the LLDB Kernel Debugger to set breakpoints and display variables and registers.
- Using the KDB Kernel Debugger and Command to set breakpoints and display variables and registers.
- Error logging to record device-specific hardware or software abnormalities.
- Using the Debug and Performance Tracing to monitor entry and exit of device drivers and selectable system events.
- Using the Memory Overlay Detection System (MODS) to help detect memory overlay problems in the AIX kernel, kernel extensions, and device drivers.

System Dump

The system dump copies selected kernel structures to the dump when an unexpected system halt occurs, when the reset button is pressed, or when the special system dump key sequences are entered. You can also initiate a system dump through the System Management Interface Tool (SMIT). For more information, see "Start a System Dump" in *AIX Version 4.3 Problem Solving Guide and Reference*.

The dump device can be dynamically configured, which means that either the tape or logical volumes on hard disk can be used to receive the system dump. Use the **sysdumpdev** command to dynamically configure the dump device.

You can also define primary and secondary dump devices. A primary dump device is a dedicated dump device, while a secondary dump device is shared.

The system kernel **dump** routine contains all the vital structures of the running system, such as the process table, the kernel's global memory segment, and the data and stack segment of each process.

Be sure to refer to the system header files in the **/usr/include/sys** directory. The name of the file tells which structure and associated information it contains. For example, the user block is defined in **sys/user.h**. The process block is defined in **sys/proc.h**.

When you examine system data that maps into these structures, you can gain valuable kernel information that can explain why the dump was called.

Initiating a System Dump

A system dump initiated by a kernel panic is written to the primary dump device. If you initiate a system dump by pressing the reset button, the system dump is written to the primary dump device.

Use the special key sequences to determine whether the write of a system dump goes to the primary dump device or to the secondary dump device. To write to the primary dump device, use the sequence Ctrl-Alt-NumPad1. To write to the secondary dump device, use the sequence Ctrl-Alt-NumPad2.

To use SMIT, select **Problem Determination** from the main menu, then select **System Dump**. This presents a menu that allows you to initiate a system dump to either the primary or secondary device, and manipulate the dump devices and the system dump files.

If you prefer to initiate the system dump from the command line, use the **sysdumpstart** command. Use the **-p** flag to write to the primary device or the **-s** flag to write to the secondary device.

If you want your device to be a primary or secondary device, the driver must contain a **dddump** routine.

When the system dump completes, the system either halts or reboots, depending upon the setting of the **autorestart** attribute of **sys0**. This can be shown and altered using SMIT by selecting **System Environments**, then **Change /Show Characteristics of Operating System**. The **Automatically REBOOT system after a crash** item shows and sets this value.

Including Device Driver Information in a System Dump

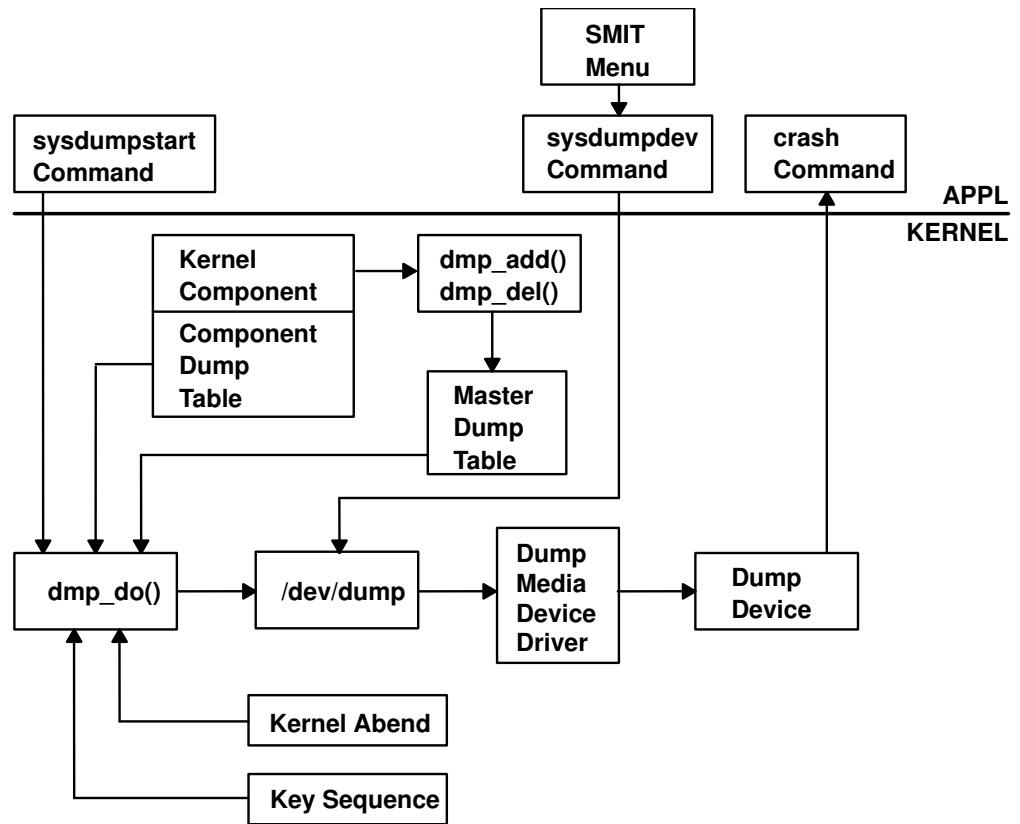
The system dump is table driven. The two parts of the table are:

master dump table	Contains a pointer to a function which is provided by the device driver. The function is called by the kernel dump routine when a system dump occurs. The function must return a pointer to a component dump table.
component dump table	Specifies memory areas to be included in a system dump.

Both the master dump table and the component dump table must reside in pinned global memory.

When a dump occurs, the kernel dump routine calls the function pointed to in the master dump table twice. On the first call, an argument of 1 indicates that the kernel dump routine is starting to dump the data specified by the component dump table.

On the second call, an argument of 2 indicates that the kernel dump routine has finished dumping the data specified by the component dump table. The component dump table should be allocated and pinned during initialization. The entries in the component dump table can be filled in later. The function pointed to in the master dump table must not attempt to allocate memory when it is called. The "System Dump Flow" figure shows the flow of a system dump.



System Dump Flow

To have your device driver data areas included in a system dump, you must register the data areas in the master dump table. Use the **dmp_add** kernel service to add an entry to the master dump table. Conversely, use the **dmp_del** kernel service to delete an entry from the master dump table. The syntax is as follows:

```
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/dump.h>
int dmp_add(cdt_func) or int dmp_del(cdt_func)
int cdt * ((*cdt_func) ());
```

The **cdt** structure is defined in the **sys/dump.h** header file. A **cdt** structure consists of a fixed-length header (**cdt_head** structure) and an array of one or more **cdt_entry** structures.

The **cdt_head** structure contains a component name field, containing the name of the device driver, and the length of the component dump table. Each **cdt_entry** structure describes a contiguous data area, giving a pointer to the data area, its length, a segment register, and a name for the data area. Use the name supplied for the data area to refer to it when the **crash** command formats the dump. The "Kernel Dump Image" figure illustrates a dump image.

Component Dump Table – A
Bitmap for 1st data area
1st data area for component A
Bitmap for 2nd data area
2nd data area for component A
...
Component Dump Table – N
Bitmap for 1st data area
1st data area for component N
Bitmap for 2nd data area
2nd data area for component N

Kernel Dump Image

Formatting a System Dump

Each device driver that includes data in a system dump can install a unique formatting routine in the `/usr/lib/ras/dmprtms` directory. A formatting routine is a command that is called by the `crash` command. The name of the formatting routine must match the component name field of the corresponding component dump table.

The `crash` command forks a child process that runs the formatting routines. If a formatting routine is not provided for a component name, the `crash` command runs the `_default_dmp_fmt` default-formatting routine, which prints out the data areas in hex.

The `crash` command calls the formatting routine as a command, passing the file descriptor of the open dump image file as a command line argument. The syntax for this argument is `-ffile_descriptor`.

The dump image file includes a copy of each component dump table used to dump memory. Before calling a formatting routine, the `crash` command positions the file pointer for the dump image file to the beginning of the relevant component dump table copy.

The dumped memory is laid out in the dump image file with the component dump table and is followed by a bitmap for the first data area, then the first data area itself. A bitmap for the next data area follows, then the next data area itself, and so on.

The bitmap for a given data area indicates which pages of the data area are actually present in the dump image and which are not. Pages that were not in memory when the dump occurred were not dumped. The least significant bit of the first byte of the bitmap is set to 1 if the first page is present. The next least significant bit indicates the presence or absence of the second page, and so on. A macro for determining the size of a bitmap is provided in `sys/dump.h`.

Note: A sample dump formatter is shipped with `bos.sysmgt.serve_aid` in the `/usr/samples/dumpfmt` directory.

The crash Command

The `crash` command is a particularly useful tool for device-driver development and debugging, which interprets and formats the system structures. The `crash` command is interactive and allows you to examine an operating system image or an active system. An operating system image is held in a system dump file, either as a file or on the dump device. When you run the `crash` command, you can optionally specify a system image file and kernel file, as shown in the syntax below:

```
crash [-a] [-i IncludeFile] [ SystemImageFile [ KernelFile ] ]
```

The default `SystemImageFile` is `/dev/mem` and the default `KernelFile` is `/usr/lib/boot/unix`.

To run the `crash` command on the active system, enter:

```
crash
```

Because the command uses `/dev/mem`, you need root permissions.

To invoke the `crash` command on a system image file, enter:

```
crash SystemImageFile
```

where `SystemImageFile` is either a file name or the name of the dump device.

Note that by convention, the symbol names for function entry points always begin with a `.` (period). In most cases, there is a corresponding symbol name without the period that points to the function descriptor. However, when you specify a function symbol name on a crash command, without a leading period, crash inserts the period for you. For data items, there usually are table-of-contents (TOC) entries corresponding to each data item, but there are no differences in the names. The crash command assumes that when a data item symbol is specified, it is the actual data item that is wanted, not the TOC entry.

Use the `-a` flag to generate a list of data structures without using subcommands. The resulting list is large, so you can redirect the output to either a file or to a printer.

Use the `-i` flag to read the given include file, allowing the `print` subcommand to output data according to the include file structures.

You can use a variety of subcommands to view the system structures. These subcommands can have flags that modify the format of the data. If you do not use a flag to specify what you want to see, all valid entries are displayed.

Addresses in crash

Many of the commands in crash take addresses as parameters. Addresses are always specified in hexadecimal, and can usually be specified in one of the following forms:

addr	An 8 digit hexadecimal number, which is treated as an effective address within the context of the current process and thread, or (in some cases) the context of the thread specified on a previous cm command. addr can be prefixed with the characters 0x.
segid:offset	segid is the segment ID for a virtual memory segment. The maximum size is 6 hex digits. offset is the offset (in bytes) from the beginning of that segment. The maximum size is 7 hex digits.
r:realaddr	r is the literal character "r". realaddr is a real memory address. This form can only be used when running crash against a system dump, and it only will display dump data areas that were dumped by real address instead of virtual address. readaddr can be up to 12 hexadecimal digits.

To enhance readability, you may include underscores ("_") anywhere within these values.

Examples:

```
18340050
2314:55300
r:14_3370_0560 (same as r:1433700560)
```

Command-line Editing

The **crash** command provides command line editing features similar to those provided by the Korn shell. **vi** mode provides vi-like editing features, while **emacs** mode gives you controls similar to emacs. You can turn these features on by using the crash subcommand **set edit**. So, to turn on vi-style command-line editing, you would type the subcommand `set edit vi`.

Output Redirection

The crash command provides a subset of Korn shell input/output redirection. Specifically, the following operators are provided:

| (pipe symbol)

Pipes all output of the command before the symbol to the input of the command after the symbol. Both standard output and error output are affected, which is different than standard shell behavior.

> filename

Writes the output of the command before the > to filename. Both standard and error output are written to the file.

>> filename

Adds the output of the command before the >> to the end of filename. Both standard and error output are written to the file.

crash Subcommands

Once you initiate the **crash** command, `>` is the prompt character. For a list of the available subcommands, type the `?` character. To exit, type `q`. You can run any shell command from within the **crash** command by preceding it with an `!` (exclamation mark).

Since the **crash** command only deals with kernel threads, the word "thread" when used alone will be used to mean kernel thread in the **crash** documentation that follows. The default thread for several subcommands is the current thread (the thread currently running). On a multiprocessor system, you can use the **cpu** subcommand to change the current processor; the default thread becomes the running thread on the selected processor.

The parameters *ProcessSlotNumber* and *ThreadSlotNumber* are used in many subcommands to indicate a process or thread respectively. These parameters are simply numbers for table entry indexes which can be displayed using the **proc** and **thread** subcommands.

Note that many structures displayed are longer than one screen length.

buf [BufferHeaderNumber]

The **buf** subcommand displays the system buffer headers. A buffer header contains the information required to perform block I/O. If you type the **buf** subcommand with no *BufferHeaderNumber*, a summary of the system buffer headers is displayed.

Aliases = **bufhdr**, **hdr**

```
> buf
BUF MAJ  MIN    BLOCK  FLAGS
  0 000a 000b      8  done stale
  1 000a 000b    243  done stale
  2 000a 000b     24  done stale
...
```

If you type the **buf** subcommand with a *BufferHeaderNumber*, a single complete header is displayed:

```
> buf 3
BUFFER HEADER 3:
  b_forw: 0x014d0528, b_back: 0x014d0160, b_vp: 0x00000000
  av_forw: 0x014d0160, av_back: 0x014d0528, b_iodone: 0x000185f8
  b_dev: 0x000a000b, b_blkno: 0, b_addr: 0x014e9000
  b_bcount: 4096, b_error: 0, b_resid: 0
  b_work: 0x80000000, b_options:0x00000000, b_event: 0xffffffff
  b_start.tv_sec: 0, b_start.tv_nsec: 0
  b_xmemd.aspace_id: 0x00000000, b_xmemd.subspace_id: 0x00000000
  b_flags: read done stale
```

Refer to the **sys/buf.h** header file for the structure definition.

buffer [Format] [BufferHeaderNumber]

The **buffer** subcommand displays the data in a system buffer according to the *Format* parameter. When specifying a buffer header number, the buffer associated with that buffer header is displayed. If you do not provide a *Format* parameter, the previous *Format* is used. Valid options are **decimal**, **octal**, **hex**, **character**, **byte**, **i-node**, **directory**, and **write**. The **write** option creates a file in the current directory containing the buffer data.

Aliases = b

```
> buffer hex 3
BUFFER FOR BUF_HDR 3
00000: 41495820 4c564342 00006a66 73000000
00020: 00000000 00000000 00000000 00000000
00040: 00000000 00000000 00003030 30303033
...
```

callout

The **callout** subcommand displays all active entries on the active **trblist**. When the **time-out** kernel extension is used in a device driver, this timer request is entered on a system-wide list of active timer requests. This list of timer requests is the **trblist**. Any timer which is active is on this list until it expires.

Aliases = c, call, calls, time, timeout, tout

```
>callout
TRB's On The Active List Of Processor 0.
TRB #1 on Active List
Timer address.....0x0
trb.to_next.....0x0
trb.knext.....0x59aa100
trb.kprev.....0x0
Thread id (-1 for dev drv).....0xffffffffe
Timer flags.....0x12
trb.timerid.....0x0
trb.eventlist.....0xfffffffff
trb.timeout.it_interval.tv_nsec....0x0
trb.timeout.it_interval.tv_sec....0x0
Next scheduled timeout (secs).....0x2d63f6a8
Next scheduled timeout (nanosecs)..0xc849a80
Timeout function.....0x8c748
Timeout function data.....0x59aa040
TRB #2 on Active List
...
```

Refer to **sys/timer.h** for the structure definitions, and to InfoExplorer for a description of the time-out mechanism.

cm [*ldron|ldroff*] [*vmmon|vmmoff*] [*ThreadSlotNumber SegmentNumber*]

The **cm** subcommand changes the current segment map used by the **od** subcommand. The **cm** subcommand changes the map of the **crash** command internal pointers for any process thread segment not paged out, if you specify *ThreadSlotNumber* and *SegmentNumber*. This allows the **od** subcommand to display data from the segment desired rather than the segment for the current thread. Specification of *vmmon* or *vmmoff* allows selection of whether effective addresses in the range 0x70000000 through 0xffffffff are to be interpreted by the **od** subcommand as kernel or VMM data references. Similarly, the **ldron** and **ldroff** options allow selection of whether effective addresses in segment 11 (0xbxxxxxx) and segment 13 (0xdxxxxxx) are to be interpreted by the **od** subcommand as references to loader data. Using the **cm** subcommand without any parameters resets the map of internal pointers.

The following example sets the map to *ThreadSlotNumber* 3 and *SegmentNumber* 2, then displays 20 words from segment 2 for the thread in slot number 3. It then resets to the normal mapping by executing the **cm** subcommand with no parameters.

Aliases = none

```
> cm 3 2
t3,2 >> od 2ff3b400 20
2ff3b400: 00000000 00000000 2ff22e28 00000000
2ff3b410: 00000306 00000000 0002a7ec 000010b0
2ff3b420: 82202220 0002a7ec 00000000 0000001c
2ff3b430: 00000000 00000000 00000000 00000000
2ff3b440: d80a5000 40000000 00002c0b d80a5000
t3,2 >> cm
```

The following example shows how the **crash** prompt changes as the various **cm** options are used. First, the **cm** subcommand is issued to indicate that effective address in segment 11 and 13 are to be considered loader references. Second, a **cm** subcommand is used to indicate that effective address in the range 0x70000000 through 0xffffffff are to be considered VMM references. Then, the **cm** subcommand is used to indicate that reference to effective addresses for 0x20000000 through 0x2fffffff are to use the segment id from segment register 2 of the thread in thread slot 3 (see previous example). Then these options are individually cleared.

```
> cm ldron
LDR > cm vmmon
VMM LDR > cm 3 2
t3,2 VMM LDR >> cm ldroff
t3,2 VMM >> cm vmloff
t3,2 >> cm
>
```

cpu [ProcessorNumber]

If no argument is given, the **cpu** subcommand displays the number of the currently selected processor. Initially, the selected processor is processor 0 (on a running system) or the processor on which the crash occurred (when running **crash** against a dump). If the *ProcessorNumber* argument is given, the **cpu** subcommand selects the specified processor as the current processor. By extension, this selects the current kernel thread (the running kernel thread on the selected processor). Processor numbering starts from zero.

Aliases = none

```
>cpu
Selected cpu number : 0
```

dblock [Address]

The **dblock** subcommand displays the allocated streams data-block headers. The address parameter is required. If the address is not supplied, this command will print an error message stating that the address is required. Refer to the **sys/stream.h** file for the **datab** structure definitions. The **freep** and **db_size** definitions are not included in **/usr/include/sys/stream.h**. These structure members are described here:

```
freep      Address of the free pointer
db_size    Size of the data block
```

There is no checking performed on the address passed in as the required parameter. The **dblock** subcommand will accept any address. It is up to the user to be sure that a valid address is specified.

To determine a valid address, run the **mblock** subcommand. From the output of the **mblock** subcommand, select a nonzero data block address under the **DATABLOCK** column heading.

This subcommand can be issued from **crash** on either a running system or a system dump.

Aliases = dblk

```
> queue 59d5a74
  QUEUE   QINFO     NEXT  PRIVATE  FLAGS     HEAD   OTHERQ     COUNT
59d5a74 1884c1c 59d5474 59d5500 0x003e 59e1c00 59d5a00     4096
> mblk 59e1c00
  ADDRESS  NEXT  PREVIOUS  CONT  RPTR  WPTR  DATABLOCK
59e1c00   0     0         0 59e2000 59e3000 59e1c44
> dblk 59e1c44
  ADDRESS  FREEP  BASE  LIM  REFCNT  TYPE  SIZE
59e1c44   0 59e2000 59e3000 1      0     1000
```

dlock [ThreadIdentifier | -p [ProcessorNumber]]

Displays deadlock analysis information about all types of locks (simple, complex, and lockl). The **dlock** subcommand searches for deadlocks from a given start point. If *ThreadIdentifier* is given, the corresponding kernel thread is the start point. If **-p** is given without a *ProcessorNumber*, the start point is the running kernel thread on the current processor. If **-p ProcessorNumber** is given, the running kernel thread on the specified processor is the start point. If no arguments are given, **dlock** searches for deadlocks among all threads on all processors.

The first output line gives information about the starting kernel thread, including the lock which is blocking the kernel thread, and a stack trace showing the function calls which led to the blocking lock request. Each subsequent line shows the lock held by the blocked kernel thread from the previous line, and identifies the kernel thread or interrupt handler which is blocked by those locks. If the information required for a full analysis is not available (paged out), an abbreviated display is shown; in this case, examine the stack trace to locate the locking operations which are causing the deadlock. The display stops when a lock is encountered for a second time, or no blocking lock is found for the current kernel thread.

Aliases = none

```
>dlock
Deadlock from tid 00d3f. This tid waits for the first line lock,
owned by Owner-Id that waits for the next line lock, and so on...
  LOCK NAME | ADDRESS | OWNER-ID | WAITING FUNCTION
  lockC1 | 0x001f79e0 | Tid 113d | .lock_write_ppc
           |          |         | called from : .times + 0000020c
Dump data incomplete.Only 0 bytes found out of 4.
           |          |         | called from : .file + 0000000b
  lockC2 | 0x001f79e8 | Tid d3f | .lock_write_ppc
           |          |         | called from : .times + 000001c8
Dump data incomplete.Only 0 bytes found out of 4.
           |          |         | called from : .file + 0000000b
```

dmodsw

The **dmodsw** subcommand displays the streams drivers-switch table. The information printed is contained in an internal structure. The following members of this internal structure are described here:

address	Address of dmodsw
d_next	Pointer to the next driver in the list
d_prev	Pointer to the previous driver in the list
d_name	Name of the driver
d_flags	Flags specified at configuration time

d_sqh Pointer to synch queue for driver-level synchronization
d_str Pointer to streamtab associated with the driver
d_sq_level Synchronization level specified at configuration time
d_refcnt Number of open or pushed count
d_major Major number of a driver

The flags structure member, if set, is based on one of the following values:

#define	Value	Description
F_MODSW_OLD_OPEN	0x1	Supports old-style (V.3) open/close parameters
F_MODSW_QSAFETY	0x2	Module requires safe timeout/bufcall callbacks
F_MODSW_MPSAFE	0x4	Non-MP-Safe drivers need funneling

The synchronization level codes are described in the `/usr/include/sys/strconf.h` file.

This subcommand can be issued from **crash** on either a running system or a system dump.

Aliases = none

```

> dmodsw
NAME      ADDRESS  NEXT    PREVIOUS FLAG  SYNCHQ  STREAMTAB S-LVL COUNT MAJOR
sad       5a0cf40  5a0cf00 5a0c9c0 0x0  5a0ad40  188c600   3    0   12
slog     5a0cf00  5a0cec0 5a0cf40 0x0  5a0ad20  188c8a0   3    0   13
en       5a0cec0  5a0ce80 5a0cf00 0x0  5a0ad00  1893ee0   3    0   27
et       5a0ce80  5a0ce40 5a0cec0 0x0  5a0ace0  1893ee0   3    0   28
tr       5a0ce40  5a0ce00 5a0ce80 0x0  5a0acc0  1893ee0   3    0   29
fi       5a0ce00  5a0cdc0 5a0ce40 0x0  5a0aca0  1893ee0   3    0   30
echo     5a0cdc0  5a0cd80 5a0ce00 0x0           0  18951a0   5    0   31
nuls     5a0cd80  5a0cd40 5a0cdc0 0x0           0  1895190   5    0   32
spx      5a0cd40  5a0cd00 5a0cd80 0x0  5a0ac80  1895d70   3    0   33
unixdg   5a0cd00  5a0ccc0 5a0cd40 0x0  5a0ac60  18a14e0   3    0   34
unixst   5a0ccc0  5a0cc80 5a0cd00 0x0  5a0ac40  18a14e0   3    0   35
udp      5a0cc80  5a0cc40 5a0ccc0 0x0  5a0ac20  18a14e0   3    0   36
tcp      5a0cc40  5a0cb40 5a0cc80 0x0  5a0ac00  18a14e0   3    0   37
rs       5a0cb40  5a0cb00 5a0cc40 0x0           0  18b31d0   5    1   15
pts      5a0cb00  5a0ca40 5a0cb40 0x0           0  18fc930   4    7   24
ptc      5a0ca40  5a0ca00 5a0cb00 0x0           0  18fa5c0   4    2   23
ttyp     5a0ca00  5a0c9c0 5a0ca40 0x0           0  18fc950   4    0   26
pty      5a0c9c0  5a0cf40 5a0ca00 0x0           0  18fc940   4    0   25
  
```

ds [Address]

The **ds** subcommand returns the symbols closest to the given address. The **ds** subcommand can take either a text address or a data address.

Aliases = ts

```

> ds 012345
      .ioctl_systrace + 0x000001b5
  
```

When a number such as `0x000001b5` is displayed, it shows the number of bytes by which the given address is offset from the entry point of the routine.

du [SlotNumber ThreadSlotNumber]

Uses the specified process slot number to display a combined hex and ASCII dump of the user block for any process that is not swapped out. The default is the current process. Displays a combined hex and ASCII dump of the specified thread's uthread structure and of the user structure of the process which owns the thread. If the data is not available (paged out), a message is displayed. The default is the current thread.

Aliases = none

```
> du 3
Uthread structure of thread slot 3
 00000000 00000000 00000000 2ff7fec0 00000000 *...../.....*
 00000010 00000303 00000000 00030644 000010b0 *.....D....*
 00000020 22222828 00030644 00006244 00000009 *" (...D..bd....*
.
.
.
```

dump

The **dump** subcommand displays the name of each component for which there is data present. After you select a component name from the list, the **crash** program loads and runs the associated formatting routine contained in the **/usr/lib/ras/dmprtns** directory.

If there is more than one data area for the selected component, the formatting routine displays a list of the data areas and allows you to select one. The **crash** command then displays the selected data area. You can enter the **quit** subcommand to return to the previously displayed list and make another selection or enter **quit** a second time to leave the **dump** subcommand loop.

Aliases = none

errpt [*count*]

Displays messages in the error log. *Count* is the number of messages to print that have already been read by the **errdemon** process. (The default is 3 messages.) **errpt** always prints all messages that have not yet been read by the **errdemon** process.

Aliases = none

file [FileSlotNumber]

The **file** subcommand displays the file table. Unless you request specific file entries, the command displays only those with a nonzero reference.

Aliases = **files**, **f**

```
> f 3
SLOT REF    INODE  FLAGS
   3   1 0x018e53f0  read
```

Refer to **sys/file.h** for the structure definition.

find [-u] [-s] [-p slot] [-c context] [-a alignment] pattern

Recognized by the **x** subcommand alias. Search user-space for a given pattern. The default is to search the GPR save areas in the mstsav areas which are both on the

Current Save Area Chain (CSA) and in each uthread area for every thread.

-u	Search all process private segments, (Stack, Uarea,...)
-s	Search all process private segment from the current stack pointer.
-c context	Number of bytes of context to print on a match.
-a alignment	Byte alignment for <i>pattern</i> . The default is 4.
-p slot	Search only specified process. The default is to search all processes.

Note: Using the **find** command on a running system may cause system crashes.

Rules for *pattern*

pattern is a search pattern of any arbitrary length that contains either a hexadecimal number or a string. To specify a hexadecimal pattern, just type the hex digits. "Don't care" digits can be represented with the character x. To specify a string pattern, enclose the pattern in double quotes. "Don't care" characters can be represented with the sequence \x.

Examples:

```
> find -k 02x4
00110a28: 02140008      |....|
00110af0: 02640004      |.d..|
00110c80: 02e40004      |....|
003f0ed8: 02242ff8      |.$/.|
...
> find -k "b\xt"
00012534: 6269745f      |bit_|
00012618: 6269745f      |bit_|
0001264c: 6269745f      |bit_|
00021cb0: 62797465      |byte|
00021d60: 62797465      |byte|
...
> find -k "i_ena" 0 250000
001ceaa8: 695f656e 61626c65 |i_enable|
```

find -k [-c context][-a alignment] pattern [start[end]]

Recognized by the **x** subcommand alias. Search the kernel segments. The default range is the whole of each kernel segment.

-c context	Number of bytes of context to print on a match.
-a alignment	Byte alignment for <i>pattern</i> . The default is 4.

find -b branch_addr [start_addr[end_addr]]

Recognized by the **x** subcommand alias. Search for a branch to the given address. The default range is the whole of each kernel segment.

find -m [-a addr] [-t type] [-c] [-i] [start[end]]

Recognized by the **x** subcommand alias. Search the things that look like mbufs. The default search range is the network memory heap.

-a	Search for mbufs that point to this cluster address.
-t type	Only search for this type of mbuf.
-c	Only search for clusters.
-i	Ignore length sanity checks.

find -v [-f] wordval [start[end]]

Recognized by the `x` subcommand alias. Search for the first word not matching the given value. The default is to search the kernel segments.

`-f` Force scan to continue when a region not in the dump is scanned.

find -U seg_id

Recognized by the `x` subcommand alias. Search for processes whose segment registers contain the given segment ID.

fmodsw

The `fmodsw` subcommand displays the streams modules-switch table. The information printed is contained in an internal structure. The following members of this internal structure are described here:

<code>address</code>	Address of <code>fmodsw</code>
<code>d_next</code>	Pointer to the next module in the list
<code>d_prev</code>	Pointer to the previous module in the list
<code>d_name</code>	Name of the module
<code>d_flags</code>	Flags specified at configuration time
<code>d_sqh</code>	Pointer to synch queue for module-level synchronization
<code>d_str</code>	Pointer to streamtab associated with the module
<code>d_sq_level</code>	Synchronization level specified at configuration time
<code>d_refcnt</code>	Number of open or pushed count
<code>d_major</code>	-1

The flags structure member, if set, is based one of the following values:

#define	Value	Description
<code>F_MODSW_OLD_OPEN</code>	0x1	Supports old-style (V.3) open/close parameters
<code>F_MODSW_QSAFETY</code>	0x2	Module requires safe timeout/bufcall callbacks
<code>F_MODSW_MPSAFE</code>	0x4	Non-MP-Safe drivers need funneling

The synchronization level codes are described in the `/usr/include/sys/strconf.h` file.

This subcommand can be issued from `crash` on either a running system or a system dump.

Aliases = none

```
> fmodsw
NAME      ADDRESS  NEXT     PREVIOUS  FLAG  SYNCHQ  STREAMTAB  S-LVL  COUNT  MAJOR
bufcall   5a0cf80  5a0cc00  5a0ca80  0x1   5a0ad60  188bf80    3     0    -1
sc        5a0cc00  5a0cbc0  5a0cf80  0x0   5a0abe0  18a29b0    3     0    -1
timod     5a0cbc0  5a0cb80  5a0cc00  0x0   5a0abc0  18a34b0    3     0    -1
tirdwr    5a0cb80  5a0cac0  5a0cbc0  0x0   5a0aba0  18a4010    3     0    -1
ldterm    5a0cac0  5a0ca80  5a0cb80  0x0           0  18ef460    4     8    -1
```

fs [ThreadSlotNumber]

Traces a kernel stack for the thread specified by *ThreadSlotNumber*. Displays the called subroutines with a hex dump of the stack frame for the subroutine that contains the parameters passed to the subroutine. By default, the current thread is traced. This subcommand will not work on the current thread of a running system because it uses stack tracing; however, it does work on a dump image.

Aliases = none

```
> fs
STACK TRACE:
**** .et wait ****
2ff97e78 2FF97ED8 0080D568 00000000 018F4C60 /.^....h.....L'
2ff97e88 2FF97EE8 0080D568 00082BC0 000BA020 /.^....h..+.....
2ff97e98 2FF97ED8 28008044 00082418 2FF98000 /.^(..D..B./...
2ff97ea8 00000000 000B8468 00000000 00000000 .....h.....
2ff97eb8 2FF97F38 0000000B 00000004 00000004 /..8.....
2ff97ec8 00000005 01DFE258 00000000 E3000600 .....X.....
```

hide symbol...

Hide the specified symbol from the **crash** commands that convert addresses to symbols and offsets. The main reason for this ability is to hide symbols that may show up in the middle of a function. This occurs in assembly routines. See the **unhide symbol...** subcommand on “unhide symbol...” on page 329.

hide

Show all hidden symbols. See the **unhide symbol...** subcommand on “unhide symbol...” on page 329.

inode [-] [<Major <Minor <INumber]

The **inode** subcommand displays the i-node table and the i-node data block addresses. You can display a specific i-node by specifying the major and minor device numbers of the device where the i-node resides and the i-node number. The command displays the i-node only if it is currently on the system hash list.

Aliases = **ino**, **i**

```
>inode
ADDRESS MAJ MIN INUMB REF LINK UID GID SIZE MODE SMAJ SMIN FLAGS
0x018e4e50 010 0007 11264 0 1 2 2 30 ---777 - -
0x018f9fd0 010 0009 16384 1 6 201 0 512 d--755 - -
0x018ea940 010 0011 0 1 0 0 0 0 --- 0 - -
...
```

kfp [FramePointer]

If you use the **kfp** subcommand without parameters, it displays the last kernel frame pointer address that was set using **kfp**. If you specify a frame pointer address, it sets the kernel frame pointer to the new address. Use this subcommand in conjunction with the **-r** flag of the **trace** subcommand.

Aliases = **fp**, **rl**

```
> kfp
kfp: 00000000
```

knlist [Symbol]

The **knlist** subcommand displays the addresses of all the specified symbol names. If there is no such symbol, the subcommand displays a no match message. Run this subcommand only on an active system.

The **knlist** subcommand runs a subroutine to the active kernel to obtain the address from the system's knlist. The **nm** subcommand provides the same function but searches the symbol table in the Kernel Image File for the address and therefore can be used on a dump.

Aliases = none

```
> knlist open
    open:0x000bbc98
```

le [-l32|-l64|-p proc_slot|-a] [[ADDRESS|NAME]...]

The **le** subcommand displays load list entries; the default is to display load list entries starting at the kernel load anchor. If an address is specified, without the **-a** option, only load list entries which include the address within the text or data area are displayed. If a name is specified all load list entries which have a name that includes the input string are displayed. If an attempt is made to display a paged-out loader entry, the subcommand displays an error message. The following options control the list entry chain that is searched/displayed:

-l32	Use the 32-bit shared library load list anchor
-l64	Use the 64-bit shared library load list anchor
-p proc_slot	Use the load list anchor contained in the indicated processes user area
-a	Display a single load list entry at a specified address (an address must be specified with this option)

Aliases = none

The following example displays all load list entries starting at the kernel load anchor.

```
> le
LoadList entry at 0x04e77700
Module start:0x00000000_0509c000  Module filesize:0x00000000_00086820
Module *end:0x00000000_05122820
*data:0x00000000_05113c60  data length:0x00000000_0000ebc0
Use-count:0x0003  load_count:0x0001  *file:0x00000000
flags:0x00000272 TEXT KERNELEX DATAINTEXT DATA DATAEXISTS
*exp:0x05123000  *lex:0x00000000  *deferred:0x00000000  *expsize:0x622f6c69
Name: /usr/vice/etc/dkload/afs.ext
ndepend:0x0001  maxdepend:0x0001
*depend[00]:0x04e77680
le_next: 04e77a80
```

... other loader entries would follow ...

The following example displays any entry from the 32-bit shared library load list chain for which the input address is between the Module start and Module *end values.

```
> le -l32 d0384101
LoadList entry at 0xb0a99a80
Module start:0x00000000_d0384100  Module filesize:0x00000000_00000eb7
Module *end:0x00000000_d0384fb7
*data:0x00000000_b0bcea48  data length:0x00000000_00002138
```

```

Use-count:0x0001 load_count:0x0000 *file:0x10000c30
flags:0x000000c0 DATA LIBRARY
*exp:0xb0acb300 *lex:0x00000000 *deferred:0x00000000 *expsize:0x00000000
Name: /usr/lib/libdbm.a shr.o
ndepend:0x0002 maxdepend:0x0002
*depend[00]:0xb004c780
*depend[01]:0xb03afd80
le_next: b0a99a00

```

The following example displays load list entries for the process in process slot 20.

```

> le -p 20
LoadList entry at 0x2ff7f480
Module start:0x00000000_d0bc6000 Module filesize:0x00000000_00000304
Module *end:0x00000000_d0bc6304
*data:0x00000000_201541f0 data length:0x00000000_0000006c
Use-count:0x0002 load_count:0x0001 *file:0x10002370
flags:0x00001240 DATA DATAEXISTS DATAMAPPED
*exp:0x2ff81040 *lex:0x00000000 *deferred:0x00000000 *expsize:0x00000000
Name: /opt/dcelocal/ext/dfsloadobj
ndepend:0x0002 maxdepend:0x0002
*depend[00]:0xf0263d80
*depend[01]:0x04e77a80
le_next: 2ff7f400

```

... other loader entries would follow ...

Any of the above examples could include either addresses or names as additional arguments. These additional arguments would simply limit the entries displayed to those that contain the input addresses or names.

The following example displays a loader entry at a specified address.

```

> le -a 4e77500
LoadList entry at 0x04e77500
Module start:0x00000000_04fb0000 Module filesize:0x00000000_00000e88
Module *end:0x00000000_04fb0e88
*data:0x00000000_00000012 data length:0x00000000_00000000
Use-count:0x0002 load_count:0x0000 *file:0x00000000
flags:0x00000248 SYSCALLS DATA DATAEXISTS
*exp:0x04fb1000 *lex:0x00000000 *deferred:0x00000000 *expsize:0x00010ca4
Name: /unix
ndepend:0x0002 maxdepend:0x0002
*depend[00]:0x04e77080
*depend[01]:0x04e77480
le_next: 04cb3000

```

linkblk

The **linkblk** subcommand displays the streams linkblk table. Refer to the **/usr/include/sys/stream.h** file for the linkblk structure definitions. If there are no linkblk structures found on the system, the **linkblk** subcommand will print a message stating that no structures are found.

This subcommand can be issued from **crash** on either a running system or a system dump.

Aliases = lblk

This example shows a regular link:

```

> linkblk
   QTOP   QBOT   INDEX
5ab8b74 5ae5074 5ab4200

```

This example shows a persistent link:

```
> linkblk
      QTOP      QBOT      INDEX
      0 5ae5174 5a4ef00
```

mblock Address

The **mblock** subcommand displays the allocated streams message-block headers. The address parameter is required. If the address is not supplied, this command will print an error message stating that the address is required. Refer to the `/usr/include/sys/stream.h` file for the queue structure definitions.

The **mblock** subcommand's checking of the address parameter is limited to verifying that the address falls on a 128-byte boundary. It is up to the user to be sure that a valid address is specified.

To determine a valid address, run the **queue** subcommand. From the output of the **queue** subcommand, select a non-zero address for the head of the message queue under the HEAD column heading for either a read queue or a write queue.

This subcommand can be issued from crash on either a running system or a system dump.

Aliases = mblk

```
> queue
WRITEQ  QINFO  NEXT  PRIVATE  FLAGS  HEAD  READQ  COUNT  NAME
1802c08c 22dac00 1802c48c 1802c200 0x002a 0 1802c000 0 sth
1802c48c 2324960 1802ec8c 1807a2ec 0x0028 0 1802c400 0 mi_timod
1802ec8c 2320158 0 1802cc2c 0x8028 0 1802ec00 0 xtiso
1806ac8c 22dac00 1806c88c 1806aa00 0x002a 0 1806ac00 0 sth
1806c88c 2324960 1806ce8c 1807a02c 0x0028 0 1806c800 0 mi_timod
1806ce8c 2320158 0 1806cc2c 0x8028 0 1806ce00 0 xtiso
1806ae8c 22dac00 1806a88c 1806a000 0x002a 0 1806ae00 0 sth
1806a88c 2324960 1806a28c 1807a56c 0x0028 0 1806a800 0 mi_timod
1806a28c 2320158 0 1806a42c 0x8028 0 1806a200 0 xtiso
1802e68c 22dac00 1802ea8c 1802e400 0x002a 0 1802e600 0 sth
1802ea8c 2b2b580 1802e88c 1802e200 0x0028 0 1802ea00 0 ldterm
1802e88c 25a48d0 0 2b19130 0x0020 180abe00 1802e800 684 rs
> mblk 180abe00
ADDRESS  NEXT  PREVIOUS  CONT  RPTR  WPTR  DATABLOCK
180abe00 180ab800 0 0 180abe6c 180abeb8 0
```

mbuf [-c] [-d] [-l] [addr]

The **mbuf** subcommand displays mbuf structures in the system. These structures are memory buffers that are chained together and can be manipulated by the Memory Buffer kernel services. If you specify the **-d** flag, the subcommand also displays the data associated with the **mbuf** structure. The **-l** flag causes the subcommand to display an entire chain of mbuf structures. The **-c** flag tells **mbuf** to use the cluster free list rather than the mbuf pointer. If *addr* is not specified, then **mbuf** defaults to using the system mbuf pointer. Note that valid mbuf pointers must be on a 128-byte boundary.

```
> mbuf -l
mbuf:0x18099900 len: 120 type: header act:00000000 next:18099400
data:1809995e
mbuf:0x18099400 len: 62 type: header act:00000000 next:18099d00
data:1809942e
mbuf:0x18099d00 len: 156 type: header act:00000000 next:18096800
data:1809b858
```



```

mbuf:0x18096800 len: 156 type: header act:00000000 next:18099300
data:1809b858
mbuf:0x18099300 len: 62 type: header act:00000000 next:18099700
data:1809932e
mbuf:0x18099700 len: 4 type: data act:00000000 next:18099500
data:180a103e
...

```

mst [-f] [Address] . . .

Displays the **mstsave** portion of the **uthread** structure at the addresses specified (see the **uthread.h** and **mstsave.h** header files in **/usr/include/sys**). If you do not specify an address, it displays all of the **mstsave** entries on the current save area (CSA) chain except the first. If you specify the **-f** flag the first **mstsave** area on the CSA chain displays.

Aliases = none

ndb

Displays network kernel data structures either for a running system or a system dump. The **ndb** (network debugger) subcommand displays the following options:

?	Provides first-level help information.
help	Provides additional help information.
tcb [<i>Addr</i>]	Shows TCBS. The default is HEAD TCB.
udb [<i>Addr</i>]	Shows UDBs. The default is HEAD UDB.
sockets	Shows sockets from the file table.
mbuf [<i>Addr</i>]	Shows the mbuf at the specified address.
ifnet [<i>Addr</i>]	Shows the ifnet structures at the specified address.
quit	Stops the running option.
xit	Exits the ndb submenu.

Aliases = none

nm [Symbol]

The **nm** subcommand displays the symbol value and type found in *KernelFile*.

Aliases = none

```

> nm open
00095484 000C70 PR SD <.open>
00095484 PR LD .open
000BBC98 00000C SV SD open

```

od [ldr:] [vmm:] [*...] [SymbolName | Address] [Count] [Format]

The **od** subcommand dumps the number of data values specified by *Count* starting at the *SymbolName* value or *Address* according to *Format*. Possible formats are octal, longoct, decimal, longdec, character, hex, instruction, and byte. The default is hex. Note that if you use the *Format* parameter, you must also use *Count*. If the *SymbolName* or *Address* is preceded by an asterisk, then the symbol or address is dereferenced before displaying the data. Additionally, the strings **ldr:** and **vmm:** may be used to indicate that addresses are to be considered loader or VMM addresses, just as if the **cm ldron** and/or **cm vmmon** subcommands had been issued.

The **od** subcommand is especially useful during program development in order to see structure values at a given point in time.

Aliases = none

```
> od open 10
00095484: 7c0802a6 bf21ffe4 90010008 9421ff30
00095494: 609c0000 832202e0 607b0000 60bd0000
000954a4: 63230000 38800000
> od open 10 byte
00095484: 0174 0010 0002 0246 0277 0041 0377 0344
0009548c: 0220 0001
> od 12345
warning: word alignment performed
00012344: 480001d8
> user -s 3
MST Segment Regs
  0:0x00000000  1:0x00002c0b  2:0x00004411  3:0x007fffff
  4:0x007fffff  5:0x007fffff  6:0x007fffff  7:0x007fffff
  8:0x007fffff  9:0x007fffff 10:0x007fffff 11:0x007fffff
 12:0x007fffff 13:0x007fffff 14:0x00001004 15:0x007fffff
...
> od 4411:ff3b400 12
004411:ff3b400: 00000000 00000000 2ff22e28 00000000
004411:ff3b410: 00000306 00000000 0002a7ec 000010b0
004411:ff3b420: 82202220 0002a7ec 00000000 0000001c
> od Debug_record
001cc378: 00000000
> od *Debug_record
00000000: 00000000
```

ppd [ProcessorNumber | *]

Displays per-processor data area (PPDA) structures for the specified processor. If no processor is specified, the current processor selected by the **cpu** subcommand is used. If the asterisk argument is given, the PPDA of every enabled processor is displayed.

Aliases = none

```
> ppd
Per Processor Data Area for processor 0
alsave[ 0].....0000000000000000
alsave[ 1].....0000000000000000
alsave[ 2].....0000000000000000
alsave[ 3].....0000000000000000
alsave[ 4].....0000000000000000
alsave[ 5].....0000000000000000
alsave[ 6].....0000000000000000
alsave[ 7].....0000000000000000
alsave[ 8].....0000000000000000
alsave[ 9].....0000000000000000
alsave[10].....0000000000000000
alsave[11].....0000000000000000
alsave[12].....0000000000000000
alsave[13].....0000000000000000
alsave[14].....0000000000000000
alsave[15].....0000000000000000
csa.....003f0eb0
mstack.....003efeb0
fpowner.....00000000
curthread.....e60013ec
...
```

print [type] Address

Does **dbx**-style printing of structures. The **-i** option must be given on the command line to use this feature.

If *type* is omitted, the default type set by the last **print -d** command is used.

Aliases = none

print -d type

Recognized by the **pr**, **str**, or **struct** subcommand aliases. Sets the default type for subsequent print commands to *type*.

Aliases = set prtype

proc [-] [-r] [ProcessSlotNumber]

The **proc** subcommand displays the process table, including the kernel thread count (the number of threads in the process) and state of each process. Use the **-r** flag to display only runnable processes. Use the **-** flag to display a longer listing of the process table.

Aliases = ps, p

```
>p
SLT ST  PID  PPID  PGRP  UID  EUID  TCNT  NAME
  0 a    0    0    0    0    0    1  swapper
      FLAGS: swapped_in no_swap fixed_pri kproc
  1 a    1    0    0    0    0    1  init
      FLAGS: swapped_in no_swap
  2 a   204    0    0    0    0    1  wait
      FLAGS: swapped_in no_swap fixed_pri kproc
...
>p 20
SLT ST  PID  PPID  PGRP  UID  EUID  TCNT  NAME
 20 a  1406    1  1406    0    0    1  ksh
      FLAGS: swapped_in no_swap
> p - 0
SLT ST  PID  PPID  PGRP  UID  EUID  TCNT  NAME
  0 a    0    0    0    0    0    1  swapper
      FLAGS: swapped_in no_swap fixed_pri kproc
Links: *child:0xe30013b0 *siblings:0x00000000 *uid1:0xe3002490
      *ganchor:0x00000000 *pgrp1:0x00000000 *tty1:0x00000000
Dispatch Fields: pevent:0x00000000 *synch:0xffffffff
      lock:0x00000000 lock_d:0x00000000
Thread Fields: *threadlist:0xe6000000 threadcount:1
active:1 suspended:0 local:0 terminating:0
Scheduler Fields: fixed pri: 16 repage:0x00000000 scount:0 sched_pri:0
      *sched_next:0x00000000 *sched_back:0x00000000 cpticks:130
      msgcnt:0 majfltsec:0
Misc: adspace:0x0000340d kstackseg:0x007fffff xstat:0x0000
      *p_ipc:0x00000000 *p_dblist:0x00000000 *p_dbnext:0x00000000
Signal Information:
      pending:hi 0x00000000,lo 0x00000000
      sigcatch:hi 0x00000000,lo 0x00000000 sigignore:hi 0xffffffff,lo 0xffff7ffff
Statistics: size:0x00000000(pages) audit:0x00000000
      accounting page frames:0 page space blocks:0
```

Refer to the **sys/proc.h** header file for the structure definition.

qrun

The **qrun** subcommand displays the list of scheduled streams queues. If there are no queues found for scheduling, the **qrun** subcommand will print a message stating there are no queues scheduled for service.

This subcommand can be issued from **crash** on either a running system or a system dump.

Aliases = none

```
> qrun
  QUEUE
  59d5a74
```

queue [Address]

The **queue** subcommand displays the STREAMS queue. If the address optional parameter is not supplied, **crash** will display information for all queues available. Refer to the `/usr/include/sys/stream.h` file for the queue structure definitions.

If you wish to see the information stored for a read queue, issue the **queue** subcommand with the read queue address specified as the parameter.

When you issue the **queue** subcommand with the address parameter, the column headings do not distinguish between the read queue and the write queue. One queue address will be displayed under the column heading **QUEUE**. The other queue in the pair will be displayed under the column heading **OTHERQ**. The write queue will have a numerically higher address than the read queue.

This subcommand can be issued from **crash** on either a running system or a system dump.

Aliases = **que**

```
> queue
  WRITEQ  QINFO  NEXT  PRIVATE  FLAGS  HEAD  READQ  COUNT  NAME
1802e08c  2268c00 1802e48c 1802e200 0x002a  0 1802e000  0  sth
1802e48c  22b2960 1802cc8c 1807a32c 0x0028  0 1802e400  0 mi_timod
1802cc8c  22ae158  0 1802ec2c 0x8028  0 1802cc00  0  xtiso
1806cc8c  2268c00 1806a88c 1806ca00 0x002a  0 1806cc00  0  sth
1806a88c  22b2960 1806ae8c 1807a06c 0x0028  0 1806a800  0 mi_timod
1806ae8c  22ae158  0 1806ac2c 0x8028  0 1806ae00  0  xtiso
1806ce8c  2268c00 1806c88c 1806c000 0x002a  0 1806ce00  0  sth
1806c88c  22b2960 1806c28c 1807a5ac 0x0028  0 1806c800  0 mi_timod
1806c28c  22ae158  0 1806c42c 0x8028  0 1806c200  0  xtiso
1802c68c  2268c00 1802ca8c 1802c400 0x002a  0 1802c600  0  sth
1802ca8c  2ab9580 1802c88c 1802c200 0x0028  0 1802ca00  0  ldterm
1802c88c  25328d0  0 2aa7130 0x0028  0 1802c800  0  rs
```

quit

Exit from the **crash** command.

Aliases = **q**

search [-sn] name

Search the symbols table for *name*.

-s Prints symbols matching *name* in the **nm** format. Also prints the symbol table entry for the last symbol found.

-n Prevents the search from examining kernel extensions.

search[-n] addr

Search for the symbol with the largest value less than or equal to *addr*.

-n Prevents the search from examining kernel extensions.

segst64 [-p pslot | -t tslot] [-l limit [-s segflag[:value]][, segflag[:value]]...] [-n [start_esid [end_esid]]]

Displays segstate information for a 64-bit process. The segstate for the current process displays unless the **-p** or **-t** flags are specified. All of the segstate entries display unless limited by the **-l** flag or the starting esid, *start_esid* and possible ending esid, *end_esid*. Specifying the **-s** flag limits the display to only those segstate entries matching the given *segflags*, matching pattern types, as well as their corresponding values. The **-l** flag limits the display to a maximum number of entries. The **-n** flag also prints the segnodes for the displayed data. Segnode entries are not included in the count when limiting the data with **-l**.

-p pslot	Specifies the process slot number.
-t tslot	Specifies the thread slot number.
-s segflag[:value]	Limits the display to the segstate entries matching that <i>segflag</i> and <i>value</i> .
-l limit	Specifies the number of entries to print.
-n	Prints the uadnodes for the displayed data.

Aliases = adspace, as, sr

select

Recognized by the **sel** subcommand alias. Displays all select control blocks.

select p proc_slot

Recognized by the **sel** subcommand alias. Displays select control blocks for process in specified slot.

Note: The **p** flag is not prefixed with a **-**(dash).

select dev_id unique_id

Recognized by the **sel** subcommand alias. Displays select control blocks matching the specified device and unique IDs.

set

Display **crash** variables and values.

set allhex [no]

Causes **crash** to use only hex values for both input and output as opposed to a mixture of hex and decimal. Specify **no** to turn this option off.

set edit [emacs | gmacs | none | vi]

Sets command line editing mode.

set fpregs [yes|no|auto]

Specify whether or not floating-point registers should be displayed. If **auto** is used, the *fpeu* variable in the mtsave area determines when to display the registers.

set idarch [ppc|pwr|auto]

Set instruction decode architecture. **auto** detects the architecture from the system.

set logfile [filename]

Set logfile to given name, or turn off logging if no name is given.

set loglevel [0|1|2]

Set logging granularity to:

- 0 coarse-only commands will be logged.
- 1 medium-commands and output to terminal will be logged.
- 2 fine-commands and all outputs will be logged, including redirected.

set prtype [type]

Set the default print type. This is equivalent to **print -d type**.

set quiet [no]

Suppress error messages concerning missing or swapped out threads and processes. Specify **no** to turn off this option.

socket [-]

The **socket** subcommand displays the system socket structures. Use the **-** flag to also display the socket buffers.

Aliases = sock

```
> sock
1802e800: type:0x0002 (DGRAM) opts:0x0000 ()
        state:0x0082 (ISCONNECTED|PRIV) linger:0x0000
        pcb:0x1807a6c0 proto:0x000bbd30 q0:0x00000000 qlen:0
        q:0x00000000 qlen:0 qlimit:0 head:0x00000000
        timeo:0 error:0 oobmark:0 pgid:0
18074400: type:0x0002 (DGRAM) opts:0x0000 ()
        state:0x0080 (PRIV) linger:0x0000
        pcb:0x1807a100 proto:0x000bbd30 q0:0x00000000 qlen:0
        q:0x00000000 qlen:0 qlimit:0 head:0x00000000
        timeo:0 error:0 oobmark:0 pgid:0
...
```

Refer to the **sys/socket.h** header file for structure definitions.

sr64 [-p pslot | -t tslot] [-l limit [-n [start_esid [end_esid]]]

Recognized by the **segst** and **seg** subcommand aliases. Displays the effective segment IDs (esid) and their corresponding segvals for a 64-bit process. If you do not specify the **-p** or **-t** flags, **sr64** uses the current process. Otherwise, it uses **pslot** as the process slot number for the desired process, or **tslot** as the thread slot number of a thread contained within the desired process. It lists all entries in the address space unless a starting esid, **start_esid** and possible ending esid, **end_esid** is

given. Also, it stops listing if the number of entries specified by the **-l** flag have printed. Since `adspace_t` holds 16 entries, each line consists of an `esid`, its corresponding value, and the 3 subsequent values following it in the `adspace_t`. The **-n** flag also prints the uadnodes for the displayed data. uadnode entries are not included in the count when limiting the data with **-l**.

-p pslot Specifies the process slot number.
-t tslot Specifies the thread slot number.
-l limit Specifies the number of entries to print.
-n Prints the uadnodes for the displayed data.

stack [ProcessSlotNumber ThreadSlotNumber]

The **stack** subcommand displays a dump of the kernel stack of a process the kernel thread identified by *ThreadSlotNumber*. The addresses are virtual data addresses rather than true physical addresses. If you do not specify an entry, the subcommand displays information about the last running process kernel thread. You cannot trace the stack of the current running process kernel thread on a running system.

Aliases = s, stk, k, kernel

```
> s
KERNEL STACK:
2ff97a50:      8eaa4          16 2ff97ac8          2
2ff97a60:      90b0          8e8b4 2ff97ad8          0
2ff97a70:              1          26 2ff97ac8 2ff98938
...
```

stat

The **stat** subcommand displays statistics found in the dump. These statistics include the panic message (if there is one), time of crash, and system name. If the memory overlay detection system (MODS) was enabled (shown here as **xmalloc debug**), and it detected a problem, **stat** displays the MODS error message.

Aliases = none

```
> stat
  sysname: AIX
  nodename: riva
  release: 3
  version: 4
  machine: 000018654100
  time of crash: Thu Aug 28 21:03:49 1997
  age of system: 18 min.
  xmalloc debug: enabled
  dump code: 300
  csa: 0x2ff3b400
  exception struct:
    dar: 0x00000000
    dsir: 0x00000000:
    srv: 0x00000000
    dar2: 0x00000000
    dsirr: 0x00000000: (errno) "Error 0"
  Debug kernel error message: A program has tried to access freed
  xmalloc memory.
```

status [ProcessorNumber]

Displays a description of the kernel thread scheduled on the designated processor. If no processor is specified, the **status** subcommand displays information for all

processors. The information displayed includes the processor number, kernel thread identifier, kernel thread table slot, process identifier, process table slot, and process name.

Aliases = none

```
> status 0
CPU    TID  TSLOT    PID  PSLOT  PROC_NAME
  0    1fe1    31    1fd8    31    crash
```

stream

The **stream** subcommand displays the stream head table. The information printed is contained in an internal structure. The following members of this internal structure are described here:

address	Address of stream head
wq	Address of streams write queue
dev	Associated device number of the stream
read_error	Read error on the stream
write_error	Write error on the stream
flags	Stream head flag values
push_cnt	Number of modules pushed on the stream
wroff	Write offset to prepend M_DATA
ioc_id	ID of outstanding M_IOCTL request
pollq	List of active polls
sigsq	List of active M_SETSIGs

The flags structure member, if set, is based on combinations of the following values:

#define	Value	Description
F_STH_READ_ERROR	0x0001	M_ERROR with read error received, fail all read calls.
F_STH_WRITE_ERROR	0x0002	M_ERROR with write error received, fail all writes.
F_STH_HANGUP	0x0004	M_HANGUP received, no more data.
F_STH_NDELOK	0x0008	Do TTY semantics for ONDELAY handling.
F_STH_ISATTY	0x0010	This stream acts a terminal.
F_STH_MREADON	0x0020	Generate M_READ messages.
F_STH_TOSTOP	0x0040	Disallow background writes (for job control).
F_STH_PIPE	0x0080	Stream is one end of a pipe or FIFO.
F_STH_WPIPE	0x0100	Stream is the "write" side of a pipe.
F_STH_FIFO	0x0200	Stream is a FIFO.
F_STH_LINKED	0x0400	Stream has one or more lower streams linked.
F_STH_CTTY	0x0800	Stream controlling tty.

F_STH_CLOSED	0x4000	Stream has been closed, and should be freed.
F_STH_CLOSING	0x8000	Actively on the way down.

This subcommand can be issued from **crash** on either a running system or a system dump.

Aliases = none

```
> stream
ADDRESS WRITEQ MAJ/MIN RERR WERR FLAGS IOCID WOFF PCNT POLQNEXT SIGQNEXT
59b1900 59c2a74 15, 0 0 0 0x0838 0 0 2 0 0
59d3c00 59d5a74 23, 5 0 0 0x0020 0 0 1 0 0
59ffe00 59ff074 23, 4 0 0 0x0020 0 0 1 0 0
5ab4c00 5ab4374 24, 0 0 5 0x0816 0 0 2 0 0
59d3f00 59ee974 24, 1 0 5 0x0816 0 0 2 0 0
59d3800 59dff74 24, 2 0 5 0x0816 0 0 2 0 0
59d3700 5a9c174 24, 3 0 5 0x0816 0 0 2 0 0
59ff800 59ff774 24, 4 0 0 0x0810 0 0 2 0 0
5a94d00 59ee574 24, 5 0 0 0x0830 0 0 2 0 0
5a94600 5a96c74 24, 6 0 5 0x0816 0 0 2 0 0
```

symptom [-e]

Displays the symptom string for a dump. It is not valid on a running system. The optional **-e** option will create an error log entry containing the symptom string, and is normally only used by the system and not entered manually. The symptom string can be used to identify duplicate problems.

tcb [ThreadSlotNumber] . . .

Displays the mstsave portion of the user structures of the named kernel threads (see the **user.h** and **mstsave.h** header files). If you do not specify an entry, information about the last running kernel thread is displayed. Floating point register information is only shown if the **fpeu** value in the mstsave area is set to 1, or if the command **set fpregs yes** has been run. This subcommand replaces the **pcb** subcommand.

Aliases = none

```
> tcb
          UTHREAD AREA FOR SLOT 25 (ucfgxmdbg)
SAVED MACHINE STATE
  curid:0x00001736  m/q:0xb651f200  iar:0x0014397c  cr:0x20228824
  msr:0x000090b0   lr:0x0014395c   xer:0x00000008  kjmpbuf:0x00000000
  backtrack:0x00   tid:0x00000000  fpeu:0x00     excp_type:0x00000000
  ctr:0x00000000   *prevmst:0x00000000 *stackfix:0x00000000  intpri:0x0b
  o_iar:0x00000000 o_toc:0x00000000 o_arg1:0x00000000  excbranch:0x00000000
  o_vaddr:0x00000000
  msr flags: EE ME AL IR DR
  cr flags: | = |   | = | = | |
  Exception Struct
    0x0114807c 0x40000000 0x00000000 0x0114807c 0x00000106
  MST Segment Regs
    0:0x00000000 1:0x00002c0b 2:0x0000759d 3:0x007fffff
    4:0x007fffff 5:0x007fffff 6:0x007fffff 7:0x007fffff
    8:0x007fffff 9:0x007fffff 10:0x007fffff 11:0x007fffff
    12:0x007fffff 13:0x40003c0f 14:0x00001004 15:0x007fffff
  alloc flags: 0xe01f0000 (Seg Regs: 0, 1, 2, 11, 12, 13, 14, 15)
  General Purpose Regs
    0:0x0113b000 1:0x2ff3b2d0 2:0x001cc368 3:0x01148040
    4:0x0000d040 5:0xd0000000 6:0x00000000 7:0x000090b0
    8:0x80000000 9:0x40003c0f 10:0x00000001 11:0xb8000080
```

```

12:0x034123f0 13:0xdeadbeef 14:0xdeadbeef 15:0xdeadbeef
16:0xdeadbeef 17:0xdeadbeef 18:0x20000474 19:0xdeadbeef
20:0xdeadbeef 21:0xdeadbeef 22:0xdeadbeef 23:0xdeadbeef
24:0x2ff3b6e0 25:0x2ff3b400 26:0x100002e4 27:0x42424424
28:0xe3002058 29:0xe60013ec 30:0x00143dac 31:0x00000000
Kernel stack address: 0x2ff3b400

```

thread [-] [-r] [-p ProcessSlotNumber | -a Address | ThreadSlotNumber]

Displays the contents of the kernel thread table. The - (minus) flag displays a longer listing of the thread table. The -r flag displays only runnable kernel threads. The -p flag displays only those kernel threads which belong to the process identified by *ProcessSlotNumber*. The -a flag displays the kernel thread structure at *Address*. If *ThreadSlotNumber* is given, only the corresponding kernel thread is displayed.

Aliases = th

```

> thread 1
SLT ST  TID      PID    CPUID  POLICY PRI CPU   EVENT  PROCNAME
  1 s    1e1      1  unbound  other  3c   0             init
      FLAGS: wakeonsig

> th
SLT ST  TID      PID    CPUID  POLICY PRI CPU   EVENT  PROCNAME
  0 s    3        0      0      FIFO  10  78             swapper
      t_flags: wakeonsig kthread
  1 s   105      1      0      other  3c   0             init
      t_flags: wakeonsig
  2 r   205     204     0      FIFO  7f  78             wait
      t_flags: sig_avail kthread
  3 s   307     306     0      RR    24   0             netm
      t_flags: sig_avail kthread
...
> th - 3
SLT ST  TID      PID    CPUID  POLICY PRI CPU   EVENT  PROCNAME
  3 s   307     306     0      RR    24   0             netm
      t_flags: sig_avail kthread
Links: *procp:0xe3000438 *uthreadp:0x2ff3b400 *userp:0x2ff3b6e0
      *prevthread:0xe6000264 *nextthread:0xe6000264, *stackp:0x00000000
      *wchan1(real):0x00000000 *wchan2(VMM):0x00000000 *swchan:0x00000000
      wchan1sid:0x00000000 wchan1offset:0x00000000
      pevent:0x00000000 wevent:0x00000912 *slist:0x00000000
Dispatch Fields: *prior:0xe6000264 *next:0xe6000264
      polevel:0x00000000 ticks:0x0000 *synch:0xffffffff result:0x00000000
      *eventlst:0x00000000 *wchan(hash):0x00000000 suspend:0x0001
      thread waiting for: event(s)
Scheduler Fields: cpuid:0x0000 scpuid:0x0000 pri: 36 policy:RR
      affinity:0x0000 cpu:0x0000 lpri: 0 wpri:127 time:0xff
sav_pri:0x24
Misc: lockcount:0x00000000 ulock:0x00000000 *graphics:0x00000000
      dispct:0x0000a5ae fpuct:0x00000000 boosted:0x0000
      userdata:0x00000000
Signal Information: cursig:0x00 *scp:0x00000000
      pending:hi 0x00000000,lo 0x00000000 sigmask:hi 0x00000000,lo 0x00000000

```

trace [-r | -m [-f]] [-k | -s] [-r] [ThreadSlotNumber] . . .

The **trace** subcommand displays a kernel stack trace of the process kernel thread identified by *ThreadSlotNumber*. The trace starts at the top of the stack and attempts to find valid stack frames deeper in the stack. By default, the current process kernel thread is used.

When using the **-k** flag, the stack frame addresses indicate the stack frame containing the link register of the function that is displayed. The **-m** flag causes **trace** to display the traceback associated with each mstsave area on the current save area chain, except the first. To see a traceback from the first mstsave area, specify the **-f** flag. When either the **-m** or **-k** flags are used, trace may also show the LR (link register) and top stack frame pointer. These are not part of the stack trace and are therefore marked with an asterisk. The **-s** flag displays saved register information for each stack frame.

Use the **-r** flag to use the kernel frame pointer set up by the **kfp** subcommand as the starting address instead of the frame pointer found in the *SystemImageFile*. The **trace** subcommand stops and reports an error if an invalid frame pointer is encountered.

Aliases = t

```
> t
STACK TRACE:
    .et_wait ()
    .e_sleep ()
    .e_sleep1 ()
    .sleepx ()
    .fifo_read ()
    .fifo_rdwr ()
    .vno_rw ()
    .rwuio ()
    .rdwr ()
    .kreadv ()

> t -k 5
STACK TRACE:
0x2ff3b400 (excpt=2ef636d8:40000000:00004812:2ef636d8:00000106) (intpri=0)
    IAR:      .e_block_thread+23c (00029e04):      1   r0,0xc(r30)
    LR:       .e_block_thread+23c (00029e04)
    2ef62ee8: .e_sleep_thread+ec (0002a454)
    2ef62f48: .netisr_thread+28 (0006f854)
    2ef62f88: .threadentry+18 (00047244)
    2ef62fc8: .low+0 (00000000)
```

tty

Aliases = **term**, **dz**, **dh**

Refer to the **sys/tty.h** header file for the structure definition.

unhide symbol...

Unhide the specified symbol. See “hide symbol...” on page 315.

unhide

Unhide all hidden symbols. See the **hide symbol...** subcommand on “hide symbol...” on page 315.

user [ThreadSlotNumber]

Recognized by the **uarea**, **u_area**, and **u** subcommand aliases. Displays the uthread structure and the associated user structure of the thread identified by *ThreadSlotNumber*. (See the **usr/include/sys/user.h** file for the user structure definition.) If you do not specify the entry, the information about the last running thread displays. The **-s** flag limits the output to segment register information.

Aliases = u, uarea, u_area

```
> u 4
      UTHREAD AREA FOR SLOT 4 (gil)
SAVED MACHINE STATE
  curid:0x00000408 m/q:0x0000ff34 iar:0x0002a7ec cr:0x82002820
  msr:0x000010b0 lr:0x0002a7ec xer:0x00000000 kjmpbuf:0x00000000
  backtrack:0x00 tid:0x00000000 fpeu:0x00 excp_type:0x00000000
  ctr:0x00000000 *prevmst:0x00000000 *stackfix:0x2ff22ea8 intpri:0x00
  o_iar:0x00000000 o_toc:0x00000000 o_arg1:0x00000000 excbranch:0x00000000
  o_vaddr:0x00000000
  msr flags: ME AL IR DR
  cr flags: |
```

var

The var subcommand displays the tunable system parameters.

Aliases = tune, tunable, tunables

```
> var
buffers 20
files 255
e_files 255
procs 131072
e_procs 45
threads 262144
e_threads 40
c_lists 16384
maxproc 40
iostats 1
locks 200
e_locks 4443672
```

vfs [-] [Vfs SlotNumber]

The **vfs** uses the specified *Vfs SlotNumber* to display an entry in the **vfs** table. Use the - flag to display the vnodes associated with the **vfs**. The default displays the entire **vfs** table.

Aliases = m, mnt, mount

```
> vfs 3
VFS ADDRESS TYPE OBJECT STUB NUM FLAGS PATHS
  3 1a62494 jfs 1a6d47c 1a6d650 5 D /dev/hd1 mounted over /u
      flags: C=disconnected D=device I=remote P=removable
             R=readonly S=shutdown U=unmounted Y=dummy

> vfs - 3
VFS ADDRESS TYPE OBJECT STUB NUM FLAGS PATHS
  3 1a62494 jfs 1a6d47c 1a6d650 5 D /dev/hd1 mounted over /u
ADDRESS VFS MVFS VNTYPE FSTYPE COUNT ISLOT INODE FLAGS
1a6e0ac 3 - vreg jfs 1 - 18f82c0
1a6e218 3 - vreg jfs 1 - 18f8770
1a6e24c 3 - vreg jfs 1 - 18f8590
1a6e17c 3 - vdir jfs 3 - 18f7f00
1a6dea4 3 - vreg jfs 2 - 18f65b0
1a6dfa8 3 - vdir jfs 5 - 18f6100
1a6d47c 3 - vdir jfs 1 - 18ea580 vfs_root
```

Refer to the **sys/vfs.h** header file for structure definitions.

vnode [VNodeAddress]

The **vnode** subcommand displays data at the specified *VNodeAddress* as a **vnode**. *VNodeAddress* must be specified in hexadecimal notation. The default is to display all **vnodes** in the **vnode** table.

Aliases = none

```
> vnode 1a6e078
ADDRESS VFS MVFS VNTYPE FSTYPE COUNT ISLOT DATAPTR FLAGS
1a6e078 0 - vreg jfs 4 - 18f6790
Total VNODES printed 1
```

Refer to the `sys/vnode.h` header file for the structure definition.

xmalloc

Recognized by the `xm` and `malloc` subcommand aliases. Prints information concerning the allocation and usage of kernel memory (the `pinned_heap` and the `kernel_heap`). If a MODS-related system crash has occurred, `xmalloc` displays information about the address involved in the crash (if available).

xmalloc [addr]

Recognized by the `xm` and `malloc` subcommand aliases. Prints `xmalloc` information about *addr*.

xmalloc -s [addr]

Recognized by the `xm` and `malloc` subcommand aliases. Prints debug `xmalloc` allocation records associated with *addr*.

Note: The `-s` flag requires that the memory overlay detection system (MODS) has been turned on.

xmalloc -h [addr]

Recognized by the `xm` and `malloc` subcommand aliases. Prints MODS `xmalloc` free list records associated with *addr*.

Note: The `-h` flag requires that the memory overlay detection system (MODS) has been turned on.

xmalloc [-l] -f

Recognized by the `xm` and `malloc` subcommand aliases. Prints allocation records on free list from earliest-freed to latest-freed. The `-l` flag prints a long listing.

Note: The `-f` flag requires that the memory overlay detection system (MODS) has been turned on.

xmalloc [-l] -a

Recognized by the `xm` and `malloc` subcommand aliases. Prints the allocation record table. The `-l` flag prints a long listing.

Note: The `-a` flag requires that the memory overlay detection system (MODS) has been turned on.

xmalloc [-l] -p pageno

Recognized by the `xm` and `malloc` subcommand aliases. Prints page descriptor information for page *pageno*. The `-l` flag prints additional information.

xmalloc -d [addr]

Recognized by the **xm** and **malloc** subcommand aliases. Prints debug **xmalloc** allocation record hash chain associated with the record hash value for *addr*.

xmalloc -v

Recognized by the **xm** and **malloc** subcommand aliases. Verify allocation trailers of allocated records, and free fill patterns of freed records.

Note: The **-v** flag requires that the memory overlay detection system (MODS) has been turned on.

Low Level Kernel Debugger (LLDB)

This chapter provides information about the available procedures for debugging a device driver which is under development. Topics discussed include:

- LLDB Kernel Debug Program
- LLDB Kernel Debug Program Commands
- Maps and Listing as Tools for the LLDB Kernel Debug Program
- Using the LLDB Kernel Debug Program
- Error Messages for the LLDB Kernel Debug Program

LLDB Kernel Debug Program

The Low Level Kernel Debug Program (LLDB) on AIX 4.3 will provide new commands to display new kernel data added to 64-bit applications support. The LLDB Kernel Debug Program on AIX 4.3 would now be able to handle all the 64-bit user addresses or data that are typed in on the command line wherever applicable. The user address space ranges from 0x0 through 0xFFFFFFFFFFFFFFFF.

In AIX 4.3, though the kernel execution is always 32-bit, it is possible that while in the LLDB Kernel Debug Program, the currently active process be a 64-bit program and the current execution mode is in Problem State. It is possible to enter such state (henceforth mentioned as 64-bit context) by invoking the debugger from the native or tty keyboard key sequence.

When the debugger is in 64-bit context, the display format of the screen would be different on AIX 4.3 in order to display 64-bit wide GPRs and other relevant 64-bit wide register contents.

The LLDB Kernel Debug Program on AIX 4.3 also supports debugging 64-bit real mode kernel code. The screen display format would be similar to that of the 64-bit context debug mode.

Use the kernel debug program for debugging the kernel, device drivers, and other kernel extensions. The kernel debug program provides the following functions:

- Setting breakpoints within the kernel or within kernel extensions
- Formatting and displaying selected kernel data structures
- Viewing and modifying memory for any kernel data
- Viewing and modifying memory for kernel instructions
- Modifying the state of the machine by altering system registers
- Displaying 64-bit real mode context when stopped in 64-bit real mode code.
- Setting breakpoints and watchpoints in 64-bit real mode code.
- Allowing step execution of 64-bit real mode code.

- Executing a stack trace back when stopped in 64-bit real mode code.

Loading and Starting the LLDB Kernel Debug Program

The kernel debug program must be loaded by using the **bosboot** command before it can be started. Use either of the following commands:

```
bosboot -a -d /dev/ipldevice -D
```

OR

```
bosboot -a -d /dev/ipldevice -I
```

The **-D** flag causes the kernel debugger program to be loaded. The **-I** flag also causes the kernel debug program to be loaded, but it is also invoked at system initialization. This means that when the system starts, it will trap the kernel debug program.

After issuing the **bosboot** command, you must restart the machine. The kernel debug program will not be loaded until the system is restarted. When started, the debug program accepts the commands described in “LLDB Kernel Debug Program Commands” on page 338.

If the kernel debug program is invoked during initialization, use the **go** command to continue the initialization process.

Notes:

1. The debug program disables all external interrupts while it is in operation.
2. On AIX Version 4.1.4 systems (and later), it is no longer required that the key switch be in the service position to operate the kernel debugger. To debug the kernel program, use the **bosboot** command with the **-D** or **-I** flags. This change was instituted to allow use of the debugger on systems without a key.

Using a Terminal with the LLDB Kernel Debug Program

The debug program opens an asynchronous ASCII terminal when it is first started, and subsequently upon being started due to a system halt. Native serial ports are checked sequentially starting with port 0 (zero). Each port is configured at 9600 bps, 8 bits, and no parity. If carrier detect is asserted within 1/10 seconds, then the port is used. Otherwise, the next available native port is checked. This process continues until a port is opened or until every native port available on the machine has been checked. If no native serial port is opened successfully, then the result is unpredictable.

You can only display the kernel debugger on an ASCII terminal connected to a native serial port. The kernel debugger does *not* support any displays connected to any graphics adapters. The debugger has its own device driver for handling the display terminal. It is also possible to connect a serial line between two machines and define the serial line port as the port for the console. In that case, use the **cu** command to connect to the target machine and run the debugger.

Note: If a serial device, other than a terminal connected to a native serial port, is selected by the kernel debugger, the system may appear to hang up.

Entering the LLDB Kernel Debug Program

It is possible to enter the kernel debug program using one of the following procedures:

- From a native keyboard, press Ctrl-Alt-Numpad4.
- From the tty keyboard, enter Ctrl-4 (IBM 3151 terminals) or Ctrl-\ (BQ 303, BQ 310C, and WYSE 50).
- The system can enter the debugger if a breakpoint is set. To do this, use the **break** debugger command. See “Breakpoints” on page 338 and “Setting Breakpoints” on page 381 for information on setting a breakpoint.
- The system can also enter the debugger by calling the **brkpoint** subroutine from C code. The syntax for calling this subroutine is:

```
brkpoint();
```

- The system can also enter the debugger if a static debug trap (SDT), is compiled into the code. To do this, place the assembler language instruction:

```
t 0x4, r1 r1
```

at the desired address. One way to do this is to create an assembler language routine that does this, then call it from your driver code.

Note: After the debug program is started, SDTs are treated the same as other processor instructions. The **step** command (see “step Command for the LLDB Kernel Debug Program” on page 365) can be used to step over SDTs. The **go** command or **loop** command can be used to resume execution at the instruction following the SDT.

- The system can also enter the debugger if a system halt is caused by a fatal system error. In such a case, the system creates a log entry in the system log and if the kernel debugger is available, calls the kernel debugger. A system dump is generated on exit from the debugger.

If the kernel debug program is not available (nothing happens when you type in the above key sequence), you must load it. To do this, see “Loading and Starting the LLDB Kernel Debug Program” on page 333.

Note: You can use the **crash** command to determine whether the kernel debug program is available. Use the **od** subcommand:

```
# crash  
>od dbg_avail
```

If the **od** subcommand returns a 0 or 1, the kernel debug program is available. If it returns 2, the debug program is not available.

Debugging Multiprocessor Systems

On multiprocessor systems, entering the kernel debug program stops all processors (except the current processor running the debug program itself). Generally, when the debugger returns control to the program being debugged, other processors are released to run again. However, other processors are not released during the **step** command. On multiprocessor systems, the kernel debug program prompt indicates the current processor as follows:

```
<ProcessorNumber>>
```

where *ProcessorNumber* identifies the current processor. Example: 3>

LLDB Kernel Debug Program Concepts

When the kernel debugger is invoked, it is the only running program. All processes are stopped, interrupts are disabled, and the cache is flushed. The system creates a new **mstsave** (machine state save) area for use by the debugger. However, the data displayed by the debugger comes from the **mstsave** area of the thread that was interrupted when the debugger was entered. After exiting from the kernel debugger, all the processes will continue to run unless you entered the debugger through a system halt.

The data displayed by the debugger in 64-bit context comes from the **mstsave64** area of the thread (of a 64-bit process) that was interrupted.

Commands

The kernel debug program must be loaded and started before it can accept commands.

Once in the kernel debugger, use the commands to investigate and make alterations. Each command has an alias or a shortened form. This is the minimum number of letters required by the debugger to recognize the alias as unique. See “LLDB Kernel Debug Program Commands” on page 338 for lists and descriptions of the commands.

Numeric Values and Strings

Numeric arguments are required to be hexadecimal for all commands except the **loop** and **step** commands and the **slotnumber** option of the **drivers** command, which all take a numeric count in decimal. Decimal numbers must either be decimal constants (0-9), variables, or expressions involving both options (see “Expressions” on page 337). Hexadecimal numbers can also include the letters A through F.

In some cases, only numeric constants are allowed. Wherever appropriate, this restriction is clearly identified.

On the other hand, a string is either a hexadecimal constant or a character constant of the form “*String*”. Hexadecimal constants can be no longer than 8 digits. Double quotation marks separate string constants from other data.

Variables

Variable names must start with a letter and can be up to eight characters long. Variable names cannot contain special symbols. Variables usually represent locations or values which are used again and again. A variable must not represent a valid number. Use the **set** command (see “set Command for the LLDB Kernel Debug Program” on page 361) to define and initialize variables. Variables can contain from 1 to 4 bytes of numeric data or up to 32 characters of string data. You can release a variable with the **reset** command (see “reset Command for the LLDB Kernel Debug Program” on page 358). You cannot use the **reset** command with reserved variables.

For example:

```
set name 1234 Sets your variable called name=1234
set s8 820c00e0 Sets seg reg 8 to point to the IOCC
```

Note that **s8** is a reserved variable.

Reserved Variables

There is a set of variables that have a reserved meaning for the LLDB Kernel Debug Program. You can reference and change these variables, but they represent the actual hardware registers. There are also two variables (*fx* and *org*) reserved for use by the kernel debug program, which can be changed or set. If you change any registers while in the kernel debug program, the change remains in effect when you leave the kernel debug program. The reserved variables are:

asr	Address space register
bat0l	BAT register 0, lower.
bat0u	BAT register 0, upper.
bat1l	BAT register 1, lower.
bat1u	BAT register 1, upper.
bat2l	BAT register 2, lower.
bat2u	BAT register 2, upper.
cppr	Current processor priority register.
cr	Condition register.
ctr	Count register.
dar	Data address register.
dec	Decrementer.
dsier	Data storage interrupt error register.
dsisr	Data storage interrupt status register.
eim0	External interrupt mask (low).
eim1	External interrupt mask (high).
eis0	External interrupt summary (low).
eis1	External Interrupt summary (high).
fp0-fp31	Floating point registers 0 through 31.
fpscr	Floating point status and control register.
fx	Address of the last item found by the find command.
iar	Instruction Address Register (program counter). Points to the current instruction.
lr	Link register.
mq	Multiply quotient.
msr	Machine State register.
org	The current value of origin. It is useful to set this to the program load point.
peis0	Pending external interrupt status register 0.
peis1	Pending external interrupt status register 1.
r0 - r31	General Purpose Registers 0 through 31. These registers have the following usage conventions: r0 Used on prologs. Not preserved across calls. r1 Stack pointer. Preserved across calls. r2 TOC. Preserved across calls. r3 - r10 Parameter list for a procedure call. The first argument is r3, the second is r4 and so on until r10 is the 8th argument. These registers are not preserved across calls. r11 Scratch. Pointer to FCN; DSA pointer to <code>int proc(env)</code> . r12 PL8 exception return. Value preserved across calls. r13-r31 Scratch. Value preserved across calls.
rtcl	Real Time clock (nano seconds).
rtcu	Real Time clock (seconds).

s0-s15	Segment registers. If a segment register is <i>not</i> in use, it has a value of 007FFFFF.
sdr0	Storage description register 0.
sdr1	Storage description register 1.
sisr	Data Storage-Interrupt Status register.
srr0	Machine status save/restore 0.
srr1	Machine status save/restore 1.
tbl	Time base register, lower.
tbu	Time base register, upper.
tid	Transaction register (fixed point).
xer	Exception register (fixed point).
xirr	External interrupt request register.

Expressions

The LLDB Kernel Debug Program does not allow full expression processing. Expressions can only contain decimal or hex constants, variables and operators. The variable operators include:

+	addition
-	subtraction
*	multiplication
/	division
>	dereference

The **>** operator indicates that the value of the preceding expression is to be taken as the address of the target value. The contents of the address specified by the evaluated expression are used in place of the expression.

You can enter expressions in the form *Expression(Expression)*. This form causes the two expressions to be evaluated separately and then added together. This form is similar to the base address syntax used in the assembler.

You can also enter expressions in the form *+Expression* or *-Expression*. This form causes the expression to be added to or subtracted from the origin (the reserved variable **org**.)

Expressions are processed from left to right only. The type of data specified must be the same for all terms in the expression.

Pointer Dereferences

A pointer dereference can be used to refer indirectly to the contents of a memory location. For example, assume that the 0xC50 location contains a counter. An expression of the form `c50>` can be used to refer to the counter. Any expression can be placed before the **>** (greater than) operator, including an expression involving another **>** operator. In this case multiple levels of indirection are used. To extend the example, if the FF7 location contains the C50 value, the expression `FF7>>` refers to the above counter.

The following examples show how to use a pointer dereference with the **alter** command (see “alter Command for the LLDB Kernel Debug Program” on page 341):

```
alter 124> 0582
alter addr1>+8 d96e
```

In the first case, data is placed into the memory location pointed to by the word at the 124 address. The second case places the `d96e` variable into memory at the address computed by adding 8 to the word at the address in the `addr1` variable.

Breakpoints

The LLDB Kernel Debug Program creates a table of breakpoints that it internally maintains. The **break** command creates breakpoints (see “break Command for the LLDB Kernel Debug Program” on page 342). The **clear** command clears breakpoints (see “clear Command for the LLDB Kernel Debug Program” on page 344). When the breakpoint is set, the debugger temporarily replaces the corresponding instruction with the trap instruction. The instruction overlaid by the breakpoint operates when you issue a **step** command or **go** command.

A breakpoint can only be set if the instruction is not paged out. Breakpoints should not be set in any code used by the debugger.

For more information, see “Setting Breakpoints” on page 381.

LLDB Kernel Debug Program Commands

View a list of the LLDB Kernel Debug Program Commands grouped by:

- Alphabetical order (see 338)
- Task Category (see “LLDB Kernel Debug Program Commands grouped by Task Category” on page 340)

View detail descriptions of the LLDB Kernel Debug Program Commands (see “Descriptions of the LLDB Kernel Debug Program Commands” on page 341)

LLDB Kernel Debug Program Commands grouped in Alphabetical Order

The following table shows the LLDB Kernel Debug Program commands in alphabetical order:

Command	Alias	Description
alter	a	Alters memory.
back	b	Decrements the Instruction Address Register (IAR).
break	br	Sets a breakpoint.
breaks	breaks	Lists currently set breakpoints.
buckets	bu	Displays contents of kmembucket kernel structures.
clear	cl	Clears (removes) breakpoints.
cpu	cp	Sets the current processor or shows processor states.
display	d	Displays a specified amount of memory.
dmodsw	dm	Displays the STREAMS driver switch table.
drivers	dr	Displays the contents of the device driver (devsw) table.
find	f	Finds a pattern in memory.
float	fl	Displays the floating point registers.
fmodsw	fm	Displays the STREAMS module switch table.
fs	fs	Displays the internal file system tables.

Command	Alias	Description
go	g	Starts the program running.
help	h	Displays the list of valid commands.
loop	l	Run until control returns to this point.
map	m	Displays the system loadlist.
mblk	mb	Displays the contents of message block structures.
mst64	ms	Displays mstsave64 of a 64-bit process.
netdata	net	Displays the mbuf, ndd, socket, inpcb, and tcpcb data structures.
next	n	Increments the IAR.
origin	o	Sets the origin.
ppd	pp	Displays per-processor data.
proc	pr	Displays the formatted process table.
queue	que	Displays contents of STREAMS queue at specified address.
quit	q	Ends a debugging session.
reason	rea	Displays the reason for entering the debugger.
reboot		Reboots the machine.
reset	r	Releases a user-defined variable.
screen	s	Displays a screen containing registers and memory.
segst64	seg	Displays the states of all memory segments of a 64-bit process.
set	se	Defines or initialize a variable.
sregs	sr	Displays segment registers.
sr64		Displays segment registers only in 64-bit context.
st	st	Stores a fullword in memory.
stack	sta	Displays a formatted kernel stack trace.
stc	stc	Stores one byte in memory.
step	ste	Performs an instruction single-step.
sth	sth	Stores a halfword in memory.
stream	str	Displays stream head table.
swap	sw	Switches from the current display and keyboard to another RS232 port.
sysinfo	sy	Displays the system configuration information.
thread	th	Displays thread table entries.
trace	tr	Displays formatted trace information.
trb	trb	Displays the timer request blocks.
tty	tt	Displays the tty structure.
un		Displays the assembly instruction(s).
user	u	Displays a formatted user area.
user64		Displays the user structure of a 64-bit process.
uthread	ut	Displays the uthread structure.
vars	v	Displays a listing of the user-defined variables.

Command	Alias	Description
vmm	vm	Displays the virtual memory data structure.
watch	w	Watches for load and/or store at an address.
xlate	x	Translates a virtual address to a real address.

LLDB Kernel Debug Program Commands grouped by Task Category

The kernel debug program commands can be grouped into the following task categories:

- “Displaying Registers”
- “Modifying Registers”
- “Setting, Specifying, and Deleting Breakpoints”
- “Displaying Data”
- “Manipulating Memory” on page 341
- “Controlling the Debugger” on page 341

Displaying Registers

cpu	Selects the current processor.
float	Displays the floating-point registers.
origin	Sets the origin of the IAR.
screen	Displays a screen containing registers and memory.
sr64	Displays the segment registers of a 64-bit process.
sregs	Displays segment registers.

Modifying Registers

back	Decreases the instruction address register (IAR).
next	Increments the IAR.
set	Define or initialize a user-defined variable.

Setting, Specifying, and Deleting Breakpoints

break	Sets a breakpoint.
breaks	Lists currently set breakpoints.
clear	Removes breakpoints.
go	Starts the operation of the program following a breakpoint or static debug trap.
loop	Operates until control returns to this point a number of times.
step	Performs a single-step instruction.
watch	Watches for load and/or store at address.

Displaying Data

buckets	Displays statistics on the <i>net_malloc</i> kernel memory pool by bucket size.
display	Displays a specified amount of memory.
dmodsw	Displays the internal STREAMS driver switch table.
drivers	Displays the contents of the device driver (devsw) table.
fmodsw	Displays the internal STREAMS module switch table.
fs	Displays the internal file system tables.

map	Displays a system load list.
mblk	Displays the contents of the STREAMS message blocks.
mst64	Displays the mstsave64 structure of a 64-bit process.
netdata	Displays the mbuf, ndd, socket, inpcb and tcpcb data structures.
ppd	Displays a formatted per-processor data structure.
proc	Displays the formatted process table.
queue	Displays the contents of the STREAMS queues.
reason	Displays the reason for entering the debugger.
screen	Displays a screen containing registers and memory.
segst64	Display the states of all memory segments of a 64-bit process.
stack	Displays a formatted kernel stack trace.
stream	Displays the contents of the stream head table.
sysinfo	Displays the system configuration information.
thread	Displays the formatted thread table.
trace	Displays formatted trace information.
trb	Displays the timer request blocks.
tty	Displays tty information.
un	Displays the assembly instruction(s).
user	Displays a formatted user area.
user64	Displays the user64 structure of a 64-bit process.
uthread	Displays a formatted uthread structure.
vmm	Displays the virtual memory information menu.

Manipulating Memory

alter	Alters memory.
display	Displays a specified amount of memory.
find	Finds a pattern in memory.
st	Stores a fullword in memory.
stc	Stores 1 byte in memory.
sth	Stores a halfword in memory.
vmm	Displays the virtual memory information menu.
xlate	Translates a virtual address to a real address.

Controlling the Debugger

help	Displays the list of valid commands.
quit	Ends the debugging session.
reboot	Reboots the machine.
reset	Clear a user-defined variable.
set	Define or initialize a user-defined variable.
swap	Switches from the current display and keyboard to an RS-232 port.
vars	Displays a listing of user-defined variables.

Descriptions of the LLDB Kernel Debug Program Commands

This includes a description of each of the kernel debug program commands. The commands are in alphabetical order.

alter Command for the LLDB Kernel Debug Program

Purpose

Alters a memory location to the hexadecimal value entered.

Syntax

— **alter** — *Address* — *Data* —|

Description

The **alter** command changes the memory location specified by the *Address* parameter to the hexadecimal value specified by the *Data* parameter. The **alter** command can be used to change one or several bytes of memory. The number of bytes modified with this command depends on the number of bytes you specified. If you specified an odd number of hexadecimal digits, only the first four bits of the last byte are changed.

The **alter** command cannot be used to modify storage to the value of a variable or an expression. Instead, use the **st** command, the **stc** command, or the **sth** command.

Examples

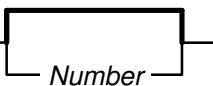
1. To store the 16-bit ffff value at the 1000 address, enter:
alter 1000 ffff
2. To store the 8-bit 2C value in the high-order byte at the 1000 address, enter:
a 1000 2C

back Command for the LLDB Kernel Debug Program

Purpose

Decreases the instruction address register (IAR).

Syntax

— **back** —| 

Description

The **back** command decreases the IAR by the number of bytes specified by the *Number* parameter and displays the new current instruction.

Examples

1. To decrement the IAR by 4 bytes, enter:
back
2. To decrement the IAR by 16 bytes, enter:
b 16

break Command for the LLDB Kernel Debug Program

Purpose

Sets a breakpoint.

Syntax

— **break** — Address —

Description

The **break** command sets a breakpoint in a program at the address specified by the *Address* parameter. The *Address* parameter should be a hexadecimal expression. A breakpoint starts the loaded debug program when the instruction at the specified address is run.

There is a maximum of 32 breakpoints.

Examples

1. To set a breakpoint at the instruction address register (IAR), enter: <PRE >break
2. To set a breakpoint at address 521A, enter:
break 521a
3. To set a breakpoint at A0+8300, enter:
br 8300+A0
4. To set a breakpoint at the origin plus A0, enter:
break +A0
5. To set a breakpoint at the address in the link register, enter:
break lr

breaks Command for the LLDB Kernel Debug Program

Purpose

Lists the current breakpoints, and the watchpoint.

Syntax

— **breaks** —

Description

The **breaks** command lists all currently active breakpoints. For each breakpoint, an offset into a segment is given along with the segment register value at the time the breakpoint was set. This information is required to distinguish between breakpoints set at identical offsets from different segment register values.

Following the list of breakpoints, a currently active watchpoint, an offset into a segment is given along with the segment register value and access value, namely load, store or both, at the time the watchpoint is set.

buckets Command for the LLDB Kernel Debug Program

Purpose

Displays statistics on the *net_malloc* kernel memory pool by bucket size.

Syntax

— **buckets** —|

Description

The **buckets** command displays the contents of the **kmembucket** kernel structures. These structures contain information on the *net_malloc* memory pool by size of allocation.

All output values are printed in hexadecimal format.

This command can also be invoked via the alias, **bu**.

Example

To display **kmembucket** kernel structure for offset 0 and allocation size of 2 enter:

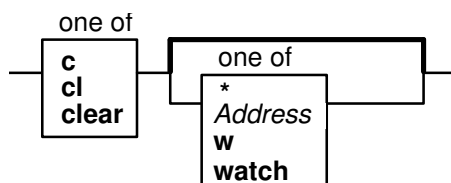
```
buckets
```

clear Command for the LLDB Kernel Debug Program

Purpose

Removes one or all breakpoints and the watchpoint.

Syntax



Description

The **clear** command removes one or all breakpoints, or a watchpoint. The *Address* parameter specifies the location of the breakpoint to be removed. If you specify no flags, the breakpoint pointed to by the instruction address register (IAR) is removed. The **clear** command can be initiated by entering **clear**, **c**, or **cl** at the command line.

Addresses are maintained as offsets from the start of their segment. In the event that two breakpoints are set at the same offset at the start of two different segments, and one breakpoint is then removed, the address specified to the **clear** command is not unique. In this case, each of the conflicting segment IDs are displayed, and the **clear** command displays a prompt requesting the ID of the segment whose breakpoint you want to remove.

The **clear** command, when specified with **watch** or **w** flag, clears the watchpoint.

Examples

1. To clear the breakpoint at the IAR, enter:
`clear`
2. To clear the breakpoint at the 10000200 address, enter:
`cl 10000200`
3. To clear all breakpoints, enter:
`clear *`
4. To clear the watchpoint, enter:
`clear w`

cpu Command for the LLDB Kernel Debug Program

Purpose

Switches the current processor, and reports the kernel debug state of processors.

Syntax



Description

The `cpu` command places the processor specified by the *ProcessorNumber* parameter in debug mode; the processor enters the debugger and is ready to accept commands. The processor where the debugger was previously running is stopped. This command is available only on multiprocessor systems.

If no processor is specified, the `cpu` command displays the kernel debug state of each processor. The possible states are as follows:

Debug	The processor has entered the debugger.
Stopped	The processor has been stopped by another processor in the debug state.
Waiting	The processor has hit a breakpoint while another processor is in the debug state, without having been stopped by the other processor. A particular example is the race condition where two processors both hit breakpoints. One of the processors will enter the debug state; the other will enter the waiting state.

Example

To select the first processor, enter:

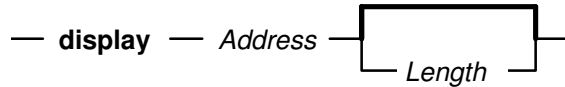
```
cpu 0
```

display Command for the LLDB Kernel Debug Program

Purpose

Displays a specified amount of memory.

Syntax



Description

The **display** command displays memory storage, starting at the address specified by the *Address* parameter. The *Length* parameter indicates the number of bytes to display, and has a default value of 16.

The **display** command displays the contents of the specified region of memory in a two-column format. The left column displays the contents of memory in hexadecimal, and the right column displays the printable ASCII representation of the hexadecimal data.

The **display** command also shows the exact amount of storage requested when you specify a length of 1, 2, or 4 bytes. In this instance, it uses the processor load character, load halfword, or load fullword instruction, respectively. These instructions should be used when displaying input and output address space. Any other value for the *Length* parameter causes memory to be loaded one byte at a time.

Examples

1. To display 16 bytes at the IAR, enter:
`display iar`
2. To display 12 bytes at address 152F, enter:
`d 152F 12`
3. To display 16 bytes at the origin + B7, enter:
`display +B7`
4. To display 16 bytes at the address in r3, enter:
`disp r3`
5. To display from the address contained in the address in r3, enter:
`d r3>`

dmodsw Command for the LLDB Kernel Debug Program

Purpose

Displays the internal STREAMS driver switch table.

Syntax



Description

The **dmodsw** command displays the internal STREAMS driver switch table, one entry at a time. By pressing the Enter key, you can walk through all the **dmodsw**

entries in the table. The contents of the first entry are meaningless except for the *d_next* pointer. When the last entry has been reached, the **dmodsw** command will print the message, "This is the last entry."

The information printed is contained in an internal structure. The following members of this internal structure are described here:

<i>address</i>	Address of dmodsw
<i>d_next</i>	Pointer to the next driver in the list
<i>d_prev</i>	Pointer to the previous driver in the list
<i>d_name</i>	Name of the driver
<i>d_flags</i>	Flags specified at configuration time
<i>d_sqh</i>	Pointer to synch queue for driver-level synchronization
<i>d_str</i>	Pointer to streamtab associated with the driver
<i>d_sq_level</i>	Synchronization level specified at configuration time
<i>d_refcnt</i>	Number of open or pushed count
<i>d_major</i>	Major number of a driver

The *flags* structure member, if set, is based one of the following values:

#define	value	description
F_MODSW_OLD_OPEN	0x1	Supports old-style (V.3) open/close parameters
F_MODSW_QSAFETY	0x2	Module requires safe timeout/bufcall callbacks
F_MODSW_MPSAFE	0x4	Non-MP-Safe drivers need funneling

The synchronization level codes are described in the `/usr/include/sys/strconf.h` header file.

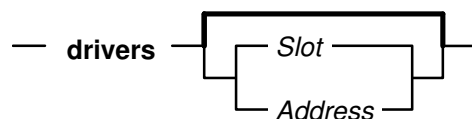
This command can also be invoked via the alias, **dm**.

drivers Command for the LLDB Kernel Debug Program

Purpose

Displays the contents of the device driver (**devsw**) table.

Syntax



Description

The **drivers** command displays the contents of the **devsw** table. If no parameters are specified, then each entry in the table is displayed. If a parameter is specified and is a valid slot number (less than 256), then the corresponding slot in the **devsw** table is displayed. If the parameter is not a valid slot number, then it is understood as an address and the slot with the last entry point prior to the given address is displayed, along with the name of that entry point.

Each **devsw** entry consists of a number of entry points (read, write, and so on) into the specified driver. Each entry consists of a function descriptor, and the address of the function.

Examples

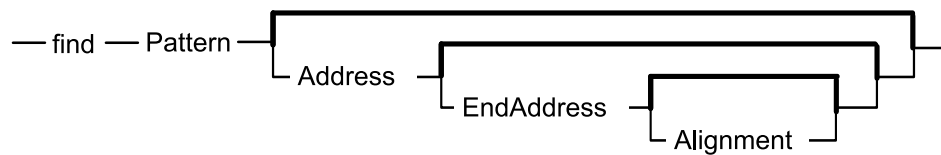
1. To display the entire **devsw** table, enter:
drivers
2. To display the tenth slot of the **devsw** table, enter:
drivers 10
3. To display the last entry point before the address 0x130000F, enter:
dr 130000f

find Command for the LLDB Kernel Debug Program

Purpose

Searches storage.

Syntax



Description

The **find** command searches storage for a pattern beginning at the address specified by the *Address* parameter. If the specified argument is found, the search stops and storage containing the specified argument is displayed. The address of the storage is placed into the *fx* variable.

The following defaults apply to the first execution of the **find** command:

- *Address* = 0
- *EndAddress* = 0x0FFFFFFFFFFFFFFFFF (for 64-bit process)
- *EndAddress* = 0xFFFFFFFF (for 32-bit process)
- *Alignment* = 1 (byte alignment)

An asterisk (*) can be substituted for any of the parameters. An asterisk causes the **find** command to use the value for that parameter that was used in the previous execution of the command.

Examples

1. To find the first occurrence of 7c81 in virtual memory starting at 0, enter:
find 7c81
2. To find the first occurrence of the string TEST, enter:
find "TEST"
3. To find the first occurrence of 7c81 after address 10000, enter:
f 7c81 10000
4. To find the first occurrence of 7c81 between 0 and the user-defined top variable, enter:
f 7c81 0 top
5. To find the first occurrence of 7c81 starting at the last address used, enter:
find 7c81 *
6. To find the first of occurrence of 7c81 starting at the last address used and aligned on a halfword, enter:
f 7c81 * * 2

7. To find the next occurrence of 7c starting at 1 plus the last address at which the **find** command stopped, enter:
f 7c fx+1 * 2
8. To search for the last pattern used, enter:
find *
9. To search for the last pattern starting at the next location (the **find** command remembers the alignment which was used in the previous search), enter:
f * fx+1

float Command for the LLDB Kernel Debug Program

Purpose

Displays floating-point registers.

Syntax

— float —|

Description

The **float** command displays the contents of floating-point registers and other control registers.

In a 64-bit context, the segment register contents will not be displayed.

fmodsw Command for the LLDB Kernel Debug Program

Purpose

Displays the internal STREAMS module switch table.

Syntax

— fmodsw —|

Description

The **fmodsw** command displays the internal STREAMS module switch table, one entry at a time. By pressing the Enter key, you can walk through all the **fmodsw** entries in the table. The contents of the first entry are meaningless except for the *d_next* pointer. When the last entry has been reached, the **fmodsw** command will print the message This is the last entry. This command can also be invoked via the alias, **fm**.

The information printed is contained in an internal structure. The following members of this internal structure are described here:

<i>address</i>	Address of fmodsw
<i>d_next</i>	Pointer to the next module in the list

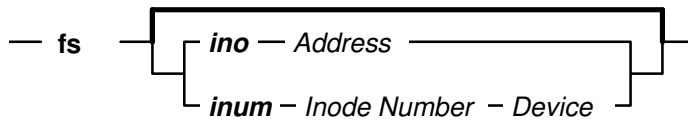
<i>d_prev</i>	Pointer to the previous module in the list
<i>d_name</i>	Name of the module
<i>d_flags</i>	Flags specified at configuration time
<i>d_sqh</i>	Pointer to synch queue for module-level synchronization
<i>d_str</i>	Pointer to streamtab associated with the module
<i>d_sq_level</i>	Synchronization level specified at configuration time
<i>d_refcnt</i>	Number of open or pushed count
<i>d_major</i>	-1

fs Command for the LLDB Kernel Debug Program

Purpose

Displays the internal file system tables.

Syntax



Description

The **fs** command displays the internal inode data structures, vnode data structures and vfs tables. If you specify no flags, the **fs** command displays a menu of commands.

The *flags* structure member, if set, is based one of the following values:

#define	value	description
F_MODSW_OLD_OPEN	0x1	Supports old-style (V.3) open/close parameters
F_MODSW_QSAFETY	0x2	Module requires safe timeout/bufcall callbacks
F_MODSW_MPSAFE	0x4	Non-MP-Safe drivers need funneling

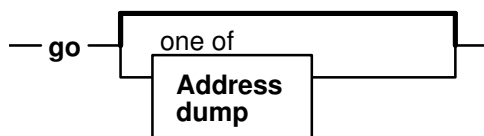
The synchronization level codes are described in the `/usr/include/sys/strconf.h` header file.

go Command for the LLDB Kernel Debug Program

Purpose

Starts executing the program under test or generates a system dump.

Syntax



Description

The **go** command resumes operation of your program. Program operation begins at the current instruction address register (IAR) setting. Specify an address with the *Address* parameter to set the Instruction Address Register (IAR) to a new address and begin running there.

If you specify dump flag, the go command generates a system dump and the machine will halt.

Examples

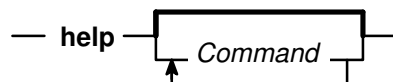
1. To continue running your program at the IAR, enter:
go
2. To set the IAR to 1000 and begin running there, enter:
g 1000

help Command for the LLDB Kernel Debug Program

Purpose

Displays the help screen of the kernel debug program.

Syntax



Description

The **help** command displays a two-line help message for each debug program command. The first line gives the **help** message and the second line gives the syntax of that command. A list of commands or their alias names can be typed as parameters to the help command.

Examples

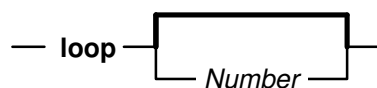
1. To display the list of valid kernel debug program commands, enter:
help
2. To display the **help** messages for Break, Clear and Next commands, enter:
help br c next

loop Command for the LLDB Kernel Debug Program

Purpose

Runs the program being tested until the IAR reaches the current value several times.

Syntax



Description

The **loop** command causes the system to continue running and to stop when the instruction address register (IAR) returns to the current value the number of times specified by the *Number* parameter. All other breakpoints are ignored. The *Number* parameter specifies the number of loops that execute before the debug program regains control, and must be a valid decimal expression. The default value for the *Number* parameter is 1.

The **loop** command is similar to setting a breakpoint at the current IAR, but allows you to stop on a specified instance when the IAR returns to the current point.

Example

To execute until the second time the IAR has the current value, enter:

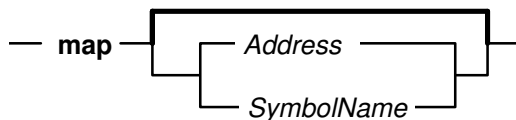
```
loop 2
```

map Command for the LLDB Kernel Debug Program

Purpose

Displays the system load list.

Syntax



Description

The **map** command displays information from the system load list. The system load list is the list of symbols exported from the kernel. If the **map** command is entered with no parameters, then the entire load list is displayed one page at a time. If an address is given, the name and value of the last symbol located before the given address is displayed. If a symbol name is given, then the load list is searched for the symbol and any matching entries are displayed. There can be more than one entry for a given symbol table.

Since the load list contains only symbols exported from the kernel, a given symbol name can be in the kernel but not reported by the **map** command.

The symbol value for a data structure is the address of that data structure. The symbol value for a function is not the address of the function, but the address of the function descriptor. The first word of the function descriptor is the address of the function. For example, if entering `map execexit` displays `0x1000`, then entering `display 1000` displays the address of the `execexit` function in the first word of the displayed memory.

Examples

1. To display the entire load list, enter:

```
map
```

2. To display the symbol with a value closest to `0xe3000000`, enter:

```
m e3000000
```

3. To display the value of the function `execexit`, enter:

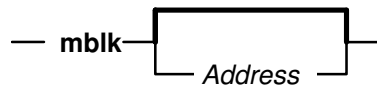
```
map execexit
```

mbk Command for the LLDB Kernel Debug Program

Purpose

Displays the contents of the STREAMS message blocks defined by the `msgb` structure in the `/usr/include/sys/stream.h` header file.

Syntax



Description

The `mbk` command displays the contents of the `msgb` structure that is defined in the `/usr/include/sys/stream.h` headerfile. If you do not specify an *Address*, the command displays the contents of the message blocks of type `M_MBLK` and `M_MBDATA`, as well as displays the address of `mh_freelater`.

The `mh_freelater` parameter is a pointer to the message blocks that are just now freed and are scheduled to be given back to the system, but are not yet given back.

All output values are printed in hexadecimal format.

This command can also be invoked via the alias, `mb`.

Examples

1. To display the contents of the message blocks of type `M_MBLK` and `M_MBDATA`, and the address of `mh_freelater`, enter:

```
mbk
```
2. To display the contents of the message block structure at address `0005ec80`, enter:

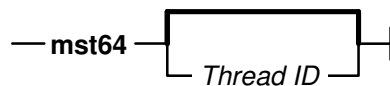
```
mbk 0005ec80
```

mst64 Command for the LLDB Kernel Debug Program

Purpose

Displays the `mstsave64` structure of a 64-bit process. It also displays the kernel remap structure containing all the remapped 64-bit user addresses.

Syntax



Description

The `mst64` command displays the `mstsave64` structure if you specify the thread id of any thread of a 64-bit process. With no parameter specified, the `mstsave64` structure of the currently active thread of a 64-bit process is displayed. In addition, the kernel remap structure containing all the remapped 64-bit user addresses are displayed.

Syntax

— **origin** — *Number* —|

Description

The **origin** command sets the address origin. The origin address specified by the *Number* parameter is added to any hexadecimal expression beginning with a + (plus sign). This command is especially useful when setting breakpoints. Use the **screen** command to display the value of the origin and the origin displacement of the IAR.

The **origin** command also sets the reserved **org** variable. For example, entering `origin 652C0` does the same as entering `set org 652C0`.

Examples

1. To set the origin to 178D, enter:
`origin 178D`
2. To set the origin to 59cc, enter:
`o 59cc`

ppd Command for the LLDB Kernel Debug Program

Purpose

Displays per-processor data.

Syntax

— **ppd** — *ProcessorNumber* —|

Description

The **ppd** command displays the per-processor data structure of the specified processor. If no argument is given, data for the current processor, as selected by the **cpu** command, is displayed.

Note: The **ppd** command is available only on multiprocessor systems.

Examples

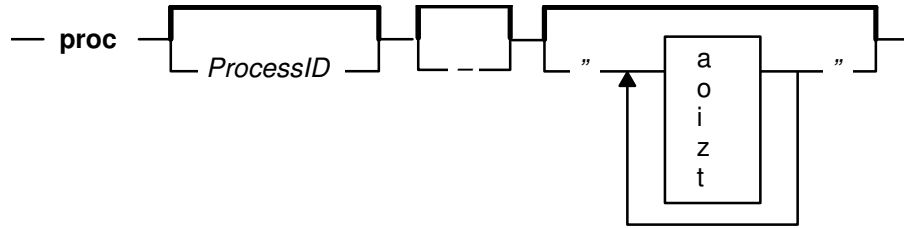
1. To display per-processor data for the current processor, enter:
`ppd`
2. To display per-processor data for processor 2, enter:
`ppd 2`

proc Command for the LLDB Kernel Debug Program

Purpose

Displays the formatted process table.

Syntax



Description

The **proc** command displays the process table in a format similar to the output of the **ps** command, with an * (asterisk) placed next to the currently running process on the processor where the debugger is active. If the *ProcessID* (pid) parameter is specified, the **proc** command displays information pertaining to this process only, and gives more detailed information.

If you specify - flag, then sid, tty, pgrp, ganchor fields of the **proc** table will be displayed. If you specify a string of flags of desired process states, then only the list of process that match the desired process states will be displayed.

A #(pound) is placed next to the process state column for all the 64-bit processes if any.

Flags

List of process states indicated by flags:

a	active
o	swap
i	idle
z	zombie
t	stop

Examples

1. To display the process table, enter:
p
2. To display the process table entry for the process with processID (pid) 1, enter:
proc 1
3. To display some more detailed information still as table of entries, enter:
proc -
4. To display only the entries of active and zombie processes, enter:
p "az"

queue Command for the LLDB Kernel Debug Program

Purpose

Displays the contents of the STREAMS queues.

Syntax

— **queue** — *Address* —|

Description

The **queue** command displays the contents of the STREAMS queue at the specified *Address*. Refer to the `/usr/include/sys/stream.h` header file for the queue structure definition.

In the output, an X indicates that the value is printed in hexadecimal format.

This command can also be invoked via the alias, **que**.

Example

To display the contents of the STREAMS queue stored at address 59c1874, where 59c1874 is a valid queue address, enter:

```
queue 59c1874
```

quit Command for the LLDB Kernel Debug Program

Purpose

Ends the debug program session.

Syntax

— **quit** — dump —|

Description

The **quit** command terminates the debug session. Use this command when you have completed debugging and want to clear all breakpoints. The **quit** command performs the following tasks:

- Clears all breakpoints, and the watchpoint
- Issues the **go** command.

If you specify dump flag, the quit command generates a system dump and the machine will halt.

To use the debug program again after issuing the **quit** command, use one of the keyboard sequences described in “Entering the LLDB Kernel Debug Program” on page 334.

reason Command for the LLDB Kernel Debug Program

Purpose

Displays the reason for entering the debugger.

Syntax

— reason —|

Description

The **reason** command displays the actual reason why the debugger was entered.

reboot Command for the LLDB Kernel Debug Program

Purpose

Reboots the machine.

Syntax

— reboot —|

Description

The **reboot** command reboots the system, after getting confirmation from the user by an input prompt.

Note: The system cannot be rebooted using this command at boot-time debugger prompt.

reset Command for the LLDB Kernel Debug Program

Purpose

Clears a user-defined variable.

Syntax

— reset — *VariableName* —|

Description

The **reset** command clears those variables specified with the *VariableName* parameter. Resetting a variable effectively deletes it, and allows the variable slot to be used again. Currently, 16 user-defined variables are allowed, and when they are all in use, you cannot set any more. Use the **vars** command to display all variables currently set.

Variables that are not user-defined, such as registers, cannot be reset. If you specify a variable that is not user-defined, or a variable that is not defined, an error message is displayed.

Example

To delete the user-defined variable `foo`, enter:

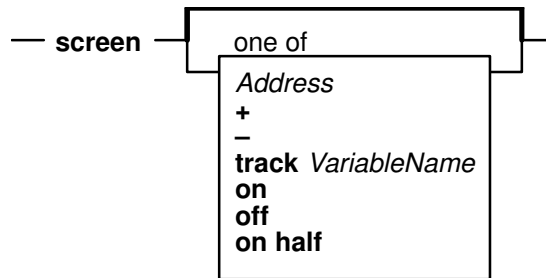
```
reset foo
```

screen Command for the LLDB Kernel Debug Program

Purpose

Displays a screen of data.

Syntax



Description

The **screen** command primarily displays memory and registers, but it is also used to control the format of subsequent **screen** commands. By default, memory is displayed starting at the instruction address register (IAR), or at the variable currently tracked. Variables can be tracked by specifying them with the **track** *VariableName* flag.

The **track** option changes the address that the screen displays as the expression that is being tracked changes. This option is useful in a case where, at a breakpoint, the memory to be displayed is addressed by a register.

You can also use parameters to modify the format of the screen so that only half of the physical screen is used, or even turn off the screen display entirely. The format modification parameters are useful if important information can be scrolled off the screen when the debugger is entered. Restore the default (full) screen by entering:
`screen on`

In 64-bit context, the screen command displays 64-bit wide GPRs and other control registers that exist only on 64-bit hardware. The memory display is limited. All screen operations remain same as before.

Flags

+	Displays the next 0x70 bytes of data.
-	Displays the previous 0x70 bytes of data.
track <i>VariableName</i>	Instructs the screen display to track to the specified variable.
on	Turns the display on.
off	Turns the display off so that the screen display does not appear when the debug program is started. This flag is useful if a slow, asynchronous terminal is used.
on half	Displays only the top half of the display screen. The memory display is omitted.

Examples

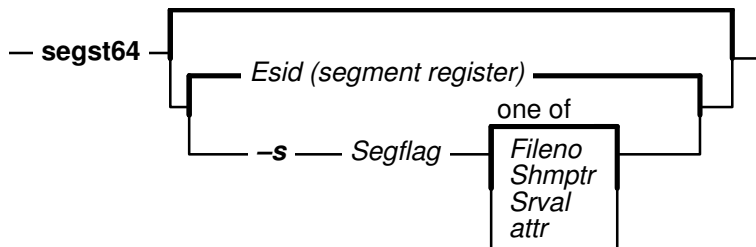
1. To display the next 112 bytes of data, enter:
screen +
2. To display the previous 112 bytes of data, enter:
screen -
3. To display memory starting at 20000FF7, enter:
s 20000ff7
4. To display memory at the address contained in location 200, enter:
s 200>
5. To turn on the display, enter:
screen on
6. To turn off the display, enter:
screen off
7. To set the display format to use about half of the screen, enter:
screen on half
8. To track memory starting at the value in general purpose register 3, enter:
sc track r3

segst64 Command for the LLDB Kernel Debug Program

Purpose

Displays the states of all memory segments of a 64-bit process.

Syntax



Description

The **segst64** command displays the states of all the segments starting from the specified Esid (segment register). You can also specify Segflag parameter, with a string to identify the type of the segment (SEG_AVAIL_, SEG_MAPPED, etc.) and optionally either fileno or pointer to shared memory segment or srval, or segment attribute (attr) to uniquely locate the segment you are looking for. You will be prompted to specify ProcessID (pid) of a 64-bit process. You will also be prompted to specify starting Esid (segment register) if you do not specify it as a parameter. The currently active 64-bit process's ProcessID (pid) with starting register value of 3 would be the default. If no parameter was specified, the **segst64** command displays states of all the segments of the currently active 64-bit process starting from segment register value 3.

Examples

1. To display the states of all the segments of the currently active 64-bit process starting from Esid (segment register) value 3.
segst64

2. You will be prompted to specify the ProcessID (pid) and Esid (segment register). Press the Enter key at the prompt if you want to accept the default values.
3. To display states of all the segments in SEG_AVAIL, state, of the currently active 64-bit process starting from Esid (segment register) value 3, enter:

```
seget64 -s "SEG_AVAIL"
```

You will be prompted to specify the ProcessID (pid) and Esid (segment register). Press the Enter key at the prompt if you want to accept the default values.

4. To display states of all the segments in SEG_MAPPED state with fileno value of 1, of the currently active 64-bit process starting from Esid (segment register) value 3, enter:

```
seg -s "SEG_MAPPED" 1
```

You will be prompted to specify the ProcessID (pid) and Esid (segment register). Press the Enter key at the prompt if you want to accept the default values.

5. To display the states of all the segments of the currently active 64-bit process starting from Esid (segment register) value 257, enter:

```
segst64 257
```

You will be prompted to specify the ProcessID (pid). Press the Enter key at the prompt if you want to accept the default value.

set Command for the LLDB Kernel Debug Program

Purpose

Create and change values of debugger variables.

Syntax

```
— set — [ Variable ] — Value —
           [ Register ]
```

Description

This command sets debugger variables. Use the **set** command to create new variables or modify the value of old variables. Certain debugger variables are symbolic names for machine registers, which you can modify. See “Reserved Variables” on page 336 for a list of these variables.

An additional debugger variable **asr** has been introduced in AIX4.3. Also, in 64-bit context, you can set segment register values to all the possible segment registers ranging from 0 to FFFFFFFF, using a debugger variable **sx<nnnnnnnn>** (i.e., **sx** followed by segment register number). In 64-bit context, the segment registers are emulated in memory. An error message will be displayed if the assignment to the memory location is paged out.

The **sr64** subcommand could be used to verify the **srval** just set to the **sx<nnnnnnnn>register**.

Note: The reset command cannot be used to release the `sxxxxnnnnn` debugger variable.

Examples

1. To assign value 100 to variable `start`.
`set start 100`
2. To set general purpose register 12 to 0.
`set r12 0`
3. To set segment register 3 to 10000.
`se s3 10000`
4. To assign 45F0 to the `Iar`.
`set iar 45F0`
5. To assign string "AIX" to variable `name`.
`se name "AIX"`
6. To set segment register 257 to 10000
`set sx257 10000`

sregs Command for the LLDB Kernel Debug Program

Purpose

Displays segment registers all times except in 64-bit context.

Syntax

— **sregs** —|

Description

The **sregs** command displays the contents of the segment registers and other control registers. The display created is similar to that created by the **screen** command (see "screen Command for the LLDB Kernel Debug Program" on page 359).

The screen display format changes in 64-bit context. If the debugger is in 64-bit context, then the segment registers will not be displayed. All GPRs and other control registers with 64-bits wide contents will be displayed.

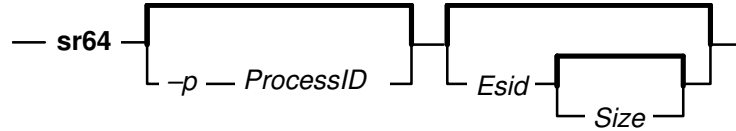
Note: `sr64` command must be used to look at the contents of the segment registers for a 64-bit process.

sr64 Command for the LLDB Kernel Debug Program

Purpose

Displays segment registers only in 64-bit context.

Syntax



Description

The `sr64` command displays all the segment register values of a 64-bit process specified by a ProcessID (pid) parameter starting from specified Esid (segment register) parameter. The default Pid value process would be that of the currently active 64-bit process and the default Esid value would be zero. An error message will be displayed if either the specified process or the default currently active process is not a 64-bit process.

Examples

1. To display the segment registers of currently active 64-bit process, starting from esid (segment register number) 257, enter:
`sr64`
2. To display the segment registers of a 64-bit process of pid 204, starting from Esid (segment register number) zero, enter:
`sr64 -p 204`
3. To display the segment registers of a 64-bit process of pid 204, starting from Esid (segment register number) 257, enter:
`sr64 -p 204 257`

st Command for the LLDB Kernel Debug Program

Purpose

Stores a fullword into memory.

Syntax

```
— st — Address — Data —
```

Description

The `st` command stores a fullword of data into memory by using the processor fullword store instruction. If the address specified by the *Address* parameter is not word-aligned, it is rounded down to a fullword. The `st` command is the correct way to place a fullword of data into input and output memory.

This is similar to the `alter` command, but the word size is implicit in the command. `stc` and `sth` are used to perform similar functions for bytes and halfwords.

Example

To store the 32-bit value 5 at address 1000, enter:

```
st 1000 5
```

stack Command for the LLDB Kernel Debug Program

Purpose

Displays a formatted stack traceback.

Syntax

— **stack** — ThreadID —|

Description

The **stack** command displays a formatted kernel-stack traceback for the specified kernel thread. If no thread is specified, the currently running thread is used. Stack frames show return addresses and can be used to trace the calling sequence of the program. Be aware that the first few parameters are passed in registers to the called functions, and are not usually available on the stack. Generally only the stack chain (stacks back-chain pointer) and return address (address where the current function returns upon completion) are valid. To interpret the stack thoroughly, it is necessary to use an assembler language listing for a procedure to determine what has been stored on the stack. Stack frames for the specified thread are not always accessible.

Examples

1. To format any existing stack frames, enter:
stack
2. To format stack frames for the thread with thread ID 251 enter:
sta 251

stc Command for the LLDB Kernel Debug Program

Purpose

Stores one byte into memory.

Syntax

— **stc** — *Address* — *Data* —|

Description

The **stc** command stores a byte of data specified by the *Data* parameter into memory at the address specified by the *Address* parameter by using the processor store-character instruction. The **stc** command is the correct way to place a byte of data into input and output memory.

This is similar to the **st** and **sth** commands, which are used for fullwords and halfwords.

Example

To store the 8-bit value FF at address 1000, enter:

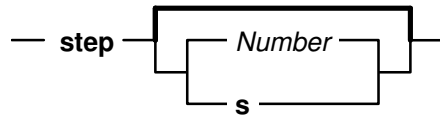
```
stc 1000 ff
```

step Command for the LLDB Kernel Debug Program

Purpose

Runs instructions single-step.

Syntax



Description

The **step** command causes the processor to enter a single instruction and return control to the debug program. If a branch is the next instruction to be run, the **s** flag causes the processor to step over a subroutine call. An integer *Number* parameter is used as the number of instructions to run before returning control to the debug program.

Note: On multiprocessor systems, other processors are not released during **step**, contrary to most commands.

Flag

s Executes a subroutine as if it were one instruction.

Examples

1. To single step the processor, enter:
step
2. To single step and skip over a subroutine call, enter:
step s
3. To step for 20 instructions, enter:
step 20

sth Command for the LLDB Kernel Debug Program

Purpose

Stores a halfword into memory.

Syntax



Description

The **sth** command stores a halfword of data specified by the *Data* parameter into memory by using the processor store halfword instruction. If the address specified by the *Address* parameter is not halfword-aligned, it is rounded down to a halfword boundary. The **sth** command is the correct way to place a halfword into input and output memory space.

This is similar to the **st** and **stc** commands, which are used for fullwords and bytes.

Example

To store the 16-bit value 14 at address 1000, enter:

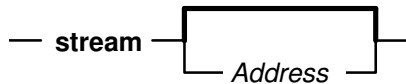
```
sth 1000 0014
```

stream Command for the LLDB Kernel Debug Program

Purpose

Displays the contents of the stream head table.

Syntax



Description

The stream command displays the contents of the stream head table. If no address is specified, the command displays the first stream found in the STREAMS hash table. If the address is specified, the command displays the contents of the stream head stored at that address.

The information printed is contained in an internal structure. The following members of this internal structure are described here:

<i>sth</i>	address of stream head
<i>wq</i>	address of streams write queue
<i>rq</i>	address of streams read queue
<i>dev</i>	associated device number of the stream
<i>read_mode</i>	read mode
<i>write_mode</i>	write mode
<i>close_wait_timeout</i>	close wait timeout in microseconds
<i>read_error</i>	read error on the stream
<i>write_error</i>	write error on the stream
<i>flags</i>	stream head flag values
<i>push_cnt</i>	number of modules pushed on the stream
<i>wroff</i>	write offset to prepend M_DATA
<i>ioc_id</i>	id of outstanding M_IOCTL request
<i>ioc_mp</i>	outstanding ioctl message
<i>next</i>	next stream head on the link
<i>pollq</i>	list of active polls
<i>sigsq</i>	list of active M_SETSIGs
<i>shtty</i>	pointer to tty information

The *read_mode* and *write_mode* values are defined in the `/usr/include/sys/stropts.h` header file.

The *read_error* and *write_error* variables are integers defined in the `/usr/include/sys/errno.h` header file.

The *flags* structure member, if set, is based on combinations of the following values:

#define	Value	Description
F_STH_READ_ERROR	0x0001	M_ERROR with read error received, fail all read calls.
F_STH_WRITE_ERROR	0x0002	M_ERROR with write error received, fail all writes.
F_STH_HANGUP	0x0004	M_HANGUP received, no more data.
F_STH_NDELOK	0x0008	Do TTY semantics for ONDELAY handling.
F_STH_ISATTY	0x0010	This stream acts a terminal.
F_STH_MREADON	0x0020	Generate M_READ messages.
F_STH_TOSTOP	0x0040	Disallow background writes (for job control).
F_STH_PIPE	0x0080	Stream is one end of a pipe or FIFO.
F_STH_WPIPE	0x0100	Stream is the "write" side of a pipe.
F_STH_FIFO	0x0200	Stream is a FIFO.
F_STH_LINKED	0x0400	Stream has one or more lower streams linked.
F_STH_CTTY	0x0800	Stream controlling tty.
F_STH_CLOSED	0x4000	Stream has been closed, and should be freed.
F_STH_CLOSING	0x8000	Actively on the way down.

In the output, values marked with X are printed in hexadecimal format.

This command can also be invoked via the alias, `str`.

Examples

1. To display the first stream head found in the stream head table, enter:
`stream`
2. To display the contents of the particular stream head located at address 59b2e00 (where 59b2e00 is a valid stream head address), enter:
`stream 59b2e00`

swap Command for the LLDB Kernel Debug Program

Purpose

Switches to the specified RS-232 port.

Syntax

— **swap** — *Port* —|

Description

The **swap** command allows control of the debug program to be transferred to another terminal. The *Port* parameter specifies which asynchronous tty port to transfer control. The **swap** command does not support returning to a port that was previously used.

Specify 0 for port 0 (s1) or 1 for port 1 (s2).

Ports must be configured the same as the port on which the debug program is currently running: 9600 baud, 8 data bits, no parity. The device attached to the port must respond with a carrier detect within 1/10 seconds or the command fails and control will not be transferred.

Example

To switch display to RS-232 port 1, enter:

```
swap 1
```

sysinfo Command for the LLDB Kernel Debug Program

Purpose

Displays the system configuration information.

Syntax

— **sysinfo** —|

Description

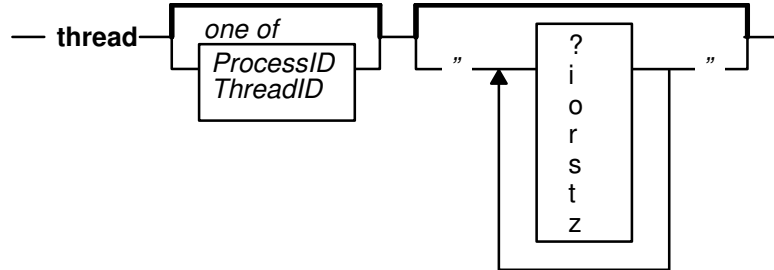
The **sysinfo** command displays the system configuration information such as the model, architecture and more details.

thread Command for the LLDB Kernel Debug Program

Purpose

Displays thread table entries.

Syntax



Description

The **thread** command displays the contents of the kernel thread table. If the *ProcessID* (pid) parameter is given, information about all kernel threads belonging to that process is displayed. If the *ThreadID* parameter is given, detailed information about the specified kernel thread is displayed. If no parameters are given, information about all kernel threads in the kernel thread table is displayed. Note that the *ProcessID* (pid) and *ThreadID* parameters share a common name space: even numbers are always used for ProcessIDs, whereas odd numbers are used for threads (the init processes, PID 1, is an exception).

If you specify a string of flags of desired thread states, then only the list of threads that match the desired threads states will be displayed.

Flags

List of process states indicated by flags:

i	idle
o	swap
r	runnable
s	sleep
t	stop
z	zombie

Examples

1. To display information about all threads in the thread table, enter:

```
thread
```

The output is similar to:

```
SLT ST TID  PID CPUID  POLICY PRI CPU  EVENT
PROCNAME  FLAGS
0 s   3    0   ANY  OTHER  10  78          swapper 0x00001400
1 s  103   1   ANY  OTHER  3C   0          init    0x00000400
2*r  205  204   0   OTHER  7F  78          wait    0x00001000
3 r   307  306   1   OTHER  7F  78          wait    0x00001000
4 s   409  408  ANY  OTHER  24   0          netm    0x00001000
5 s   50B  50A  ANY  OTHER  24   0          gil     0x00001000
6 s  60D  50A  ANY  OTHER  24   0 000B2DA8 gil     0x00001000
7 s   70F  50A  ANY  OTHER  24   0 000B2DA8 gil     0x00001000
8 s   811  50A  ANY  OTHER  24   0 000B2DA8 gil     0x00001000
9 s   913  50A  ANY  OTHER  24   1 000B2DA8 gil     0x00001000
10 s  A15  60C  ANY  OTHER  3C   0          sh      0x00000400
11 s  B17  70E  ANY  OTHER  3C   0          sh      0x00000400
```

2. To display information about the threads in process 2106, enter:

th 2106

3. To display information about the thread with thread ID 1497, enter:

th 1497

4. To display only the entries of sleeping and zombie threads, enter:

th "sz"

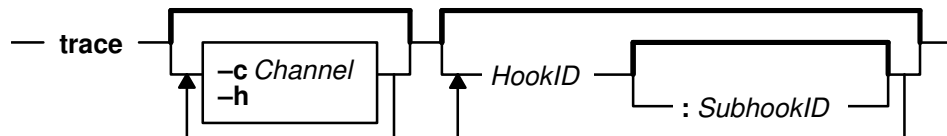
Note: All the flags must be entered as a single string.

trace Command for the LLDB Kernel Debug Program

Purpose

Displays formatted kernel trace buffers.

Syntax



Description

The **trace** command displays the last 128 entries of a kernel trace buffer in reverse chronological order. There are 8 trace buffers, each associated with a trace channel. Each can trace any combination of trace events. Trace data gives an indication of system activity at a very low level; interrupts, input/output, and process scheduling are examples of event types that can be traced.

The **trace** command displays headers for the trace buffers that contain pointers into the trace buffers and the state of the trace driver. Following this are the last 128 entries from the selected trace buffer. Trace entries consist of a major and a minor number for the trace hook, an ASCII trace ID, an ASCII trace hook type, followed by either a hexadecimal dump of the trace data or a pointer to the start of a variable-length block of trace data.

The **trace** command is not meant to replace the **trcfmt** command, which formats the trace data in more detail. It is a facility for viewing system trace data in the event of a system crash before the data has been written to disk.

Flags

-c Channel Specifies the trace channel used.
-h Displays the trace headers.

Examples

1. To display a sequence of trace entries, enter:

```
trace
```

The system then returns the following question:

```
Display channel (0 - 8): 0
```

2. To display a sequence of trace entries with hookword 105, enter:

```
trace 105 -c 0
```

3. To display a sequence of trace entries with hookword 105 and subhook d, enter:

```
trace 105:d -c 0
```

4. To display all entries with hookword 105 or 10b, enter:

```
trace 105 10b
```

5. To display all entries with hookword 105 and a 300 in the trace data, enter:

```
trace 105 #300
```

6. To display the trace headers, enter:

```
trace -h
```

trb Command for the LLDB Kernel Debug Program

Purpose

Displays the timer request blocks (TRBs).

Syntax

```
— trb —|
```

Description

The **trb** command displays a menu of commands to display timer request block (TRB) information.

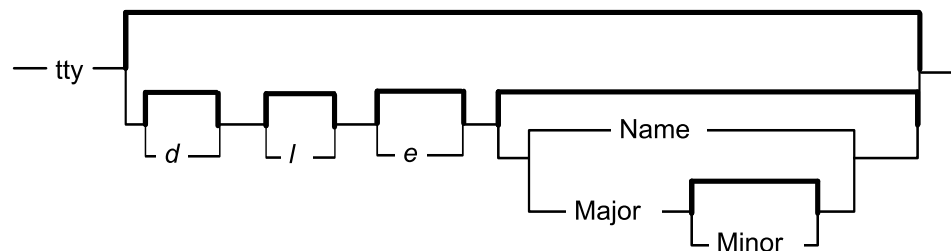
The **trb** command allows you to traverse the active and free TRB chains; examine TRBs by process, slot number, or address; and examine the clock interrupt handler information.

tty Command for the LLDB Kernel Debug Program

Purpose

Displays the tty structure.

Syntax



Description

The **tty** command displays tty data structures. If no parameters are specified, a verbose listing of all terminals is displayed. Short forms of the listings can be requested showing all terminals or all currently open terminals. If no parameters are specified, a short listing of all opened terminals is displayed. Selected terminals can be displayed by specifying the terminal name in the *Name* parameter, such as **tty1**, or a major device number with optional minor and channel numbers. If the *Major* parameter is specified, all terminals with the specified major number are listed. If the *Major* and *Minor* parameters are both specified, all the terminals with both the specified major and minor numbers are listed.

Selected type of information can be displayed, according to the specified flags.

Flags

a	Displays a short listing of all terminals.
o	Displays a short listing of all open terminals.
v	Displays a verbose listing.
d	Displays the driver information.
l	Displays the line discipline information.
e	Displays information for every module and driver present in the stream for the selected lines.

Examples

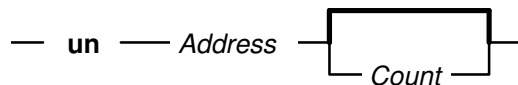
1. To display listings for each open terminal, enter:
`tty`
2. To display the driver and line discipline information for terminal **tty1**, enter:
`tty d l tty1`
3. To display the listing for the terminal with a major number 7 and a minor number 1, enter:
`tty 7 1`

un Command for the LLDB Kernel Debug Program

Purpose

Displays the assembly instruction(s).

Syntax



Description

The `un` command disassembles the contents starting at the address specified by the `Addr` parameter and displays the assembly instructions. The `Size` parameter indicates the number of instructions to be displayed and has a default value of 1.

Examples

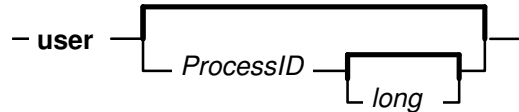
1. To disassemble and display the instruction at the address 1000, enter:
`un 1000`
2. To disassemble and display 5 instructions starting at 1000, enter:
`un 1000 5`

user Command for the LLDB Kernel Debug Program

Purpose

Displays the U-area (user area).

Syntax



Description

The **user** command with * parameter, displays the U-area for the current process. With a long flag specified, the **user** command displays more details of the U-area displayed. If the U-area is being displayed for a 64-bit process, a message will be displayed to indicate so.

Examples

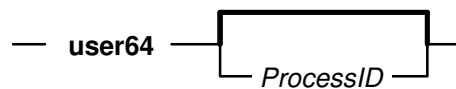
1. To display the current U-area, enter:
user
2. To display the U-area for the process with ProcessID 314, enter:
u 314
3. To display U-area of currently active process, enter:
u *
4. To display U-area of a process with ProcessID (pid) 204, giving more details, enter:
u 204 long

user64 Command for the LLDB Kernel Debug Program

Purpose

Displays the user64 structure of a 64-bit process.

Syntax



Description

The **user64** command displays the **user64** structure if you specify the processID (pid) of a 64-bit process. With no parameter specified, the **user64** structure of the currently active 64-bit process is displayed.

Examples

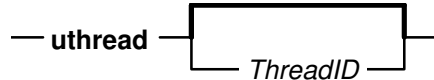
1. To display user64 structure of a 64-bit process with processID (pid) 204, enter:
user64 204

uthread Command for the LLDB Kernel Debug Program

Purpose

Displays the **uthread** structure.

Syntax



Description

The **uthread** command displays **uthread** structures. If the *ThreadID* parameter is given, the **uthread** structure of the specified kernel thread is displayed. Otherwise, the **uthread** structure of the current kernel thread is displayed.

If the **uthread** is being displayed for thread of a 64-bit process, a message will be displayed to indicate so. In 64-bit context, the segment registers will not be displayed and GPRs contents displayed will be 64 bits wide.

Examples

1. To display the **uthread** structure of the current kernel thread, enter:

```
uthread
```

The output is similar to:

using current thread:

UTHRAD AREA FOR TID 0x00000205

SAVED MACHINE STATE

curid:0x00000204 m/q:0x00000000 iar:0x000214D4 cr:0x24000000

msr:0x00009030 lr:0x00021504 ctr:0x0002147C xer:0x20000000

*prevmst:0x00000000 *stackfix:0x00000000 intpri:0x0000000B

backtrace:0x00 tid:0x00000000 fpeu:0x00 ecr:0x00000000

Exception Struct

0x00000000 0x00000000 0x00000000 0x00000000 0x00000000

Segment Regs

0:0x00000000 1:0x007FFFFFFF 2:0x00000408 3:0x007FFFFFFF

4:0x007FFFFFFF 5:0x007FFFFFFF 6:0x007FFFFFFF 7:0x007FFFFFFF

8:0x007FFFFFFF 9:0x007FFFFFFF 10:0x007FFFFFFF 11:0x007FFFFFFF

12:0x007FFFFFFF 13:0x007FFFFFFF 14:0x00000204 15:0x007FFFFFFF

General Purpose Regs

0:0x00000000 1:0x2FEAEF38 2:0x00270314 3:0x00000054

4:0x00000002 5:0x00000000 6:0x000BF9B8 7:0x00000000

8:0xDEADBEEF 9:0xDEADBEEF 10:0xDEADBEEF 11:0x00000000

12:0x00009030 13:0xDEADBEEF 14:0xDEADBEEF 15:0xDEADBEEF

16:0xDEADBEEF 17:0xDEADBEEF 18:0xDEADBEEF 19:0xDEADBEEF

20:0xDEADBEEF 21:0xDEADBEEF 22:0xDEADBEEF 23:0xDEADBEEF

24:0xDEADBEEF 25:0xDEADBEEF 26:0xDEADBEEF 27:0xDEADBEEF

28:0xDEADBEEF 29:0xDEADBEEF 30:0xDEADBEEF 31:0xDEADBEEF

Press "ENTER" to continue, or "x" to exit:>0>

Floating Point Regs

Fpscr: 0x00000000

0:0x00000000 0x00000000 1:0x00000000 0x00000000 2:0x00000000 0x00000000

3:0x00000000 0x00000000 4:0x00000000 0x00000000 5:0x00000000 0x00000000

6:0x00000000 0x00000000 7:0x00000000 0x00000000 8:0x00000000 0x00000000

9:0x00000000 0x00000000 10:0x00000000 0x00000000 11:0x00000000 0x00000000

12:0x00000000 0x00000000 13:0x00000000 0x00000000 14:0x00000000 0x00000000

15:0x00000000 0x00000000 16:0x00000000 0x00000000 17:0x00000000 0x00000000

18:0x00000000 0x00000000 19:0x00000000 0x00000000 20:0x00000000 0x00000000

21:0x00000000 0x00000000 22:0x00000000 0x00000000 23:0x00000000 0x00000000

24:0x00000000 0x00000000 25:0x00000000 0x00000000 26:0x00000000 0x00000000

27:0x00000000 0x00000000 28:0x00000000 0x00000000 29:0x00000000 0x00000000

30:0x00000000 0x00000000 31:0x00000000 0x00000000

Kernel stack address: 0x2FEAEFFC

Press "ENTER" to continue, or "x" to exit:>0>

SYSTEM CALL STATE


```

7 user stack:0x00000000 user msr:0x00000000
  errno address:0xC0C0FADE error code:0x00 *kjmpbuf:0x00000000
  ut_flags:
PER-THREAD TIMER MANAGEMENT
  Real/Alarm Timer (ut_timer.t_trb[TIMERID_ALARM]) = 0x0
  Virtual Timer (ut_timer.t_trb[TIMERID_VIRTUAL]) = 0x0
  Prof Timer (ut_timer.t_trb[TIMERID_PROF]) = 0x0
  Posix Timer (ut_timer.t_trb[POSIX4]) = 0x0
SIGNAL MANAGEMENT
  *sigsp:0x0 oldmask:hi 0x0,lo 0x0 code:0x0
Press "ENTER" to continue, or "x" to exit:>0>
Miscellaneous fields:
  fstid:0x00000000 ioctlrv:0x00000000 selchn:0x00000000
Uthread area printout terminated.

```

- To display the `uthread` structure of the kernel thread with thread ID 1497, enter:

```
ut 1497
```

vars Command for the LLDB Kernel Debug Program

Purpose

Displays a list of user-defined variables.

Syntax

```
— vars —|
```

Description

The `vars` command displays the user-defined variables and their values.

The command displays the variable name and value, and an indication of what is the base of the value. Since the value 10 can be either decimal or hexadecimal it is displayed as HEX/DEC. The command displays string variables with no quotes around the string value.

The values of the reserved variables `fx` and `org` are also displayed.

vmm Command for the LLDB Kernel Debug Program

Purpose

Displays the virtual memory information menu.

Syntax

```
— vmm —|
```

Description

The `vmm` command displays a menu of commands for displaying the virtual memory data structures. These commands examine segment register values for

kernel segments such as the ram disk and the page space disk maps. Addresses and sizes of VMM data structures are also available, as are VMM statistics such as the number of page faults and the number of pages paged in or out.

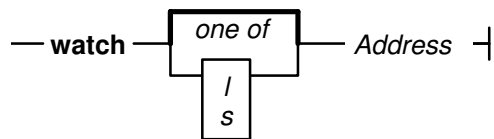
The stab contents could be displayed using one of `vmm` menu commands, for a 64-bit process.

watch Command for the LLDB Kernel Debug Program

Purpose

Watches for load and/or store at an address.

Syntax



Description

The `watch` command allows you to enter the debugger if and when there is a load and/or store at an address that you specify. The optional flag `l` or load indicates that load is to be detected, `s` or store indicates that store is to be detected. With no flag specified, either load or store will be detected by the debugger. Since the `watch` command is only available on some hardware, check the hardware technical reference information to see if this is available on your system.

Examples

1. To enter the debugger once the address 1000 is loaded (read):
`watch l 1000`
2. To enter the debugger once the address 1000 is either loaded (read) or stored (written) with some value:
`watch 1000`

xlate Command for the LLDB Kernel Debug Program

Purpose

Translates a virtual address to a real address.

Syntax



Description

The `xlate` command displays the real address corresponding to the specified virtual address.

Example

To display the real address corresponding to the virtual address 10054000, enter:

```
xlate 10054000
10054000 -virtual- 00000000_000EF004 -real-
```

00000000_000EF004 is the corresponding real address. The real address is displayed 64 bits wide, since AIX 4.3 supports real memory greater than 4GB on 64-bit systems.

Maps and Listings as Tools for the LLDB Kernel Debug Program

The assembler listing and the map files are essential tools for debugging using the LLDB Kernel Debugger. In order to create the assembler list file during compilation, use the **-qlist** option while compiling. Also use the **-qsource** option to get the C source listing in the same file:

```
cc -c -DEBUG -D_KERNEL -DIBMR2 demodd.c -qsource -qlist
```

In order to obtain the map file, use the **-bmap:FileName** option on the link editor, enter:

```
ld -o demodd demodd.o -edemoconfig -bimport:/lib/kernex.exp \
-lsys -lcsys -bmap:demodd.map -bE:demodd.exp
```

You can also create a map file with a slightly different format by using the **nm** command. For example, use the following command to get a map listing for the kernel (**/unix**):

```
nm -xv /unix > unix.m
```

Compiler Listing

The assembler and source listing is used to correlate any C source line with the corresponding assembler lines. The following is a portion of the C source code for a sample device driver. The left column is the line number in the source code:

```
.
.
185
186   if (result = devswadd(devno, &demo_dsw)){
187       printf("democonfig : failed to add entry points\n");
188       (void)devswdel(devno);
189       break;
190   }
191   dp->initd = 1;
192   demos_initd++;
193   printf("democonfig : CFG_INIT success\n");
194   break;
195
.
.
```

The following is a portion of the assembler listing for the corresponding C code shown previously. The left column is the C source line for the corresponding assembler statement. Each C source line can have multiple assembler source lines. The second column is the offset of the assembler instruction with respect to the kernel extension entry point.

```
.
.
186| 000218 1      80610098 2  L4Z  gr3=devno(gr1,152)
186| 00021C ca1    389F0000 1  LR   gr4=gr31
186| 000220 b1     4BFFFDE1 0  CALL gr3=devswadd,2,
gr3,(struct_4198576)",gr4,devswadd",gr1,cr[01567],gr0",
```

```

gr4"-gr12",fp0"-fp13"
186| 000224 cror 4DEF7B82 1
186| 000228 st 9061005C 2 ST4A #2357(gr1,92)=gr3
186| 00022C st 9061003C 1 ST4A result(gr1,60)=gr3
186| 000230 l 8061005C 1 L4A gr3=#2357(gr1,92)
186| 000234 cmpi 2C830000 2 C4 cr1=gr3,0
186| 000238 bc 41860020 3 BT CL.16,cr1,0x4/eq
187| 00023C ai 307F01A4 1 AI gr3=gr31,420
187| 000240 bl 4BFFFDC1 2 CALL gr3=printf,1,'democonfig :
failed to add entry points",gr3,printf",gr1,cr[01567],gr0",
gr4"-gr12",fp0"-fp13"
187| 000244 cror 4DEF7B82 1
188| 000248 l 80610098 2 L4Z gr3=devno(gr1,152)
188| 00024C bl 4BFFFDB5 0 CALL gr3=devswdel,1,gr3,
devswdel",gr1,cr[01567],gr0",gr4"-gr12",fp0"-fp13"
188| 000250 cror 4DEF7B82 1
189| 000254 b 48000104 0 B CL.6
186| CL.16:
191| 000258 l 80810040 2 L4Z gr4=dp(gr1,64)
191| 00025C cal 38600001 1 LI gr3=1
191| 000260 stb 98640004 1 ST1Z (char)(gr4,4)=gr3
192| 000264 l 8082000C 1 L4A gr4=.demos_inited(gr2,0)
192| 000268 l 80640000 2 L4A gr3=demos_inited(gr4,0)
192| 00026C ai 30630001 2 AI gr3=gr3,1
192| 000270 st 90640000 1 ST4A demos_inited(gr4,0)=gr3
193| 000274 ai 307F01D0 1 AI gr3=gr31,464
193| 000278 bl 4BFFFDB9 0 CALL gr3=printf,1,'democonfig :
CFG_INIT success",gr3,printf",gr1,cr[01567],gr0",gr4"-gr12",
fp0"-fp13"
193| 00027C cror 4DEF7B82 1
194| 000280 b 480000D8 0 B CL.6
.
.

```

Now with both the assembler listing and the C source listing, you can determine the assembler instruction for a C statement. As an example, consider the C source line at line 191 in the sample code:

```
191 dp->inited = 1;
```

The corresponding assembler instructions are:

```

191| 000258 l 80810040 2 L4Z gr4=dp(gr1,64)
191| 00025C cal 38600001 1 LI gr3=1
191| 000260 stb 98640004 1 ST1Z (char)(gr4,4)=gr3

```

The offsets of these instructions within the sample device driver (demodd) are 000258, 00025C, and 000260.

Map File

The binder map file is a symbol map in address order format. Each symbol listed in the map file has a storage class (CL) and a type (TY) associated with it.

Storage classes correspond to the **XMC_XX** variables defined in the **syms.h** file. Each storage class belongs to one of the following section types:

.text	Contains read-only data (instructions). Addresses listed in this section use the beginning of the .text section as origin. The .text section can contain one of the following storage class (CL) values:
DB	Debug Table. Identifies a class of sections that has the same characteristics as read only data.
GL	Glue Code. Identifies a section that has the same characteristics as a program code. This type of section has code to interface with a routine in another module. Part of the interface code requirement is to maintain TOC addressability across the call.
PR	Program Code. Identifies the sections that provide executable instructions for the module.
R0	Read Only Data. Identifies the sections that contain constants that are not modified during execution.
TB	Reserved.
TI	Reserved.
XO	Extended Op. Identifies a section of code that is to be treated as a pseudo-machine instruction.
.data	Contains read-write initialized data. Addresses listed in this section use the beginning of the .data section as origin. The .data section can contain one of the following storage class (CL) values:
DS	Descriptor. Identifies a function descriptor. This information is used to describe function pointers in languages such as C and Fortran.
RW	Read Write Data. Identifies a section that contains data that is known to require change during execution.
SV	SVC. Identifies a section of code that is to be treated as a supervisory call.
T0	TOC Anchor. Used only by the predefined TOC symbol. Identifies the special symbol TOC. Used only by the TOC header.
TC	TOC Entry. Identifies address data that will reside in the TOC.
TD	TOC Data Entry. Identifies data that will reside in the TOC.
UA	Unclassified. Identifies data that contains data of an unknown storage class.
.bss	Contains read-write uninitialized data. Addresses listed in this section use the beginning of the .data section as origin. The .bss section contain one of the following storage class (CL) values:
BS	BSS class. Identifies a section that contains uninitialized data.
UC	Unnamed Fortran Common. Identifies a section that contains read write data.

Types correspond to the **XTY_XX** variables defined in the **syms.h** file. The type (TY) can be one of the following values:

ER	External Reference
LD	Label Definition
SD	Section Definition

The following is a map file for a sample device driver:

```

1 ADDRESS MAP FOR demodd
2
3 *IE ADDRESS LENGTH AL CL TY Sym# NAME SOURCE-FILE(OBJECT) or
4 I 00000000 0008B8 2 PR SD S9 <> IMPORT-FILE{SHARED-OBJECT}
5 I 00000000 PR LD S10 .democonfig
6 I 0000039C PR LD S11 .demoopen
7 I 000004B4 PR LD S12 .democlose
8 I 000005D4 PR LD S13 .demoread
9 I 00000704 PR LD S14 .demowrite
10 I 00000830 PR LD S15 .get_dp
11 I 000008B8 000024 2 GL SD S16 <.printf> glink.s(/usr/lib/glink.o)
12 I 000008B8 GL LD S17 .printf
13 I 000008DC 000024 2 GL SD S18 <.xmalloc> glink.s(/usr/lib/glink.o)
14 I 000008DC GL LD S19 .xmalloc
15 I 00000900 000090 2 PR SD S20 .bzero
16 noname(/usr/lib/libcsys.a[bzero.o])
17 I 00000990 000024 2 GL SD S21 <.uiomove> glink.s(/usr/lib/glink.o)
18 I 00000990 GL LD S22 .uiomove
19 I 000009B4 000024 2 GL SD S23 <.devswadd> glink.s(/usr/lib/glink.o)
20 I 000009B4 GL LD S24 .devswadd
21 I 000009D8 000024 2 GL SD S25 <.devswdel> glink.s(/usr/lib/glink.o)
22 I 000009D8 GL LD S26 .devswdel
23 I 000009FC 000024 2 GL SD S27 <.xmfree> glink.s(/usr/lib/glink.o)
24 I 000009FC GL LD S28 .xmfree
25 I 00000000 000444 4 RW SD S29 <_/tmp/cliff/demodd/demodd$c$>
26 /tmp/cliff/demodd/demodd.c(demodd.o)
27 I 00000450 000004 4 RW SD S30 demo_dev
28 /tmp/cliff/demodd/demodd.c(demodd.o)
29 I 00000460 000004 4 RW SD S31 demos_inited
30 /tmp/cliff/demodd/demodd.c(demodd.o)
31 I 00000470 000080 4 RW SD S32 data
32 /tmp/cliff/demodd/demodd.c(demodd.o)
33 * E 000004F0 00000C 2 DS SD S33 democonfig
34 /tmp/cliff/demodd/demodd.c(demodd.o)
35 E 000004FC 00000C 2 DS SD S34 demoopen
36 /tmp/cliff/demodd/demodd.c(demodd.o)
37 E 00000508 00000C 2 DS SD S35 democlose
38 /tmp/cliff/demodd/demodd.c(demodd.o)
39 E 00000514 00000C 2 DS SD S36 demoread
40 /tmp/cliff/demodd/demodd.c(demodd.o)
41 E 00000520 00000C 2 DS SD S37 demowrite
42 /tmp/cliff/demodd/demodd.c(demodd.o)
43 I 0000052C 000000 2 T0 SD S38 <TOC>
44 I 0000052C 000004 2 TC SD S39 <_/tmp/cliff/demodd/demodd$c$>
45 I 00000530 000004 2 TC SD S40 <printf>
46 I 00000534 000004 2 TC SD S41 <demo_dev>
47 I 00000538 000004 2 TC SD S42 <demos_inited>
48 I 0000053C 000004 2 TC SD S43 <data>
49 I 00000540 000004 2 TC SD S44 <pinned_heap>
50 I 00000544 000004 2 TC SD S45 <xmalloc>
51 I 00000548 000004 2 TC SD S46 <uiomove>

```

```

50      0000054C 000004 2 TC SD S47 <devswadd>
51      00000550 000004 2 TC SD S48 <devswdel>
52      00000554 000004 2 TC SD S49 <xmfree>

```

In the sample map file listed previously, the **.data** section starts from the statement at line 32:

```

32      00000000 000444 4 RW SD S29 <_/tmp/cliff/demodd/demodd$c$>
/tmp/cliff/demodd/demodd.c(demodd.o)

```

The TOC (Table of Contents) starts from the statement at line 41:

```

41      0000052C 000000 2 T0 SD S38 <TOC>

```

Using the LLDB Kernel Debug Program

This section contains information on:

- “Setting Breakpoints”
- “Viewing and Modifying Global Data” on page 384
- “Displaying Registers on a Micro Channel Adapter” on page 386
- “Stack Trace” on page 387

Setting Breakpoints

Setting a breakpoint is essential for debugging kernel or kernel extensions. To set a breakpoint, use the following sequence of steps:

1. Locate the assembler instruction corresponding to the C statement.
2. Get the offset of the assembler instruction from the listing.
3. Locate the address where the kernel extension is loaded.
4. Add the address of the assembler instruction to the address where kernel extension is loaded.
5. Set the breakpoint with the **break** command.

The process of locating the assembler instruction and getting its offset is explained in the previous section. The next step is to get the address where the kernel extension is loaded.

Determine the Location of your Kernel Extension

To determine the address where a kernel extension has been loaded, use the following procedure. First, find the load point (the entry point) of the executable kernel extension. This is a label supplied with the **-e** option for the **ld** (links objects) command used while generating the kernel extension. In our example this is the **democonfig** routine.

Then use one of the following six methods to locate the address of this load point. This address is the location where the kernel extension is loaded.

Method 1

If the kernel extension is a device driver, use the **drivers** command to locate the address of the load point routine. The **drivers** command lists all the function descriptors and the function addresses for the device driver (that are in the dev switch table). Usually the **config** routine will be the load point routine. Hence in our example the function address for the **config** (**democonfig**) routine is the address where the kernel extension is loaded.

> drivers 255				
MAJ#255	Open	Close	Read	Write

func	desc	0x01B131B0	0x01B131BC	0x01B131C8	0x01B131D4
func	addr	0x01B12578	0x01B126A0	0x01B127D4	0x01B12910
		Ioctl	Strategy	Tty	Select
func	desc	0x00019F10	0x00019F10	0x00000000	0x00019F10
func	addr	0x00019A20	0x00019A20		0x00019A20
		Config	Print	Dump	Mpx
func	desc	0x01B131A4	0x00019F10	0x00019F10	0x00019F10
func	addr	0x01B121EC	0x00019A20	0x00019A20	0x00019A20
		Revoke	Dsdptr	Selptr	Opts
func	desc	0x00019F10	0x00000000	0x00000000	0x00000002
func	addr	0x00019A20			

Method 2

Another method to locate the address is to use the value of the **kmid** pointer returned by the **sysconfig(SYS_KLOAD)** subroutine when loading the kernel extension. The **kmid** pointer points to the address of the load point routine. Hence to get the address of the load point, print the **kmid** value during the **sysconfig** call from the configuration method. Then go into the low level debugger and display the value pointed to by **kmid**. For clarity, set mnemonics for **kmid**.

```
> set kmid 1b131a4
> vars
Listing of the User-defined variables:
  kmid HEX=01B131A4
  fx HEX/DEC=01B1256E
  org
There are 15 free variable slots.
> d kmid
01B131A4  01B121EC 01B131E0 00000000 01B12578
|..!...1.....%x|
> d kmid>
01B121EC  7C0802A6 BFC1FFF8 90010008 9421FF80
||.....!..|
```

Method 3

If **kmid** is also not known, use the **find** command to locate the load point routine:

```
> find democonfig 1b00000
01B1256E  66616B65 636F6E66 69677C08 02A693E1
|democonfig|.....|
```

The **find** command will locate the specified string. It initiates a search from the starting address specified in the command. The string that is located is at the end of the **democonfig** routine. Now, backup to locate the beginning of the routine.

Usually all procedures have the instruction 7C0802A6 within the first three or four instructions of the procedure (within the first 12 to 16 bytes). See the assembler listing for the actual position of this instruction within the procedure. Use the **screen** command with the - flag to keep going back to locate the instruction. You can help speed up your search by using the ASCII section of the screen output to look for occurrences of the pipe symbol (|), which corresponds to the hexadecimal value 7C, the first byte of the instruction. Once this instruction is found, you can figure out where the start of the procedure is using the assembler listing as a guide.


```

> screen fx
GPR0 000078E4 2FF7FF70 000C5E78 00000000 2FF7FFF8 00000000 00007910 DEADBEEF
GPR8 DEADBEEF DEADBEEF DEADBEEF 7C0802A6 DEADBEEF DEADBEEF DEADBEEF DEADBEEF
GPR16 DEADBEEF DEADBEEF DEADBEEF DEADBEEF DEADBEEF DEADBEEF DEADBEEF DEADBEEF
GPR24 DEADBEEF DEADBEEF DEADBEEF DEADBEEF DEADBEEF DEADBEEF DEADBEEF 00007910
MSR 000090B0 CR 00000000 LR 0002506C CTR 000078E4
MQ 00000000 XER 00000000 SRR0 000078E4 SRR1 000090B0 DSISR 40000000
DAR 30000000 IAR 000078E4 (ORG+000078E4) ORG=00000000 Mode: VIRTUAL
000078E0 00000000 48000000 4E800020 00000000 |....H...N.. ....|
| b 0x78E4 (000078E4)
000078F0 000C0000 00000000 00000000 00000000 |.....|

01B12560 80020301 00000000 0000036C 000A6661 |.....!..fa|
01B12570 6B65636F 6E666967 7C0802A6 93E1FFFC |keconfig|.....|
01B12580 90010008 9421FFA0 83E20000 90610078 |.....!......a.x|
01B12590 9081007C 90A10080 90C10084 307F0294 |...|......0...|
01B125A0 48000535 80410014 80610078 5463043E |H..5.A...a.xTc.>|
01B125B0 90610038 80610078 48000491 9061003C |.a.8.a.xH....a.<|
01B125C0 28830000 41860020 8061003C 88630004 |(..A.. .a.<.c..|

```

```
> screen -
```

```
.
.
>
>
```

```

GPR0 000078E4 2FF7FF70 000C5E78 00000000 2FF7FFF8 00000000 00007910 DEADBEEF
GPR8 DEADBEEF DEADBEEF DEADBEEF 7C0802A6 DEADBEEF DEADBEEF DEADBEEF DEADBEEF
GPR16 DEADBEEF DEADBEEF DEADBEEF DEADBEEF DEADBEEF DEADBEEF DEADBEEF DEADBEEF
GPR24 DEADBEEF DEADBEEF DEADBEEF DEADBEEF DEADBEEF DEADBEEF DEADBEEF 00007910
MSR 000090B0 CR 00000000 LR 0002506C CTR 000078E4 MQ 00000000
XER 00000000 SRR0 000078E4 SRR1 000090B0 DSISR40000000 DAR 30000000
IAR 000078E4 (ORG+000078E4) ORG=00000000 Mode: VIRTUAL
000078E0 00000000 48000000 4E800020 00000000 |....H...N.. ....|
| b 0x78E4 (000078E4)
000078F0 000C0000 00000000 00000000 00000000 |.....|

01B121E0 00000000 00000000 00000000 7C0802A6 |.....|. ...|
01B121F0 BFC1FFF8 90010008 9421FF80 83E20000 |.....!. ...|
01B12200 90610098 9081009C 90A100A0 307F0040 |.a.....0..@|
01B12210 80810098 480008C1 80410014 307F0058 |...H...A..0..X|
01B12220 83C20008 63C40000 80A2000C 80C20010 |...c.....|
01B12230 480008A5 80410014 63C30000 80810098 |H...A..c.....|
01B12240 5484043E 90810038 38800000 9081003C |T..>...88.....<|

```

The start of the democonfig routine is at 0x01B121EC.

Method 4

If the load point routine is an exported routine, use the **map** command to locate the appropriate routine:

```
>map <routine name>
```

Method 5

You can also use the **crash** command to locate the load point. After running the **crash** command, run the **le** subcommand to list the load point for all the kernel extensions. The **knlist** subcommand will list the addresses of exported symbols:

```
$ crash
>le
>quit
```

The **le** subcommand shows the module start address. The first procedure in the kernel extension would follow the module header from the module start address. Hence in the case of the example demodd kernel extension, **le** showed the module start address to be 0x01B12000 and the democonfig procedure starts at 0x01B121EC.

You can locate the start of the democonfig procedure by searching for the first instruction of the democonfig procedure which would be usually 0x7C0802A6. Use the assembler listing to determine the first instruction.

First, display memory at 0x01b12000 and then use the **screen** subcommand to search ahead.

```
>screen 01b12000
>screen +
:
```

Method 6

Use the **find** command to search for a pattern:

```
> find democonfig 1b00000
01B1256E 66616B65 636F6E66 69677C08 02A693E1
|democonfig|.....|
```

We know that the module starts before 1B1256E. We also know that the "magic" number is 01DF. The loader identifies a file as a load module by looking for 01DF as the first two bytes in the file. So, the greatest address which is less than 1B1256E that contains 01DF, will be the start of the module, provided that it is on a page boundary. This means it has a mask of FFFFF000, a 4096 boundary or 0x1000:

```
> find 01df 01900000 * 2
```

Search starting at 1900000 through the kernel storage (the *) for 01DF on a 2-byte boundary.

The greatest address, on a page boundary, that is less than 1B1256E will be the module start. This will be offset 00000000 in the map file.

Change the Origin

Set the origin to the address of the load point. By default this is zero. By changing the origin to the address of the load point, you can directly correlate the address in the assembler listing with the address for the Instruction Address Register (IAR) and break points.

```
>set fkcfg 1B121EC      set a variable called fkcfg
>origin fkcfg
```

Set the Break Point

Now set the break point with the **break** command. Assume that we want to set the breakpoint at the assembler instruction at offset 218 (using the assembler listing):

```
>break +218 If origin has been set to load point
```

OR

```
>break 1B121EC+218
```

Viewing and Modifying Global Data

You can access the global data with two different methods. To understand how to locate the address of a global variable, we use the example of our demodd device driver. Here we try to view and modify the value of the data[] character array in the sample demodd device driver.

Use the first method only when you break in a procedure for the kernel extension to be debugged. You can use the second method at any time.

Method 1

1. After getting into the low level debugger, set a break point at the **demoread** procedure call. You can use any routine in demodd for this purpose.
2. Call the **demoread** routine. When the system breaks in **demoread** and invokes the debugger, the GPR2 (general purpose register 2) points to the TOC address. Now use the offset of the address of any global variable (from the start of TOC) to determine its address. The TOC is listed in the map file.

The "Map File" on page 378 shows that the address of the data[] array is at 0x53C while the TOC is at 0x52C. The offset of the address of the data[] array with respect to the start of TOC is $0x53C - 0x52C = 0x10$. Hence the address of the data[] variable is at (r2+10). And the actual data[] variable is located at the address value in (r2 + 10):

```
> d r2
01B131E0 01B12CCC 0004E7D0 01B13114 01B1311C |...,.....1...1.|
> d r2+10>
01B13124 61626364 65666768 696A6B6C 6D6E6F70 |abcdefghijklmnop|
```

Now we can change the value of the data[] variable. As an example, we change the first four bytes of data[] to "pppp" (p = 70):

```
> st r2+10> 70707070
> d r2+10>
01B13124 70707070 65666768 696A6B6C 6D6E6F70 |ppppefghijklmnop|
```

Method 2

You can use this method at any time. This method requires the map file and the address at which the relevant kernel address has been loaded. This method currently works because of the manner in which a kernel extension is loaded. But it may not work if the procedure for loading a kernel extension changes.

The address of a variable is illustrated in the following figure.

Address of the last
function before the
variable in the map file + Length of the
function + Offset of the
variable

The following is the section of the map file showing the data[] variable and the last function (xmfree) in the .text section:

```
26 000009B4 000024 2 GL SD S23 <.devswadd> glink.s(/usr/lib/glink.o)
27 000009B4 GL LD S24 .devswadd
28 000009D8 000024 2 GL SD S25 <.devswdel> glink.s(/usr/lib/glink.o)
29 000009D8 GL LD S26 .devswdel
30 000009FC 000024 2 GL SD S27 <.xmfree> glink.s(/usr/lib/glink.o)
31 000009FC GL LD S28 .xmfree
32 00000000 000444 4 RW SD S29 <_/tmp/cliff/demodd/demodd$c$>
/tmp/cliff/demodd/demodd.c(demodd.o)
33 00000450 000004 4 RW SD S30 demo_dev
/tmp/cliff/demodd/demodd.c(demodd.o)
34 00000460 000004 4 RW SD S31 demos_inited
/tmp/cliff/demodd/demodd.c(demodd.o)
35 00000470 000080 4 RW SD S32 data
/tmp/cliff/demodd/demodd.c(demodd.o)
36 * E 000004F0 00000C 2 DS SD S33 democonfig
/tmp/cliff/demodd/demodd.c(demodd.o)
37 E 000004FC 00000C 2 DS SD S34 demoopen
/tmp/cliff/demodd/demodd.c(demodd.o)
```

The last function in the `.text` section is at lines 30-31. The offset address of this function from the map is 0x000009FC (line 30, column 2). The length of the function is 0x000024 (line 30, column 3). The offset address of the `data[]` variable is 0x00000470 (line 35, column 2). Hence the offset of the address of the `data[]` variable is:

```
0x000009FC + 0x000024 + 0x00000470 = 0x00000E90
```

Add this address value to the load point value of the `demodd` kernel extension. If, as in the case of the sample `demodd` device handler, this is 0x1B131A4, then the address of the `data[]` variable is:

```
0x1B121EC + 0x00000E90 = 0x1B1307C
>display 1B1307C
01B1307C 61626364 65666768 696A6B6C 6D6E6F70 |abcdefghijklmnop|
```

Now change the value of the `data[]` variable as in Method 1.

Note that in Method 1, using the TOC, you found the address of the address of `data[]`, while in Method 2 you simply found the address of `data[]`.

Displaying Registers on a Micro Channel Adapter

When you write a device driver for a new Micro Channel adapter, you often want to be able to read and write to registers that reside on the adapter. This is a way of seeing if the hardware is functioning correctly. For example, to examine a register on the Token Ring adapter, first see where this adapter resides in the bus I/O space:

```
$!sdev -C
sys0      Available 00-00 System Object
sysunit0  Available 00-00 System Unit
sysplanar0 Available 00-00 CPU Planar
.
.
scsi0     Available 00-01 SCSI I/O Controller
tok0      Available 00-02 Token-Ring High-Performance Adapter
ent0      Available 00-03 Ethernet High-Performance LAN Adapter
$!sattr -l tok0 -E
bus_intr_lvl 3 Bus interrupt level False
intr_priority 3 Interrupt priority False
.
.
rdto      92 RECEIVE DATA TRANSFER OFFSET True
bus_io_addr 0x86a0 Bus I/O address False
dma_lvl 0x5 DMA arbitration level False
dma_bus_mem 0x202000 Address of bus memory used DMA False
```

We now know that the token ring adapter is located at 0x86A0.

To read a specific register, enter the kernel debugger and use the `sregs` command to display the segment registers. Find an unused segment register (=007FFFFF). For this example, assume `s9` is not used. Enable the Micro Channel bus addressing with the `set` command:

```
set s9 820c0020
```

Use the `sregs` command to display the segment register values to check that you typed it in correctly.

From the *POWERstation and POWERserver Hardware Technical Information-Options and Devices*, we know that the address of the Adapter Communication and Status

register is P6a6. The value of P is based on the Bus I/O address (bus_io_addr) of the adapter. In the above example, this is 86A0. It could have been anything from 86A0 to F6A0 on a 0x1000 byte boundary. Hence P is 8, and the address of the Communication and Status register is 86A6. The **display** command now displays the two-byte register:

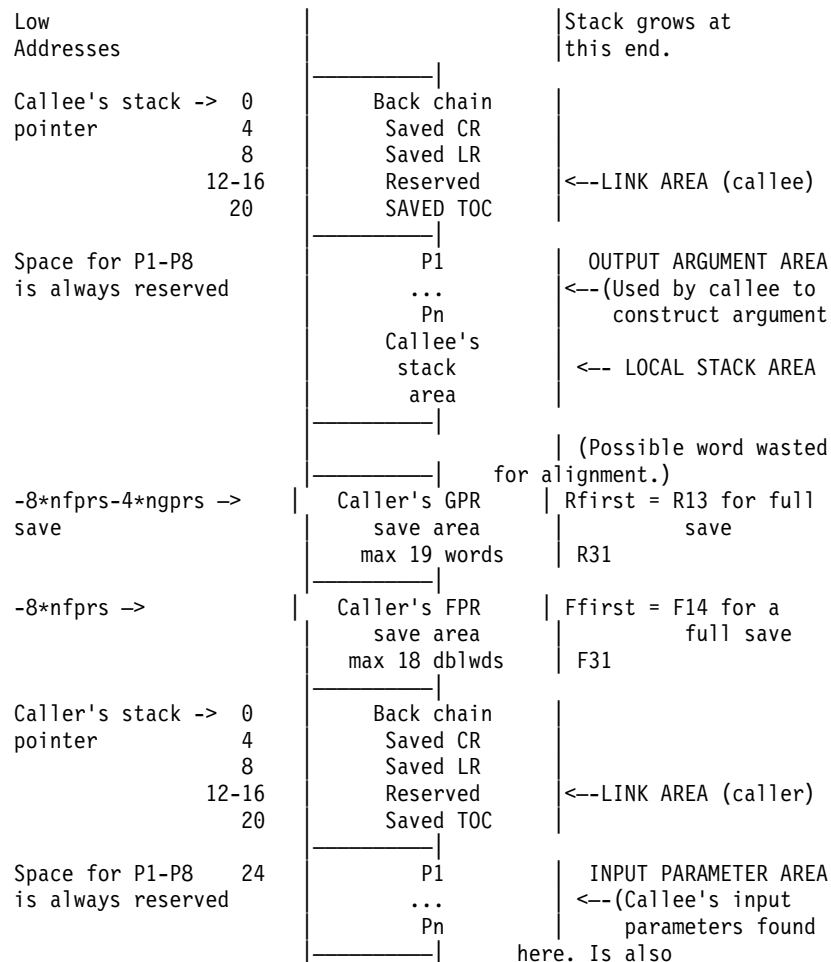
```
d 900086a6 2
```

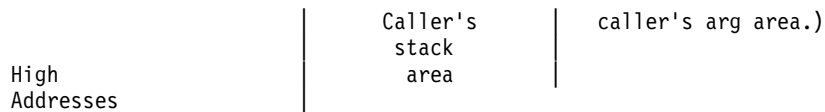
The key is to load a segment register with 820c0020 and then use that segment register to reference registers and memory on your adapter. You can use the same method to access registers resident on the IOCC. In that case, load the segment register with a value of 820c00e0.

Stack Trace

The stack trace gives the stack history which provides the sequence of procedure calls leading to the current IAR. The **Ret Addr** is the address of the instruction calling this procedure. You can use the map file to locate the name of the procedure. Note that the first stack frame shown is almost always useless, since data either has not been saved yet, or is from a previous call. The last function preceding the **Ret Addr** is the function that called the procedure.

You can also use the **map** command to locate the function name if the function was exported. The **map <addr>** command locates the symbol before the given address. The following is a concise view of the stack:





The following is a sample stack history with a break in the sample **demodd** kernel extension. The breakpoint was set at the start of the **demoread** routine at 0x1B127D4 (Beginning IAR). This was called from an instruction at 0x000824B0 (**Ret Addr**). This in turn is called by the instruction at address 0x00085F54 (**Ret Addr**), and so on.

The low values of the addresses (0x000824B0 and 0x00085F54) suggest that the instructions are in **/unix**. You can use the **crash** command and the **le** subcommand to determine the right kernel extension that is loaded in an address range.

```
0x1b127d4          beginning demoread in demodd
0x000824b0          .rdevread in /unix
0x00085f54          .cdev_rdwr in /unix
```

```
> stack
Beginning IAR: 0x01B127D4      Beginning Stack: 0x2FF97C28
Chain:0x2FF97C88 CR:0x24222082 Ret Addr:0x000824B0 TOC:0x000C5E78
P1:0x2003F800 P2:0x2003F800 P3:0x0000008C P4:0x00000001
P5:0x01B11200 P6:0x00000000 P7:0x2FF97D38 P8:0x00000000
2FF97C60 00000203 00000000 2FF97CF8 2FF7FCD0 |...../|. /...|
2FF97C70 29057E6B 00001000 2FF97DC0 018E8BE0 |). k..../}. ....|
2FF97C80 00FF0000 00000000 2FF97CD8 22222044 |...../|. "" D|
Returning to Stack frame at 0x2FF97C88
Press ENTER to continue or x to exit:
>
Chain:0x2FF97CD8 CR:0x22222044 Ret Addr:0x00085F54 TOC:0x00000000
P1:0x00000000 P2:0x018C41E0 P3:0x2FF97CF8 P4:0x2FF7FCC8
P5:0x000850E0 P6:0x00000000 P7:0xDEADBEEF P8:0xDEADBEEF
2FF97CC0 DEADBEEF DEADBEEF 00000000 000BE4F8 |.....|
2FF97CD0 001E70F8 000BE7A4 2FF97D28 000BE5AC |..p..../}. ....|
Returning to Stack frame at 0x2FF97CD8
Press ENTER to continue or x to exit:
...
>
Chain:0x00000000 CR:0x22222022 Ret Addr:0x0000238C TOC:0x00000000
P1:0x00000003 P2:0x30000000 P3:0x00000080 P4:0x00000000
P5:0x00000000 P6:0x00000000 P7:0x00000000 P8:0x00000000
Returning to Stack frame at 0x0
Press ENTER to continue or x to exit:
> Trace back complete.
```

Error Messages for the LLDB Kernel Debug Program

The following error messages can appear while using the LLDB Kernel Debug Program:

- Bad type – trace terminated.
A trace event was found that had an incorrect hookword type, and the traceback was terminated. This message is for your information only.
- Channel out of range.
You entered a value that is outside of the numeric range of acceptable channel numbers. Enter the command again, selecting a channel in the range displayed in the prompt.
- Do you want to continue the search? (Y/N)

Ten consecutive pages were not in storage. To continue the search, enter Y (yes). To exit the search enter N (no).

- The address you specified is not in real storage.
The command was rejected because the data at the address you specified has been paged out of RAM to disk. Enter the command again with a data address that is currently in RAM.
- The page at Address is not in real storage.
The search passed over a page that was not in storage. Action is not required. This message is for your information only.
- The value cannot be found.
You specified a value that cannot be found or was not in real storage. Action is not required. This message is for your information only.
- This breakpoint is undefined or not currently addressable.
The breakpoint was not cleared because it is undefined or its segment is not currently addressable. Try to load the segment ID into a segment register with the **set** command.
- Timestamp paged out.
A trace data structure is not currently paged into physical memory. Enter the command again later when the data structure is available.
- Trace data paged out.
A trace data structure is not currently paged into physical memory. Enter the command again later when the data structure is available.
- Trace entry paged out.
A trace data structure is not currently paged into physical memory. Enter the command again later when the data structure is available.
- Trace header paged out.
A trace data structure is not currently paged into physical memory. Enter the command again later when the data structure is available.
- Trace Queue header paged out.
A trace data structure is not currently paged into physical memory. Enter the command again later when the data structure is available.
- You cannot set more than 32 breakpoints.
The breakpoint is not set because you tried to set more than the maximum number of breakpoints allowed on the system. Clear at least one breakpoint before setting another breakpoint.
- You cannot Step or Go into paged-out storage.
The command cannot run because you specified an address for the command that is in paged-out storage. Specify an address that is not in paged-out storage.
- You did not enter all required parameters.
The command was unsuccessful because you did not specify all the required parameters. Enter the command again with the necessary parameters.
- You entered a parameter that is not valid.
The command was unsuccessful because you specified a parameter that the debug program did not recognize. Check the spelling and syntax of the parameter you specified. Then, enter the command again with a valid parameter.

KDB Kernel Debugger and Command

This chapter provides information about the KDB Kernel Debugger and **kdb** Command. The **kdb** Command is primarily used for analysis of system dumps. The KDB Kernel Debugger is primarily used as a debugging tool for device driver development. The following topics are included in this chapter:

- KDB Kernel Debugger and **kdb** Command
- Subcommands for the KDB Kernel Debugger and **kdb** Command
- Using the KDB Kernel Debug Program

KDB Kernel Debugger and **kdb** Command

This document describes the KDB Kernel Debugger and **kdb** command. It is important to understand that the KDB Kernel Debugger and the **kdb** command are two separate entities. The KDB Kernel Debugger is a debugger for use in debugging the kernel, device drivers, and other kernel extensions. The **kdb** command is primarily a tool for viewing data contained in system image dumps. However, the **kdb** command may be run on an active system to view system data.

The reason that the KDB Kernel Debugger and **kdb** command are both covered together is that they share a large number of subcommands. This provides for ease of use when switching from between the kernel debugger and command. Most subcommands for viewing kernel data structures are included in both. However, the KDB Kernel Debugger includes additional subcommands for execution control (breakpoints, step commands, etc...) and processor control (start/stop CPUs, reboot, etc...). The **kdb** command also has subcommands which are unique; these involve manipulation of system image dumps.

The following sections outline how to invoke the KDB Kernel Debugger and **kdb** command. They also describe features which are unique to each.

- “The **kdb** Command”
- “KDB Kernel Debugger” on page 391
- “Loading and Starting the KDB Kernel Debugger” on page 391
- “Using a Terminal with the KDB Kernel Debugger” on page 392
- “Entering the KDB Kernel Debugger” on page 392
- “Debugging Multiprocessor Systems” on page 393
- “Kernel Debug Program Concepts” on page 393

The complete list of subcommands available for the KDB Kernel Debugger and **kdb** command are included in “Subcommands for the KDB Kernel Debugger and **kdb** Command” on page 394.

The **kdb** Command

The **kdb** command is an interactive tool that allows examination of an operating system image. An operating system image is held in a system dump file; either as a file or on the dump device. The **kdb** command may also be used on an active system for viewing the contents of system structures. This is a useful tool for device driver development and debugging. The syntax for invoking the **kdb** command is:

```
kdb [SystemImageFile [KernelFile]]
```


The *SystemImageFile* parameter specifies the file that contains the system image. The default *SystemImageFile* is **/dev/mem**. The *KernelFile* parameter contains the kernel symbol definitions. The default for the *KernelFile* is **/usr/lib/boot/unix**.

Root permissions are required for execution of the **kdb** command on the active system. This is required because the special file **/dev/mem** is used. To run the **kdb** command on the active system, enter:

```
kdb
```

To invoke the **kdb** command on a system image file, enter:

```
kdb SystemImageFile
```

where *SystemImageFile* is either a file name or the name of the dump device. When invoked to view data from a *SystemImageFile* the **kdb** command sets the default thread to the thread running at the time the *SystemImageFile* was created.

Notes:

1. When using the **kdb** command a kernel file must be available.
2. Stack tracing of the current process on a running system does not work

KDB Kernel Debugger

The KDB Kernel Debugger is used for debugging the kernel, device drivers, and other kernel extensions. The KDB Kernel Debugger provides the following functions:

- Setting breakpoints within the kernel or kernel extensions
- Execution control through various forms of step commands
- Formatted display of selected kernel data structures
- Display and modification of kernel data
- Display and modification of kernel instructions
- Modification of the state of the machine through alteration of system registers

Loading and Starting the KDB Kernel Debugger

The KDB Kernel Debugger must be loaded at boot time. This requires that a boot image be created with the debugger enabled. To enable the KDB Kernel Debugger the **bosdebug** and **bosboot** commands must be used. The first step in preparing a boot image to include the KDB Kernel Debugger is to set an indicator that the KDB Kernel is to be used in building the boot image, this indicator is checked by the **bosboot** command which actually builds the boot images. To set the indicator that a KDB Kernel is to be used in building building boot images is accomplished by executing the following command:

```
bosdebug -K
```

This indicator may be turned off by executing the following command:

```
bosdebug -o
```

Note, execution of the above command clears all flags set by previous invocations of **bosdebug**. Therefore, it may be necessary to execute additional **bosdebug** commands if there are debug options which need to be enabled.

To complete the task of building a boot image with the KDB Kernel, the **bosboot** command must be executed. The following commands may be used to build a boot image using a KDB Kernel:

1. `bosboot -a -d /dev/ipldevice`
2. `bosboot -a -d /dev/ipldevice -D`
3. `bosboot -a -d /dev/ipldevice -I`

The above commands build boot images using the KDB Kernel (if previously selected by **bosdebug**) having the following characteristics: 1) KDB Kernel debugger is disabled, 2) KDB Kernel Debugger is enabled but is not invoked during system initialization, 3) KDB Kernel Debugger is enabled and is invoked during system initialization. Note, execution of the **bosboot** builds the boot image only; the boot image is not used until the machine is restarted.

Notes:

1. External interrupts are disabled while the KDB Kernel Debugger is active
2. If invoked during system initialization the **g** subcommand must be issued to continue the initialization process.

Using a Terminal with the KDB Kernel Debugger

The KDB Kernel Debugger opens an asynchronous ASCII terminal when it is first started, and subsequently upon being started due to a system halt. Native serial ports are checked sequentially starting with port 0 (zero). Each port is configured at 9600 bps, 8 bits, and no parity. If carrier detect is asserted within 1/10 seconds, then the port is used. Otherwise, the next available native port is checked. This process continues until a port is opened or until every native port available on the machine has been checked. If no native serial port is opened successfully, then the result is unpredictable.

The KDB Kernel Debugger only supports display to an ASCII terminal connected to a native serial port. Displays connected to graphics adapters are *not* supported. The KDB Kernel Debugger has its own device driver for handling the display terminal. It is possible to connect a serial line between two machines and define the serial line port as the port for the console. In that case, the **cu** command may be used to connect to the target machine and run the KDB Kernel Debugger.

Note: If a serial device, other than a terminal connected to a native serial port, is selected by the kernel debugger, the system may appear to hang up.

Entering the KDB Kernel Debugger

It is possible to enter the KDB Kernel Debugger using one of the following procedures:

- From a native keyboard, press `Ctrl-Alt-Numpad4`.
- From a tty keyboard, enter `Ctrl-4` (IBM 3151 terminals) or `Ctrl-\` (BQ 303, BQ 310C, and WYSE 50).
- The system can enter the debugger if a breakpoint is set. To do this, use one of the `Breakpoints/Steps` Subcommands.
- The system can also enter the debugger by calling the **brkpoint** subroutine from C code. The syntax for calling this subroutine is:
`brkpoint();`
- The system can also enter the debugger if a system halt is caused by a fatal system error. In such a case, the system creates a log entry in the system log and if the KDB Kernel Debugger is available, it is called. A system dump may be generated on exit from the debugger.

If the kernel debug program is not available (nothing happens when you type in the above key sequence), you must load it. To do this, see “Loading and Starting the KDB Kernel Debugger” on page 391.

Note: You can use the **kdb** command to determine whether the KDB Kernel Debugger is available. Use the **dw** subcommand:

```
# kdb
(0)> dw kdb_avail
(0)> dw kdb_wanted
```

If either of the above **dw** subcommands returns a 0, the KDB Kernel Debugger is not available.

Once the KDB Kernel Debugger has been invoked the subcommands detailed in Subcommands for the KDB Kernel Debugger and **kdb** Command are available.

Debugging Multiprocessor Systems

On multiprocessor systems, entering the KDB Kernel Debugger stops all processors (except the current processor running the debug program itself). The prompt on multiprocessor systems indicates the current processor. For example:

- KDB(0)> - indicates processor 0 is the current processor
- KDB(5)> - indicates processor 5 is the current processor

In addition to the change in the prompt for multiprocessor systems, there are also subcommands which are unique to these systems. Refer to SMP Subcommands for details.

Kernel Debug Program Concepts

When the KDB Kernel Debugger is invoked, it is the only running program. All processes are stopped and interrupts are disabled. The KDB Kernel Debugger runs with its own Machine State Save Area (mst) and a special stack. In addition the KDB Kernel Debugger does not run AIX routines. Though this requires that kernel code be duplicated within KDB, it is possible to break anywhere within the kernel code. When exiting the KDB Kernel Debugger, all processes continue to run unless the debugger was entered via a system halt.

Commands

The KDB Kernel debugger must be loaded and started before it can accept commands. Once in the debugger, use the commands to investigate and make alterations. See Subcommands for the KDB Kernel Debugger and KDB Command for lists and descriptions of the subcommands.

Breakpoints

The KDB Kernel Debugger creates a table of breakpoints that it maintains. When a breakpoint is set, the debugger temporarily replaces the corresponding instruction with the trap instruction. The instruction overlaid by the breakpoint operates when you issue any subcommand that would cause that instruction to be executed.

For more information on setting/clearing breakpoints and execution control see “Setting Breakpoints” on page 530.

Subcommands for the KDB Kernel Debugger and kdb Command

Introduction to Subcommands

Numeric Values

Numeric arguments for the subcommands presented in this section are required to be hexadecimal, except for arguments that specify slot numbers and arguments to the **dcal** subcommand. Slot numbers and arguments for the **dcal** subcommand must be decimal values.

Registers

Register values may be referenced by the KDB Kernel Debugger and **kdb** command. Register values may be used in subcommands by preceding the register name with an "@" character. This character is also used to deference addresses as explained later. The list of registers that may be referenced include:

asr	Address space register
cr	Condition register
ctr	Count register
dar	Data address register
dec	Decrementer
dsisr	Data storage interrupt status register
fp0-fp31	Floating point registers 0 through 31.
fpscr	Floating point status and control register
iar	Instruction address register
lr	Link register
mq	Multiply quotient
msr	Machine State register
r0-r31	General Purpose Registers 0 through 31
rtcl	Real Time clock (nanoseconds)
rtcu	Real Time clock (seconds)
s0-s15	Segment registers.
sdr0	Storage description register 0
sdr1	Storage description register 1
srr0	Machine status save/restore 0
srr1	Machine status save/restore 1
tbl	Time base register, lower
tbu	Time base register, upper
tid	Transaction register (fixed point)
xer	Exception register (fixed point)

Other special purposes registers that may be referenced, if supported on the hardware, include: **sprg0**, **sprg1**, **sprg2**, **sprg3**, **pir**, **fpecr**, **ear**, **pvr**, **hid0**, **hid1**, **iabr**, **dmiss**, **imiss**, **dcmp**, **icmp**, **hash1**, **hash2**, **rpa**, **buscsr**, **l2cr**, **l2sr**, **mmcr0**, **mmcr1**, **pmc1-pmc8**, **sia**, and **sda**.

Expressions

The KDB Kernel Debugger and **kdb** command do not provide full expression processing. Expressions can only contain symbols, hex constants, references to register or memory locations, and operators. Furthermore, symbols are only allowed as the first operand of an expression. Supported operators include:

+	Addition
-	Subtraction
*	Multiplication

/ Division
 @ Dereferencing

The dereference operator indicates that the value at the location indicated by the next operand is to be used in the calculation of the expression. For example, @f000 would indicate that the value at address 0x0000f000 should be used in evaluation of the expression. The dereference operator is also used to access the contents of register. For example, @r1 references the contents of general purpose register 1. Recursive dereferencing is allowed. As an example, @@r1 references the value at the address pointed to by the value at the address contained in general purpose register 1.

Expressions are processed from left to right only. There is no operator precedence.

Examples

Valid Expressions	Results
dw @r1	displays data at the location pointed to by r1
dw @@r1	displays data at the location pointed to by value at location pointed to by r1
dw open	displays data at the address beginning of the open routine
dw open+12	displays data twelve bytes past the beginning of the open routine
Invalid Expressions	Problem
dw @r1+open	symbols can only be the first operand
dw r1	must include @ to reference the contents of r1, if a symbol r1 existed this would be valid
dw @r1+(4*3)	parentheses are not supported

Subcommand Arguments

The following table describes the most common argument types referenced in the subcommand syntax diagrams that follow.

Argument	Description
*	A wildcard used to select all entries.
count	A hex constant specifying the number of times to perform a specific operation.
cpu	A decimal value specifying a cpu number in a SMP machine
dev eaddr	An effective address for device memory.
dev paddr	A physical address for device memory.
eaddr	Effective address. This may be a hex constant or an expression.
phys. addr	A physical address
pid	A hex constant or expression specifying a process ID.
selection	Indicates that a menu is displayed from which a selection must be made.
slot	A decimal constant specifying a slot number within a table.
symb	A symbolic reference to a value. Symbols from the kernel and/or kernel extensions may be used.

Argument	Description
tid	A hex constant or expression specifying a thread ID.

KDB Kernel Debug Program Subcommands grouped in Alphabetical Order

The following table shows the KDB Kernel Debug Program subcommands in alphabetical order:

Subcommand	Alias	Alias	Function	Argument	Task Category
ames	-	-	VMM address map entries	[symb/eaddr]	VMM
apt	-	-	VMM APT entries	[selection]	VMM
asc	ascsi	-	Display ascsi	[slot/symb/eaddr]	SCSI
B	-	-	step on branch	[count]	Breakpoints/Steps
b	brk	-	set/list break point(s)	[-p/-v] [symb/addr]	Breakpoints/Steps
bt	-	-	set/list trace point(s)	[-p/-v] [symb/addr [script] [cond]]	Trace
btac	-	-	branch target	[-p/-v] [symb/eaddr]	btac/BRAT
buf	buffer	-	Display buffer	[slot/symb/eaddr]	File System
c	cl	-	clear break point	[slot [-p/-v] symb/addr]	Breakpoints/Steps
ca	-	-	clear all break points	-	Breakpoints/Steps
cat	-	-	clear all trace points	-	Trace
cbtac	-	-	clear branch target	-	btac/BRAT
cdt	-	-	Display cdt	[?]	Basic
cku	cku_priv	-	Display cku private	symb/eaddr	File System
clk	cpl	-	Display complex lock	[symb/eaddr]	System Table
cpu	-	-	Switch to cpu	[cpu number any]	SMP
ct	-	-	clear trace point	slot [-p/-v] symb/addr	Trace
ctx	-	context	switch to KDB context	[cpu number]	Basic
cw	stop-cl	-	clear watch	-	Watch
d	dump	-	display byte data	symb/eaddr [count]	Dumps/Display/Decode
dbat	-	-	display dbats	[index]	bat/Block Address Translation
dc	dis	-	display code	symb/eaddr [count]	Dumps/Display/Decode
dcal	-	-	calc/conv a decimal expr	decimal expression	Calculator Converter
dd	-	-	display double word data	symb/eaddr [count]	Dumps/Display/Decode

Subcommand	Alias	Alias	Function	Argument	Task Category
ddpb	-	-	display device byte	dev paddr [count]	Dumps/Display/Decode
ddpd	-	-	display device double word	dev paddr [count]	Dumps/Display/Decode
ddph	-	-	display device half word	dev paddr [count]	Dumps/Display/Decode
ddpw	-	-	display device word	dev paddr [count]	Dumps/Display/Decode
ddvb	diob	-	display device byte	dev eaddr [count]	Dumps/Display/Decode
ddvd	diod	-	display device double word	dev eaddr [count]	Dumps/Display/Decode
ddvh	dioh	-	display device half word	dev eaddr [count]	Dumps/Display/Decode
ddvw	diow	-	display device word	dev eaddr [count]	Dumps/Display/Decode
debug	-	-	enable/disable debug	[?]	Miscellaneous
dev	devsw	-	Display devsw table	[symb/address/major]	System Table
devno	devnode	-	Display devnode	[slot/symb/eaddr]	File System
dp	-	-	display byte data	phys. addr [count]	Dumps/Display/Decode
dpc	-	-	display code	phys. addr [count]	Dumps/Display/Decode
dpd	-	-	display double word data	phys. addr [count]	Dumps/Display/Decode
dpw	-	-	display word data	phys. addr [count]	Dumps/Display/Decode
dr	-	-	display registers	[gp sr sp]	Dumps/Display/Decode
dw	-	-	display word data	symb/eaddr [count]	Dumps/Display/Decode
e	q	g	exit	[dump]	Basic
exp	-	-	list export tables	[symb]	Kernel Extension Loader
ext	-	-	extract pattern	[-p] symb/eaddr delta [size [count]]	Dumps/Display/Decode
extp	-	-	extract pattern	[-p] phys. addr delta [size [count]]	Dumps/Display/Decode
f	stack	where	stack frame trace	[+x/-x][th {slot/eaddr}]	Basic
fb	fbuffer	-	Display freelist	[bucket/symb/eaddr]	File System
fifono	fifonode	-	Display fifonode	slot/symb/eaddr	File System
file	-	-	Display file	[slot/symb/eaddr]	File System
find	-	-	find pattern	[-s] symb/eaddr pattern [mask [delta]]	Dumps/Display/Decode
findp	-	-	find pattern	[-s] phys. addr pattern [mask [delta]]	Dumps/Display/Decode
fino	icache	-	Display icache list	[slot/symb/eaddr]	File System
gfs	-	-	Display gfs	symb/eaddr	File System
gno	gnode	-	Display gnode	symb/eaddr	File System

Subcommand	Alias	Alias	Function	Argument	Task Category
gt	-	-	go until address	[-p/-v] symb/addr	Breakpoints/Steps
h	?	help	help	topic	Basic
hb	hbuffer	-	Display buffehash	[bucket/symb/eaddr]	File System
hcal	cal	-	calc/conv a hexa expr	hexa expression	Calculator Converter
hino	hinode	-	Display inodehash	[bucket/symb/eaddr]	File System
his	hi	hist	print history	[?][count]	Basic
hno	hnode	-	isplay hnodehash	[bucket/symb/eaddr]	File System
hp	heap	-	Display kernel heap	[symb/eaddr]	Memory Allocator
ibat	-	-	display ibats	[index]	bat/Block Address Translation
ifnet	-	-	Display interface	[slot/symb/eaddr]	NET
ino	inode	-	Display inode	[slot/symb/eaddr]	File System
intr	-	-	@Display int handler	[slot/symb/eaddr]	Process
ipc	-	-	IPC information	-	VMM
ipl	iplcb	-	Display ipl proc info	[/cpu index]	System Table
kmbucket	bucket	-	Display kmembuckets	[?][-l][-c n][-i n][adr]	Memory Allocator
kmstats	-	-	Display kmemstats	[symb/eaddr]	Memory Allocator
lb	lbrk	-	set/list local bp(s)	[-p/-v] [symb/addr]	Breakpoints/Steps
lbtac	-	-	local branch target	[-p/-v] [symb/eaddr]	btac/BRAT
lc	lcl	-	clear local bp	[slot [-p/-v] symb/addr [ctx]]	Breakpoints/Steps
lcbtac	-	-	clear local br target	-	btac/BRAT
lcw	lstop-cl	-	clear local watch	-	Watch
lka	lockanch	tblk	VMM lock anchor/tblock	[slot/symb/eaddr]	VMM
lke	-	-	list loaded extensions	[?][-l] [slot symb/eaddr]	Kernel Extension Loader
lkh	lockhash	-	VMM lock hash	[slot/symb/eaddr]	VMM
lkw	lockword	-	VMM lock word	[slot/symb/eaddr]	VMM
lq	lockq	-	Display lock queues	[bucket/symb/eaddr]	Process
lvol	lvol	-	Display logical vol	symb/eaddr	LVM
lwr	lstop-r	-	local stop on read data	[[-p/-v] symb/addr [size]]	Watch
lwrw	lstop-rw	-	local stop on r/w data	[[-p/-v] symb/addr [size]]	Watch
lww	lstop-w	-	local stop on write data	[[-p/-v] symb/addr [size]]	Watch

Subcommand	Alias	Alias	Function	Argument	Task Category
m	-	-	modify sequential bytes	symb/eaddr	Modify Memory
mbuf	-	-	Display mbuf	[tcp/udp] [symb/eaddr]	NET
md	-	-	modify sequential double word	symb/eaddr	Modify Memory
mdbat	-	-	modify dbats	[index]	bat/Block Address Translation
mdpb	-	-	modify device byte	dev paddr	Modify Memory
mdpdp	-	-	modify device double word	dev paddr	Modify Memory
mdph	-	-	modify device half	dev paddr	Modify Memory
mdpwp	-	-	modify device word	dev paddr	Modify Memory
mdvb	miob	-	modify device byte	dev eaddr	Modify Memory
mdvd	miod	-	modify device double word	dev eaddr	Modify Memory
mdvh	mioh	-	modify device half	dev eaddr	Modify Memory
mdvw	miow	-	modify device word	dev eaddr	Modify Memory
mibat	-	-	modify ibats	[index]	bat/Block Address Translation
mp	-	-	modify sequential bytes	phys. addr	Modify Memory
mpdp	-	-	modify sequential double word	phys. addr	Modify Memory
mpwp	-	-	modify sequential word	phys. addr	Modify Memory
mr	-	-	modify registers	[gp sr sp]	Modify Memory
mst	-	-	Display mst area	[slot] [[-a] symb/eaddr]	Process
mw	-	-	modify sequential word	symb/eaddr	Modify Memory
n	nexti	-	next instruction	[count]	Breakpoints/Steps
nm	-	-	translate symbol to eaddr	symb	Namelist/Symbol
ns	-	-	no symbol mode (toggle)	-	Namelist/Symbol
p	proc	-	Display proc table	[*/slot/symb/eaddr]	Process
pbuf	pbuf	-	Display physical buf	[*] symb/eaddr	LVM
pdt	-	-	VMM paging device table	[*][slot]	VMM
pfhdata	-	-	VMM control variables	-	VMM
pft	-	-	VMM PFT entries	[selection]	VMM

Subcommand	Alias	Alias	Function	Argument	Task Category
ppda	-	-	Display per processor data area	[/cpunb/symb/eaddr]	Process
pta	-	-	VMM PTA segment	[?]	VMM
pte	-	-	VMM PTE entries	[selection]	VMM
pvol	pvol	-	Display physical vol	symb/eaddr	LVM
r	return	-	go to end of function	-	Breakpoints/Steps
reboot	kill	-	reboot the machine	-	machdep
rmap	-	-	VMM RMAP	[*][slot]	VMM
rmst	-	-	remove symbol table	slot symb/eaddr	Kernel Extension Loader
rno	rnode	-	Display rnode	symb/eaddr	File System
rq	runq	-	Display run queues	[bucket/symb/eaddr]	Process
s	stepi	ste	single step	[count]	Breakpoints/Steps
S	-	-	step on bl/blr	[count]	Breakpoints/Steps
scb	-	-	VMM segment control blocks	[selection]	VMM
scd	scdisk	-	Display scdisk	[slot/symb/eaddr]	SCSI
segst64	-	-	VMM SEGSTATE	[-p pid][-e esid][[-s flag] [fno shm]]	VMM
set	-	setup	display/update kdb toggles	[toggle]	Basic
slk	spl	-	Display simple lock	[symb/eaddr]	System Table
sock	-	-	Display socket	[tcp/udp] [symb/eaddr]	NET
specno	specnode	-	Display specnode	symb/eaddr	File System
sq	sleepq	-	Display sleep queues	[bucket/symb/eaddr]	Process
sr64	-	-	VMM SEG REG	[-p pid] [esid] [size]	VMM
start	-	-	Start cpu	cpu number all	SMP
stat	-	-	system status message	-	Machine Status
stbl	-	-	list loaded symbol tables	[slot symb/eaddr]	Kernel Extension Loader
ste	-	-	VMM STAB	[-p pid]	VMM
stop	-	-	Stop cpu	cpu number all	SMP
switch	sw	-	switch thread	[[{th {slot/eaddr} {u/k}}]	Machine Status
tcb	-	-	Display TCBS	[slot/symb/eaddr]	NET

Subcommand	Alias	Alias	Function	Argument	Task Category
tcpcb	-	-	Display TCP CB	[tcp/udp] [symb/eaddr]	NET
test	[-	bt condition	[symb/eaddr == symb/eaddr]	Conditional
test	[-	bt condition	[symb/eaddr != symb/eaddr]	Conditional
test	[-	bt condition	[symb/eaddr >= symb/eaddr]	Conditional
test	[-	bt condition	[symb/eaddr <= symb/eaddr]	Conditional
test	[-	bt condition	[symb/eaddr > symb/eaddr]	Conditional
test	[-	bt condition	[symb/eaddr < symb/eaddr]	Conditional
test	[-	bt condition	[symb/eaddr ^ symb/eaddr]	Conditional
test	[-	bt condition	[symb/eaddr & symb/eaddr]	Conditional
test	[-	bt condition	[symb/eaddr symb/eaddr]	Conditional
time	-	-	display elapsed time	-	Miscellaneous
th	thread	-	Display thread table	[*/slot/symb/eaddr/-w ?]	Process
tpid	th_pid	-	Display thread pid	[pid]	Process
tr	-	-	translate to real address	symb/eaddr	Address Translation
trace	-	-	Display trace buffer	[?]	System Table
trb	timer	-	Display system timer request blocks	-	System Table
ts	-	-	translate eaddr to symbol	eaddr	Namelist/Symbol
ttid	th_tid	-	Display thread tid	[tid]	Process
tv	-	-	display MMU translation	symb/eaddr	Address Translation
u	user	-	Display u_area	[-?][slot/symb/eaddr]	Process
udb	-	-	Display UDBs	[slot/symb/eaddr]	NET
var	-	-	Display var	-	System Table
vfs	mount	-	Display vfs	[slot/slot/symb/eaddr]	File System
vmdmap	-	-	VMM disk map	[slot/symb/eaddr]	VMM
vmlocks	vmlock	v1	VMM spin locks	-	VMM
vmaddr	-	-	VMM Addresses	-	VMM
vmker	-	-	VMM kernel segment data	-	VMM
vmlog	-	-	VMM error log	-	VMM

Subcommand	Alias	Alias	Function	Argument	Task Category
vmstat	-	-	VMM statistics	-	VMM
vmwait	-	-	VMM wait status	[symb/eaddr]	VMM
vno	vnode	-	Display vnode	symb/eaddr	File System
volgrp	volgrp	-	Display volume group	symb/eaddr	LVM
vrlld	-	-	VMM reload xlate table	-	VMM
vsc	vscsi	-	Display vscsi	[slot/symb/eaddr]	SCSI
wr	stop-r	-	stop on read data	[[<i>-p/-v</i>] symb/addr [size]]	Watch
wrw	stop-rw	-	stop on r/w data	[[<i>-p/-v</i>] symb/addr [size]]	Watch
ww	stop-w	-	stop on write data	[[<i>-p/-v</i>] symb/addr [size]]	Watch
xm	xmalloc	-	Display heap debug	[-?]	Memory Allocator
zproc	-	-	VMM zeroing kproc	-	VMM

KDB Kernel Debug Subcommands grouped by Task Category

Basic Subcommands

Subcommand	Alias	Alias	Function	Argument
h	?	help	help	topic
his	hi	hist	print history	[?][count]
e	q	g	exit	[dump]
set	-	setup	display/update kdb toggles	[toggle]
f	stack	where	stack frame trace	[+x/-x][th {slot/eaddr}]
ctx	-	context	switch to KDB context	[cpu number]
cdt (“cdt Subcommand” on page 419)	-	-	Display cdt	[?]

Trace Subcommands

Subcommand	Alias	Alias	Function	Argument
bt	-	-	set/list trace point(s)	[-p/-v] [symb/addr [script] [cond]]
ct	-	-	clear trace point	slot [-p/-v] symb/addr
cat	-	-	clear all trace points	-

Breakpoints/Steps Subcommands

Subcommand	Alias	Alias	Function	Argument
b	brk	-	set/list break point(s)	[-p/-v] [symb/addr]
lb	lbrk	-	set/list local bp(s)	[-p/-v] [symb/addr]
c	cl	-	clear break point	[slot [-p/-v] symb/addr]
lc	lcl	-	clear local bp	[slot [-p/-v] symb/addr [ctx]]
ca	-	-	clear all break points	-
r	return	-	go to end of function	-
gt	-	-	go until address	[-p/-v] symb/addr
n	nexti	-	next instruction	[count]
s	stepi	ste	single step	[count]
S	-	-	step on bl/blr	[count]
B	-	-	step on branch	[count]

Dumps/Display/Decode Subcommands

Subcommand	Alias	Alias	Function	Argument
d	dump	-	display byte data	symb/eaddr [count]
dw	-	-	display word data	symb/eaddr [count]
dd	-	-	display double word data	symb/eaddr [count]
dp	-	-	display byte data	phys. addr [count]
dpw	-	-	display word data	phys. addr [count]
dpc	-	-	display double word data	phys. addr [count]
dc	dis	-	display code	symb/eaddr [count]
dpc	-	-	display code	phys. addr [count]
dr	-	-	display registers	[gp sr sp >reg nam>]
ddvb	diob	-	display device byte	dev eaddr [count]
ddvh	dioh	-	display device half word	dev eaddr [count]
ddvw	diow	-	display device word	dev eaddr [count]
ddvd	diod	-	display device double word	dev eaddr [count]
ddpb	-	-	display device byte	dev paddr [count]
ddph	-	-	display device half word	dev paddr [count]
ddpw	-	-	display device word	dev paddr [count]
ddpd	-	-	display device double word	dev paddr [count]
find	-	-	find pattern	[-s] symb/eaddr pattern [mask [delta]]

Subcommand	Alias	Alias	Function	Argument
findp	-	-	find pattern	[-s] phys. addr pattern [mask [delta]]
ext	-	-	extract pattern	[-p] symb/eaddr delta [size [count]]
extp	-	-	extract pattern	[-p] phys. addr delta [size [count]]

Modify Memory Subcommands

Subcommand	Alias	Alias	Function	Argument
m	-	-	modify sequential bytes	symb/eaddr
mw	-	-	modify sequential word	symb/eaddr
md	-	-	modify sequential double word	symb/eaddr
mp	-	-	modify sequential bytes	phys. addr
mpw	-	-	modify sequential word	phys. addr
mpd	-	-	modify sequential double word	phys. addr
mr	-	-	modify registers	[gp sr sp >reg nam>]
mdvb	miob	-	modify device byte	dev eaddr
mdvh	mioh	-	modify device half	dev eaddr
mdvw	miow	-	modify device word	dev eaddr
mdvd	miod	-	modify device double word	dev eaddr
mdpb	-	-	modify device byte	dev paddr
mdph	-	-	modify device half	dev paddr
mdpw	-	-	modify device word	dev paddr
mdpd	-	-	modify device double word	dev paddr

Namelist/Symbol Subcommands

Subcommand	Alias	Alias	Function	Argument
nm	-	-	translate symbol to eaddr	symb
ns	-	-	no symbol mode (toggle)	-
ts	-	-	translate eaddr to symbol	eaddr

Watch Break Point Subcommands

Subcommand	Alias	Alias	Function	Argument
wr	stop-r	-	stop on read data	[[<i>-p/-v</i>] symb/addr [size]]
ww	stop-w	-	stop on write data	[[<i>-p/-v</i>] symb/addr [size]]
wrw	stop-rw	-	stop on r/w data	[[<i>-p/-v</i>] symb/addr [size]]
cw	stop-cl	-	clear watch	-
lwr	lstop-r	-	local stop on read data	[[<i>-p/-v</i>] symb/addr [size]]
lww	lstop-w	-	local stop on write data	[[<i>-p/-v</i>] symb/addr [size]]
lwrw	lstop-rw	-	local stop on r/w data	[[<i>-p/-v</i>] symb/addr [size]]
lcw	lstop-cl	-	clear local watch	-

Miscellaneous Subcommands

Subcommand	Alias	Alias	Function	Argument
time	-	-	display elapsed time	-
debug	-	-	enable/disable debug	[?]

Conditional Subcommands

Subcommand	Alias	Alias	Function	Argument
test	[-	bt condition	[symb/eaddr == symb/eaddr]
test	[-	bt condition	[symb/eaddr != symb/eaddr]
test	[-	bt condition	[symb/eaddr >= symb/eaddr]
test	[-	bt condition	[symb/eaddr <= symb/eaddr]
test	[-	bt condition	[symb/eaddr > symb/eaddr]
test	[-	bt condition	[symb/eaddr < symb/eaddr]
test	[-	bt condition	[symb/eaddr ^ symb/eaddr]
test	[-	bt condition	[symb/eaddr & symb/eaddr]
test	[-	bt condition	[symb/eaddr symb/eaddr]

Calculator Converter Subcommands

Subcommand	Alias	Alias	Function	Argument
hcal	cal	-	calc/conv a hexa expr	hexa expression
dcal (“hcal and dcal Subcommands” on page 439)	-	-	calc/conv a decimal expr	decimal expression

Machine Status Subcommands

Subcommand	Alias	Alias	Function	Argument
stat	-	-	system status message	-
switch	sw	-	switch thread	[[th {slot/eaddr} {u/k}]]

Kernel Extension Loader Subcommands

Subcommand	Alias	Alias	Function	Argument
lke	-	-	list loaded extensions	[?][-1][slot symb/eaddr]
stbl	-	-	list loaded symbol tables	[slot symb/eaddr]
rmst	-	-	remove symbol table	slot symb/eaddr
exp	-	-	list export tables	[symb]

Address Translation Subcommands

Subcommand	Alias	Alias	Function	Argument
tr	-	-	translate to real address	symb/eaddr
tv	-	-	display MMU translation	symb/eaddr

Process Subcommands

Subcommand	Alias	Alias	Function	Argument
ppda	-	-	Display per processor data area	[*/cpunb/symb/eaddr]
intr	-	-	@Display int handler	[slot/symb/eaddr]
mst	-	-	Display mst area	[slot] [[-a] symb/eaddr]
p	proc	-	Display proc table	[*/slot/symb/eaddr]
th	thread	-	Display thread table	[*/slot/symb/eaddr/-w ?]
ttid	th_tid	-	Display thread tid	[tid]
tpid	th_pid	-	Display thread pid	[pid]
rq	runq	-	Display run queues	[bucket/symb/eaddr]
sq	sleepq	-	Display sleep queues	[bucket/symb/eaddr]

Subcommand	Alias	Alias	Function	Argument
lq	lockq	-	Display lock queues	[bucket/symb/eaddr]
u	user	-	Display u_area	[-?][slot/symb/eaddr]

LVM Subcommands

Subcommand	Alias	Alias	Function	Argument
pbuf	pbuf	-	Display physical buf	[*] symb/eaddr
volgrp	volgrp	-	Display volume group	symb/eaddr
pvol	pvol	-	Display physical vol	symb/eaddr
lvol	lvol	-	Display logical vol	symb/eaddr

SCSI Subcommands

Subcommand	Alias	Alias	Function	Argument
asc	ascsi	-	Display ascsi	[slot/symb/eaddr]
vsc	vscsi	-	Display vscsi	[slot/symb/eaddr]
scd	scdisk	-	Display scdisk	[slot/symb/eaddr]

Memory Allocator Subcommands

Subcommand	Alias	Alias	Function	Argument
hp	heap	-	Display kernel heap	[symb/eaddr]
xm	xmalloc	-	Display heap debug	[-?]
kmbucket	bucket	-	Display kmembuckets	[?][-l][-c n][-i n][adr]
kmstats	-	-	Display kmemstats	[symb/eaddr]

File System Subcommands

Sub- command	Alias	Alias	Function	Argument
buf	buffer	-	Display buffer	[slot/symb/eaddr]
hb	hbuffer	-	Display buffehash	[bucket/symb/eaddr]
fb	fbuffer	-	Display freelist	[bucket/symb/eaddr]
gno	gnode	-	Display gnode	symb/eaddr
gfs	-	-	Display gfs	symb/eaddr
file	-	-	Display file	[slot/symb/eaddr]
ino	inode	-	Display inode	[slot/symb/eaddr]
hino	hinode	-	Display inodehash	[bucket/symb/eaddr]
fino	icache	-	Display icache list	[slot/symb/eaddr]
rno	rnode	-	Display rnode	symb/eaddr
cku	cku_priv	-	Display cku private	symb/eaddr
vno	vnode	-	Display vnode	symb/eaddr
vfs	mount	-	Display vfs	[slot/slot/symb/eaddr]
specno	specnode	-	Display specnode	symb/eaddr

Sub- command	Alias	Alias	Function	Argument
devno	devnode	-	Display devnode	[slot/symb/eaddr]
fifono	fifonode	-	Display fifonode	slot/symb/eaddr]
hno	hnode	-	isplay hnodehash	[bucket/symb/eaddr]

System Table Subcommands

Sub- command	Alias	Alias	Function	Argument
var	-	-	Display var	-
dev	devsw	-	Display devsw table	[symb/address/major]
trb	timer	-	Display system timer request blocks	-
slk	spl	-	Display simple lock	[symb/eaddr]
clk	cpl	-	Display complex lock	[symb/eaddr]
ipl	iplcb	-	Display ipl proc info	[*/cpu index]
trace	-	-	Display trace buffer	[?]

Net Subcommands

Sub- command	Alias	Alias	Function	Argument
ifnet	-	-	Display interface	[slot/symb/eaddr]
tcb	-	-	Display TCBS	[slot/symb/eaddr]
udb	-	-	Display UDBs	[slot/symb/eaddr]
sock	-	-	Display socket	[tcp/udp] [symb/eaddr]
tcpcb	-	-	Display TCP CB	[tcp/udp] [symb/eaddr]
mbuf	-	-	Display mbuf	[tcp/udp] [symb/eaddr]

VMM Subcommands

Sub- command	Alias	Alias	Function	Argument
vmker	-	-	VMM kernel segment data	-
rmap	-	-	VMM RMAP	[*][slot]
pfhdata	-	-	VMM control variables	-
vmstat	-	-	VMM statistics	-
vmaddr	-	-	VMM Addresses	-
pdt	-	-	VMM paging device table	[*][slot]
scb	-	-	VMM segment control blocks	[selection]
pft	-	-	VMM PFT entries	[selection]
pte	-	-	VMM PTE entries	[selection]
pta	-	-	VMM PTA segment	[?]
ste	-	-	VMM STAB	[-p pid]
sr64	-	-	VMM SEG REG	[-p pid] [esid] [size]
segst64	-	-	VMM SEGSTATE	[-p pid][[-e esid][[-s flag] [fno shm]]]

Sub- command	Alias	Alias	Function	Argument
apt	-	-	VMM APT entries	[selection]
vmwait	-	-	VMM wait status	[symb/eaddr]
ames	-	-	VMM address map entries	[symb/eaddr]
zproc	-	-	VMM zeroing kproc	-
vmlog	-	-	VMM error log	-
vrlid	-	-	VMM reload xlate table	-
ipc	-	-	IPC information	-
lka	lockanch	tblk	VMM lock anchor/tblock	[slot/symb/eaddr]
lkh	lockhash	-	VMM lock hash	[slot/symb/eaddr]
lkw	lockword	-	VMM lock word	[slot/symb/eaddr]
vmdmap	-	-	VMM disk map	[slot/symb/eaddr]
vmlocks	vmlock	vl	VMM spin locks	-

SMP Subcommands

Sub- command	Alias	Alias	Function	Argument
start	-	-	Start cpu	cpu number all
stop	-	-	Stop cpu	cpu number all
cpu	-	-	Switch to cpu	[cpu number any]

bat/Block Address Translation Subcommands

Sub- command	Alias	Alias	Function	Argument
dbat	-	-	display dbats	[index]
ibat	-	-	display ibats	[index]
mdbat	-	-	modify dbats	[index]
mibat	-	-	modify ibats	[index]

btac/BRAT Subcommands

Sub- command	Alias	Alias	Function	Argument
btac	-	-	branch target	[-p/-v] [symb/eaddr]
cbtac	-	-	clear branch target	-
lbtac	-	-	local branch target	[-p/-v] [symb/eaddr]
lcbtac	-	-	clear local br target	-

machdep Subcommand

Sub- command	Alias	Alias	Function	Argument
reboot	kill	-	reboot the machine	-

Basic Subcommands for the KDB Kernel Debugger and kdb Command

h Subcommand

Display the list of valid subcommands. The **help** subcommand can be reduced at only one topic. The actual list of topics is:

- basic subcommands [exit-setup-stack frame] (see “Basic Subcommands” on page 402)
- trace break point subcommands [break and continue] (see “Trace Subcommands” on page 402)
- break points/steps subcommands [break and prompt] (see “Breakpoints/Steps Subcommands” on page 403)
- dumps/display/decode/search subcommands [show memory-registers] (see “Dumps/Display/Decode Subcommands” on page 403)
- modify memory subcommands [alter memory-registers] (see “Modify Memory Subcommands” on page 404)
- namelists/symbols subcommands [symbol name<->address] (see “Namelist/Symbol Subcommands” on page 404)
- watch subcommands [data break point] (see “Watch Break Point Subcommands” on page 405)
- miscellaneous subcommands [internal KDB debug features] (see “Miscellaneous Subcommands” on page 405)
- conditionnal subcommands [how to set conditional break point] (see “Conditional Subcommands” on page 405)
- calculator converter subcommands [hex<->dec] (see “Calculator Converter Subcommands” on page 406)
- machine status subcommands [status-thread switching] (see “Machine Status Subcommands” on page 406)
- loader subcommands [show kernel extension-export table] (see “Kernel Extension Loader Subcommands” on page 406)
- address translation subcommands [V to R mapping] (see “Address Translation Subcommands” on page 406)
- process subcommands [processor-interrupt-process-thread] (see “Process Subcommands” on page 406)
- lvm subcommands [show logical volume manager info] (see “LVM Subcommands” on page 407)
- scsi subcommands [show disk driver queues (see “SCSI Subcommands” on page 407)]
- memory allocator subcommands [kernel heap-kmem bucket] (see “Memory Allocator Subcommands” on page 407)
- file system subcommands [buffer-kernel heap-LFS-VFS-SPECFS] (see “File System Subcommands” on page 407)
- system table subcommands [timer-lock-trace hooks-] (see “System Table Subcommands” on page 408)
- net subcommands [ifnet-tcb-udb-socket-mbuf] (see “Net Subcommands” on page 408)
- vmm subcommands [segment-page-paging device-disk map...] (see “VMM Subcommands” on page 408)
- SMP subcommands [start-stop-CPU status] (see “SMP Subcommands” on page 409)

- bat/Block Address Translation subcommands [show-alter BAT register] (see “bat/Block Address Translation Subcommands” on page 409)
- btac/BRAT subcommands [branch break point] (see “btac/BRAT Subcommands” on page 409)
- machdep subcommands [reboot] (see “machdep Subcommand” on page 409)

Example

KDB(0)> ? ?

help topics:

```

basic subcommands
trace subcommands
break points/steps
dumps/display/decode
modify memory
namelists/symbols
kdbx subcommands (dont use directly)
watch subcommands
conditionnal subcommand
calculator converter
machine status
loader subcommands
address translation
system table
net subcommands
vmm subcommands
trampolin subcommands
SMP subcommands
bat/Block Address Translation
btac/BRAT subcommand
machdep subcommands

```

KDB(7)> ? step

CMD	ALIAS	ALIAS	FUNCTION	ARG
			*** break points/steps ***	
b	brk		set/list break point(s)	[-p/-v] [addr]
lb	lbrk		set/list local bp(s)	[-p/-v] [addr]
c	cl		clear break point	[slot [-p/-v] addr]
lc	lcl		clear local bp	[slot [-p/-v] addr [ctx]]
ca			clear all break points	
r	return		go to end of function	
gt			go until address	[-p/-v] addr
n	nexti		next instruction	[count]
s	stepi		single step	[count]
S			step on bl/blr	
B			step on branch	

his Subcommand

The print history subcommand prints history subcommands. Argument can specify a number of subcommands. Each subcommand can be edited and executed with usual control characters (ala emacs).

Example

KDB(3)> his ?

Usage: hist [line count]

```

..... CTRL_A go to beginning of the line
..... CTRL_B one char backward
..... CTRL_D delete one char
..... CTRL_E go to end of line
..... CTRL_F one char forward
..... CTRL_N next command
..... CTRL_P previous command
..... CTRL_U kill line

```

KDB(3)> his

tpid

```

f
s 11
r
n 11
p proc+001680
c
dc .kforkx+30 11
mw .kforkx+000040
48005402
.
his ?
KDB(3)>

```

e Subcommand

The exit subcommand leaves KDB session and returns to the system, all breakpoints are installed in memory. To leave KDB without breakpoints, the **clear all** subcommand must be invoked.

The optional **dump** argument is only applicable to the KDB kernel debugger.

The **dump** argument can be specified to force an AIX dump. The method used to force a dump depends on how the debugger was invoked.

- **panic**: If the debugger was invoked by the panic call, force the dump by entering `q dump`. If another processor enters KDB after that (for example: spin-lock timeout), exit the debugger.
- **halt_display**: If the debugger was invoked by a halt display (C20 on the LED), enter `q`.
- **Soft_reset**: If the debugger was invoked by a soft reset (pressing the reset button on the server), first move the key on the server. If the key was in the SERVICE position at boot time, set it to NORMAL; otherwise, set the key to SERVICE.

Note: Forcing a dump with this method *requires* that you know what the key position was at boot time.

Then enter `quit` once for each cpu.

- Break in by `^ \` : You cannot create a dump if you invoke the debugger with the break method.

When the dump is in progress, `_0c9` displays on the LEDs while the dump is copied on disk (either on hd7 or hd6). If you entered the debugger through a panic call, then control is returned to the debugger when the dump is over, and LEDs show `xxxx`. If you entered the debugger through `halt_display`, the display shows: `888 102 700 0c0` when the dump is over.

set Subcommand

The setup subcommand may be used to list and set **kdb** toggles. Current list of toggles is:

- **no_symbol** to suppressed the symbol table management.
- **mst_wanted** to display all mst items in the stack trace subcommand, every time an interrupt is detected in the stack. To have shorter display, disable this toggle.
- **screen_size** can be set to change the integrated more window size.
- **power_pc_syntax** is used in the disassembler package to display old POWER or new POWER PC instruction mnemonics.

- **hardware_target** is also used in the disassembler package to detect invalid op-code on the specified target. Allowed targets are POWER 601, 603, 604, 620 (toggle value: 601, 603, 604, 620) and POWER RS1 RS2 (toggle value: 1, 2).
- **unix_symbol_start_from** is the lowest effective address from which symbol search is started. To force other values to be displayed in hexadecimal, set this one.
- **hexadecimal_wanted** applies to thread and process subcommand. It is possible to have information in decimal.
- **screen_previous** applies to display subcommand. When it is true, the display subcommand continues (when typing enter) with decreasing addresses.
- **display_stack_frames** applies to stack display subcommand. When it is true, the stack display subcommand prints a part of the stack in binary mode.
- **display_stacked_regs** applies to stack display subcommand. When it is true, the stack display subcommand prints register values saved in the stack.
- **64_bit** is used to print 64-bit registers on 64-bit architecture. By default only 32-bit formats are printed.
- **Thread/Cpu attached local breakpoint** (see corresponding part of the document) can be attached to the current thread or the current processor. Choose what you need. By default, on POWER RS1 it is cpu local, and on POWER PC it is thread local.
- **Emacs window** applies to the display environment. Under emacs session, set it true to suppress extra line feeds.
- **KDB stops all processors** is the default behaviour. When KDB is called (from break points, panic, control X ...) all processors are stopped, looping until KDB session is ended. It is possible to let other processors running during a KDB session by resetting this toggle.
- **tweq_r1_r1** indicates that LLDB static break-points are caught by KDB. Set it to false and LLDB will be invoked.
- **kext_IF_active** should be set to enable kernel extension subcommands. By default all subcommands registered by kernel extensions are not parsed.
- **trace_back_lookup** should be set to process trace back information on user code (text or shared-lib) and kernext code. It can be used to see function names. By default it is not set.
- **origin** ala LLDB. Sets the origin variable to the value of the specified expression. Origins are used to match addresses with assembly language listings (which express addresses as offsets from the start of the file).

Toggles **display_stack_frames** and **display_stacked_regs** can be used to find arguments of routines. Arguments are saved in non-volatile registers or in the current stack. It is an easy way to look for them.

Example

```
KDB(1)> set
No toggle name          current value
 1 no_symbol             false
 2 mst_wanted            true
 3 screen_size           24
 4 power_pc_syntax       true
 5 hardware_target       604
 6 Unix symbols start from 3500
 7 hexadecimal_wanted    true
 8 screen_previous       false
 9 display_stack_frames  false
10 display_stacked_regs  false
11 64_bit                false
```

```

12 emacs_window          false
13 Thread attached local breakpoint
14 KDB stops all processors
15 tweq_r1_r1            true
16 kext_IF_active        true
17 kext_IF_active        false
18 origin                00000000
KDB(1)> dw 000034CC display memory
000034CC: 00000002 00000008 00010006 00000020
KDB(1)> set 6 1000 toggle change
Unix symbols start from 1000
KDB(1)> dw 000034CC display memory
_system_configuration+000000: 00000002 00000008 00010006 00000020
KDB(4)> sw 464
Switch to thread: <thread+015C00>
KDB(4)> sw u to see user code
KDB(4)> dc 1000A14C
1000A14C      bl      <1000A1A4>
KDB(4)> set 17
trace_back_lookup is true
KDB(4)> dc 1000A14C
.get_superblk+00007C      bl      <.validate_super>
KDB(0)> set origin 002C5338
origin = 002C5338
KDB(0)> b init_heap1
.init_heap1+000000 (real address:002C55F4) permanent & global
KDB(0)> e
Breakpoint
.init_heap1+000000 (ORG+000002BC)      stmw      r24,FFFFFFE0(stkp) <.mainstk+001EB8>
                                     r24=00003A60,FFFFFFE0(stkp)=00384B74
KDB(0)>
In the listing you can see ...
    | 000000                                PDEF      init_heap1
    | 0 |                                PROC      heap_addr,numpages,flags,heapx,pages,gr3-gr8
    | 0 | 0002BC stm      BF01FFE0  8      STM      #stack(gr1,-32)=gr24-gr31
...

```

f Subcommand

The **stack** subcommand displays all the stacks from the current instruction as deep as possible. Interrupts and system calls are crossed and the user stack is also seen. In the user space, trace back allows to have symbolic names. But KDB can not access directly these symbols. Use the **+x** toggle to have binary addresses (by example, to put a break point on one of these addresses). This toggle is valid until **-x** is entered with another **stack** subcommand To have shorter display, you can suppressed the mst display with the corresponding toggle **mst_wanted**. To have longer display, you can set the corresponding toggle **display_stack_wanted**. It is possible to display the stack of a specified thread (by its slot entry or address).

If running code is **-O** compiled, routine parameters are not saved in the stack. KDB warns about that with **[??]** at the end of the line. In this case, showed parameters may be wrong.

Example

In the following example, we set a break point on **v_gettlock**, and when the break point is encountered, the stack is displayed. Then we try to display the first argument of the **open()** syscall. Looking at the code, we can see that argument is saved by **copen()** in register R31, and this register is saved in the stack by **openpath()**. Looking at memory pointed by register R31, argument is found: **/dev/ptc**

```

KDB(2)> f show the stack
thread+012540 STACK:
[0004AC84]v_gettlock+000000 (00012049, C0011E80, 00000080, 00000000 [??])

```



```

[00085C18]v_pregetlock+0000B4 (??, ??, ??, ??)
[000132E8]isync_vcs1+0000D8 (??, ??)
_____ Exception (2FF3B400) _____
[000131FC].backt+000000 (00012049, C0011E80 [??])
[0004B220]vm_gettlock+000020 (??, ??)
[0019A64C]iwrite+00013C (??)
[0019D194]finicom+0000A0 (??, ??)
[0019D4F0]comlist+0001CC (??, ??)
[0019D5BC]_commit+000030 (00000000, 00000001, 09C6E9E8, 399028AA,
0000A46F, 0000E2AA, 2D3A4EAA, 2FF3A730)
[001E1B18]jfs_setattr+000258 (??, ??, ??, ??, ??, ??)
[001A5ED4]vnode_setattr+000018 (??, ??, ??, ??, ??, ??)
[001E9008]spec_setattr+00017C (??, ??, ??, ??, ??, ??)
[001A5ED4]vnode_setattr+000018 (??, ??, ??, ??, ??, ??)
[01B655C8]pty_vsetattr+00002C (??, ??, ??, ??, ??, ??)
[01B6584C]pty_setname+000084 (??, ??, ??, ??, ??, ??)
[01B60810]pty_create_ptp+0002C4 (??, ??, ??, ??, ??, ??)
[01B60210]pty_open_comm+00015C (??, ??, ??, ??)
[01B5FFC0]call_pty_open_comm+0000B8 (??, ??, ??, ??)
[01B6526C]ptm_open+000140 (??, ??, ??, ??, ??)
(2)> more (^C to quit) ?
[01A9A124]open_wrapper+0000D0 (??)
[01A8DF74]csq_protect+000258 (??, ??, ??, ??, ??, ??)
[01A96348]osr_open+0000BC (??)
[01A9C1C8]pse_clone_open+000164 (??, ??, ??, ??)
[001ADCC8]spec_clone+000178 (??, ??, ??, ??, ??)
[001B3FC4]openpnp+0003AC (??, ??, ??, ??, ??)
[001B4178]openpath+000064 (??, ??, ??, ??, ??, ??)
[001B43E8]copen+000130 (??, ??, ??, ??, ??)
[001B44BC]open+000014 (??, ??, ??)
[000037D8].sys_call+000000 ()
[10002E74]doit+00003C (??, ??, ??)
[10003924]main+0004CC (??, ??)
[1000014C]_start+00004C ()
KDB(2)> set l0 show saved registers
display_stacked_regs is true
KDB(2)> f show the stack
thread+012540 STACK:
[0004AC84]v_gettlock+000000 (00012049, C0011E80, 00000080, 00000000 [??])
...
[001B3FC4]openpnp+0003AC (??, ??, ??, ??, ??)
r24 : 2FF3B6E0 r25 : 2FF3B400 r26 : 10002E78 r27 : 00000000 r28 : 00000002
r29 : 2FF3B3C0 r30 : 00000000 r31 : 20000510
[001B4178]openpath+000064 (??, ??, ??, ??, ??, ??)
[001B43E8]copen+000130 (??, ??, ??, ??, ??)
r27 : 2A22A424 r28 : E3014000 r29 : E6012540 r30 : 0C87B000 r31 : 00000000
[001B44BC]open+000014 (??, ??, ??)
...
KDB(2)> dc open 6 look for argument R3
.open+000000 stwu stkp,FFFFFFC0(stkp)
.open+000004 mflr r0
.open+000008 addic r7,stkp,38
.open+00000C stw r0,48(stkp)
.open+000010 li r6,0
.open+000014 bl <.copen>
KDB(2)> dc copen 9 look for argument R3
.copen+000000 stmw r27,FFFFFFEC(stkp)
.copen+000004 addi r28,r4,0
.copen+000008 mflr r0
.copen+00000C lwz r4,D5C(toc) D5C(toc)=audit_flag
.copen+000010 stw r0,8(stkp)
.copen+000014 stwu stkp,FFFFFFA0(stkp)
.copen+000018 cmpi cr0,r4,0
.copen+00001C mtrcrf cr5,r28
.copen+000020 addi r31,r3,0
KDB(2)> d 20000510 display memory location @R31
20000510: 2F64 6576 2F70 7463 0000 0000 416C 6C20 /dev/ptc....All

```

In the following example, the problem is to find what is `lsfs` subcommand waiting for. The answer is given with `getfssize` arguments, and there are saved in the stack.

```
# ps -ef|grep lsfs
  root 63046 39258  0  Apr 01 pts/1  0:00 lsfs
# kdb
Preserving 587377 bytes of symbol table
First symbol sys_resource
PFT:
id.....0007
raddr.....01000000 eaddr.....B0000000
size.....01000000 align.....01000000
valid..1 ros....0 holes..0 io.....0 seg....0 wimg...2
PVT:
id.....0008
raddr.....003BC000 eaddr.....B2000000
size.....001FFDA0 align.....00001000
valid..1 ros....0 holes..0 io.....0 seg....0 wimg...2
(0)> dcal 63046 print hexa value of PID
Value decimal: 63046      Value hexa: 0000F646
(0)> tpid 0000F646 show threads of this PID
      SLOT NAME      STATE  TID PRI CPUID CPU FLAGS  WCHAN
thread+025440 795 lsfs  SLEEP 31B31 03C  000 00000004 057DB5BC
(0)> sw 795 set current context on this thread
Switch to thread: <thread+025440>
(0)> f show the stack
thread+025440 STACK:
[000205C0]e_block_thread+000250 ()
[00020B1C]e_sleep_thread+000040 (??, ??, ??)
[0002AAA0]iowait+00004C (??)
[0002B40C]bread+0000DC (??, ??)
[0020AF4C]readblk+0000AC (??, ??, ??, ??)
[001E90D8]spec_rdw+00007C (??, ??, ??, ??, ??, ??, ??, ??)
[001A6328]vnop_rdw+000070 (??, ??, ??, ??, ??, ??, ??, ??)
[00198278]rwuio+0000CC (??, ??, ??, ??, ??, ??, ??, ??)
[001986AC]rdwr+000184 (??, ??, ??, ??, ??, ??)
[001984D4]kreadv+000064 (??, ??, ??, ??)
[000037D8].sys_call+000000 ()
[D0046A18]read+000028 (??, ??, ??)
[1000A0E4]get_superblk+000054 (??, ??, ??)
[100035F8]read_super+000024 (??, ??, ??, ??)
[10005C00]getfssize+0000A0 (??, ??, ??)
[10002D18]prnt_stanza+0001E8 (??, ??, ??)
[1000349C]do_ls+000294 (??, ??)
[10000524]main+0001E8 (??, ??)
[1000014C]__start+00004C ()
(0)> sw u enable user context of the thread
(0)> dc 10005C00-a0 8 look for arguments R3, R4, R5
10005B60 mflr r0
10005B64 stw r31,FFFFFFFC(stkp)
10005B68 stw r0,8(stkp)
10005B6C stwu stkp,FFFFFFE0(stkp)
10005B70 stw r3,108(stkp)
10005B74 stw r4,104(stkp)
10005B78 stw r5,10C(stkp)
10005B7C addi r3,r4,0
(0)> set 9 print stack frame
display_stack_frames is true
(0)> f show the stack
thread+025440 STACK:
[000205C0]e_block_thread+000250 ()
...
[100035F8]read_super+000024 (??, ??, ??, ??)
=====
2FF225D0: 2FF2 26F0 2A20 2429 1000 5C04 F071 71C0 /.&.* $)..\.qq.
2FF225E0: 2FF2 2620 2000 4D74 D000 4E18 F071 F83C /.& .Mt..N..q. dw
```

```

    2FF225D0+104 print arguments (offset 0x104 0x108 0x10c)
2FF226D4: 2000DCC8 2000DC78 00000000 00000004
(0)> d 2000DC78 20 print first argument
2000DC78: 2F74 6D70 2F73 7472 6970 655F 6673 2E32 /tmp/stripes_fs.2
2000DC88: 3433 3632 0000 0000 0000 0000 0000 0004 4362.....
(0)> d 2000DCC8 20 print second argument
2000DCC8: 2F64 6576 2F73 6C76 3234 3336 3200 0000 /dev/s1v24362...
2000DCD8: 0000 0000 0000 0000 0000 0000 0000 0004 .....
(0)> q leave debugger
#

```

ctx Subcommand

Note: This subcommand is only available within the **kdb** command; it is not included in the KDB Kernel Debugger.

The **context** subcommand is used in user mode to analyse memory dump. By default, KDB shows the current AIX context. But it is possible to elect the current kernel KDB context, and to see more information in stack trace subcommand. For instance, the complete stack of a kernel panic is seen. Cpu number is given in argument. No argument allows to restore initial context.

Note: KDB context is available only if the running kernel is booted with KDB.

Example

```

$ kdb dump unix dump analysis
Preserving 628325 bytes of symbol table
First symbol sys_resource
Component Names:
 1) proc
 2) thrd
 3) errlg
 4) bos
 5) vmm
 6) bscsi
 7) scdisk
 8) lvm
 9) tty
10) netstat
11) lent_dd
PFT:
id.....0007
raddr.....0000000001000000 eaddr.....0000000001000000
size.....00800000 align.....00800000
valid..1 ros....0 holes..0 io.....0 seg....1 wimg...2
PVT:
id.....0008
raddr.....00000000004B8000 eaddr.....00000000004B8000
size.....000FFD60 align.....00001000
valid..1 ros....0 holes..0 io.....0 seg....1 wimg...2
Dump analysis on POWER_PC POWER_604 machine with 8 cpu(s)
Processing symbol table...
.....done
(0)> stat machine status
RS6K_SMP_MCA POWER_PC POWER_604 machine with 8 cpu(s)
..... SYSTEM STATUS
sysname... AIX          nodename.. jumbo32
release... 3           version... 4
machine... 00920312A0 nid..... 920312A0
time of crash: Tue Jul 22 09:46:22 1997
age of system: 1 day, 0 min., 35 sec.
..... PANIC STRING
assert(v_lookup(sid,pno) == -1)
..... SYSTEM MESSAGES

```

```

AIX Version 4.3
Starting physical processor #1 as logical #1... done.
Starting physical processor #2 as logical #2... done.
Starting physical processor #3 as logical #3... done.
Starting physical processor #4 as logical #4... done.
Starting physical processor #5 as logical #5... done.
Starting physical processor #6 as logical #6... done.
Starting physical processor #7 as logical #7... done.
[v_lists.c #727]
<- end_of_buffer
(0)> ctx 0 KDB context of CPU 0
Switch to KDB context of cpu 0
(0)> dr iar current instruction
iar : 00009414
.unlock_enable+000110      lwz      r0,8(stkp)          r0=0,8(stkp)=mststack+00AD18
(0)> ctx 1 KDB context of CPU 1
Switch to KDB context of cpu 1
(1)> dr iar current instruction
iar : 000BDB68
.kunlock1+000118      blr          <.ld_usecount+0005BC> r3=0000000B
(1)> ctx 2 KDB context of CPU 2
Switch to KDB context of cpu 2
(2)> dr iar current instruction
iar : 00027634
.tstart+000284      blr          <.sys_timer+000964> r3=00000005
(2)> ctx 3 KDB context of CPU 3
Switch to KDB context of cpu 3
(3)> dr iar current instruction
iar : 01B6A580
01B6A580      ori      r3,r31,0          <00000089> r3=50001000,r31=00000089
(3)> ctx 4 KDB context of CPU 4
Switch to KDB context of cpu 4
(4)> dr iar current instruction
iar : 00014BFC
.panic_trap+000004      bl      <.panic_dump>      r3=__$STATIC+000294
(4)> f current stack
__kdb_thread+0002F0 STACK:
[00014BFC].panic_trap+000004 ()
[0003ACAC]v_inspft+000104 (??, ??, ??)
[00048DA8]v_inherit+0004A0 (??, ??, ??)
[000A7ECC]v_preinherit+000058 (??, ??, ??)
[00027BFC]begbt_603_patch 2+000008 (??, ??)
Machine State Save Area [2FF3B400]
iar : 00027AEC msr : 000010B0 cr : 22222222 lr : 00243E58
ctr : 00000000 xer : 00000000 mq : 00000000
r0 : 000A7E74 r1 : 2FF3B220 r2 : 002EBC70 r3 : 00013350 r4 : 00000000
r5 : 00000100 r6 : 00009030 r7 : 2FF3B400 r8 : 00000106 r9 : 00000000
r10 : 00243E58 r11: 2FF3B400 r12 : 000010B0 r13 : 000C1C80 r14 : 2FF22A88
r15 : 20022DB8 r16 : 20006A98 r17 : 20033128 r18 : 00000000 r19 : 0008AD56
r20 : B02A6038 r21 : 0000006A r22 : 00000000 r23 : 0000FFFF r24 : 00000100
r25 : 00003262 r26 : 00000000 r27 : B02B8AEC r28 : B02A9F70 r29 : 00000001
r30 : 00003350 r31 : 00013350
s0 : 00000000 s1 : 007FFFFFFF s2 : 0000864B s3 : 007FFFFFFF s4 : 007FFFFFFF
s5 : 007FFFFFFF s6 : 007FFFFFFF s7 : 007FFFFFFF s8 : 007FFFFFFF s9 : 007FFFFFFF
s10 : 007FFFFFFF s11 : 00001001 s12 : 00002002 s13 : 6001F01F s14 : 00004004
s15 : 007FFFFFFF
prev      00000000 kjmpbuf 00000000 stackfix 00000000 intpri 0B
curid     0008AD56 sralloc E01E0000 ioalloc 00000000 backt 00
flags     00 tid      00000000 excp_type 00000000
fpscr     00000000 fpeu      01 fpinfo      00 fpscrx     00000000
o_iar     00000000 o_toc     00000000 o_arg1     00000000
excbranch 00000000 o_vaddr 00000000 mstext    00000000
Except :
csr 00000000 dsisr 40000000 bit set: DSISR_PFT
srval 6000864B dar 2FF22FF8 dsirr 00000106
[00027AEC].backt+000000 (00013350, 00000000 [??])
[00243E54]vms_delete+0004DC (??)

```

```

[00256838]shmfreews+0000B0 ()
[000732B4]freeuspace+000010 ()
[00072EAC]kexitx+000688 (??)
(4)> ctx    AIX context of CPU 4
Restore initial context
(4)> f current stack
thread+031920 STACK:
[00027AEC].backt+000000 (00013350, 00000000 [??])
[00243E54]vms_delete+0004DC (??)
[00256838]shmfreews+0000B0 ()
[000732B4]freeuspace+000010 ()
[00072EAC]kexitx+000688 (??)
(4)>

```

cdt Subcommand

Note: This subcommand is only available within the **kdb** command; it is not included in the KDB Kernel Debugger.

The **cdt** subcommand is used in user mode to analyse memory dump. Any part of component dump can be printed. Usage is `cdt [-d] [<index>] [<entry>]`.

- `-d` to ask for data dump
- index of the component in the table
- entry of the data area in the component

Example

```

(0)> cdt
1) CDT head name proc, len 001D80E8, entries 96676
2) CDT head name thrd, len 003ABE4C, entries 192489
3) CDT head name errlg, len 00000054, entries 3
4) CDT head name bos, len 00000040, entries 2
5) CDT head name vmm, len 000003D8, entries 30
6) CDT head name sscsidd, len 0000007C, entries 5
7) CDT head name dptSR, len 00000054, entries 3
8) CDT head name scdisk, len 00000130, entries 14
9) CDT head name lvm, len 00000040, entries 2
10) CDT head name SSAGS, len 000000A4, entries 7
11) CDT head name SSAES, len 00000054, entries 3
12) CDT head name ssagateway, len 0000007C, entries 5
13) CDT head name tty, len 00000068, entries 4
14) CDT head name sio_dd, len 00000054, entries 3
15) CDT head name netstat, len 000000E0, entries 10
16) CDT head name ent2104x, len 00000054, entries 3
17) CDT head name cstokdd, len 0000007C, entries 5
18) CDT head name atm_dd_charm, len 00000040, entries 2
19) CDT head name ssadisk, len 000002AC, entries 33
20) CDT head name SSADS, len 00000040, entries 2
21) CDT head name osi_frame, len 0000002C, entries 1
(0)> cdt 12
12) CDT head name ssagateway, len 0000007C, entries 5
CDT   1 name      HashTbl addr 0000000001A25CF0, len 00000040
CDT   2 name      CfgdAdap addr 0000000001A0E044, len 00000004
CDT   3 name      OpenAdap addr 0000000001A0E048, len 00000004
CDT   4 name      LockWord addr 0000000001A0E04C, len 00000004
CDT   5 name      ssa0 addr 0000000001A2D000, len 00000B88
(0)> cdt -d 12 4
12) CDT head name ssagateway, len 0000007C, entries 5
CDT   4 name      LockWord addr 0000000001A0E04C, len 00000004
01A0E04C: FFFFFFFF      ....

```

Trace Subcommands for the KDB Kernel Debugger and kdb Command

bt Subcommand

Note: This subcommand is only available within the **kdb** command; it is not included in the KDB Kernel Debugger.

Trace break point **bt** may be used to trace each execution occurrence of the specified address. Without argument, the break point table is printed. Adding break points, there is a maximum of 32 trace break points.

The segment id (sid) is always used to identify a break point, effective address could have multiple translations in several virtual spaces. **kdb** needs to reinstall the right instruction when execution is resumed. During this short time (one step if no interrupt) it is possible to miss the trace on others processors.

It is possible to specify if the address is physical or virtual with **-p** or **-v** option. By default KDB chooses the current state of the machine: if the subcommand is entered before VMM initialisation, the address is physical (real address), else virtual (effective address).

Example

```
KDB(0)> bt open enable trace on open()
KDB(0)> bt display current active traces
0:      .open+0000000 (sid:00000000) trace {hit: 0}
KDB(0)> e exit debugger
...
open+000000000 (2FF7FF2B, 00000000, DEADBEEF)
open+000000000 (2FF7FF2F, 00000000, DEADBEEF)
open+000000000 (2FF7FF33, 00000000, DEADBEEF)
open+000000000 (2FF7FF37, 00000000, DEADBEEF)
open+000000000 (2FF7FF3B, 00000000, DEADBEEF)
...
KDB(0)> bt display current active traces
0:      .open+0000000 (sid:00000000) trace {hit: 5}
KDB(0)>
```

ct and cat Subcommands

Note: This subcommand is only available within the **kdb** command; it is not included in the KDB Kernel Debugger.

To clear trace break point, **cat** subcommand erases all of them and **ct** subcommand erases only the specified trace (by slot number, symbol or address).

It is possible to specify if the address is physical or virtual with **-p** or **-v** option. By default KDB chooses the current state of the machine: if the subcommand is entered before VMM initialisation, the address is physical (real address), else virtual (effective address).

Note: Slot numbers are not fixed. To clear slot 1 and slot 2 enter **ct 2**; **ct 1** or **ct 1**; **ct 1**, do not enter **ct 1**; **ct 2**

Example

```
KDB(0)> bt open enable trace on open()
KDB(0)> bt close enable trace on close()
KDB(0)> bt readlink enable trace on readlink()
```

```

KDB(0)> bt display current active traces
0:      .open+000000 (sid:00000000) trace {hit: 0}
1:      .close+000000 (sid:00000000) trace {hit: 0}
2:      .readlink+000000 (sid:00000000) trace {hit: 0}
KDB(0)> ct 1 clear trace slot 1
KDB(0)> bt display current active traces
0:      .open+000000 (sid:00000000) trace {hit: 0}
1:      .readlink+000000 (sid:00000000) trace {hit: 0}
KDB(0)> cat clear all active traces
KDB(0)> bt display current active traces
No breakpoints are set.
KDB(0)>

```

bt script Subcommand

By default the top of the stack is displayed, but it is possible to specify a **script** of **kdb** subcommands. These subcommands are executed when the trace breakpoint is hit.

Example

Open routine is traced with a script asking to display **iar** and **lr** registers and to show what is pointed by **r3**, the first parameter. Here **open()** is called on "**sbin**" from **svc_flih()**.

```

KDB(0)> bt open "dr iar; dr lr; d @r3" enable trace on open()
KDB(0)> bt display current active traces
0:      .open+000000 (sid:00000000) trace {hit: 0} {script: dr iar; dr lr;d @r3}
KDB(0)> e exit debugger
iar : 001C5BA0
.open+000000 mflr r0 <.svc_flih+00011C>
lr : 00003B34
.svc_flih+00011C lwz toc,4108(0) toc=TOC,4108=g_toc
2FF7FF3F: 7362 696E 0074 6D70 0074 6F74 6F00 7500 sbin.tmp.toto.u.
KDB(0)> bt display current active traces
0:      .open+000000 (sid:00000000) trace {hit: 1} {script: dr iar; dr lr;d @r3}
KDB(0)> ct open clear trace on open
KDB(0)>

```

bt [cond] Subcommand

With a conditional subcommand, it is possible to stop inside **kdb** when the condition is true.

Example

This example shows how to trace and stop when a condition is true. Here we are waiting for **time** global data to be greater than the specified value, and 923 hits have been necessary to win.

```

KDB(0)> bt sys_timer "[ @time >= 2b8c8c00 ] " enable trace on sys_timer()
KDB(0)> e exit debugger
...
Enter kdb [ @time >= 2b8c8c00 ]
KDB(0)> bt display current active traces
0:      .sys_timer+000000 (sid:00000000) trace {hit: 923} {script: [ @time >= 2b8c8c00 ] }
KDB(0)> cat clear all traces

```

Breakpoints/Steps Subcommands for the KDB Kernel Debugger and kdb Command

b Subcommand

Note: This subcommand is only available within the **kdb** command; it is not included in the KDB Kernel Debugger.

The **b** subcommand sets permanent global break point in the code. With trace break points, there is a maximum of 32 break points. KDB checks that a valid instruction will be trapped and prints a warning message. If it does, the breakpoint should be removed, else memory can be corrupted (the break point has been installed).

It is possible to specify if the address is physical or virtual with **-p** or **-v** option. By default KDB chooses the current state of the machine: if the subcommand is entered before VMM initialisation, the address is physical (real address), else virtual (effective address). After VMM is setup, to set break-point in real-mode code that is not mapped V=R, **-p** must be used, else KDB expects a virtual address and translates the address.

Example before VMM setup

```
KDB(0)> b vsi set break point on vsi()
.vsi+000000 (real address:002AA5A4) permanent & global
KDB(0)> e exit debugger
...
Breakpoint
.vsi+000000      stmw   r29,FFFFFFF4(stkp) <.mainstk+001EFC>
                r29=isync_sc1+000040,FFFFFFF4(stkp)=.mainstk+001EFC
```

Example after VMM setup

```
KDB(0)> b display current active break points
No breakpoints are set.
KDB(0)> b 0 set break point at address 0
WARNING: break point at 00000000 on invalid instruction (00000000)
00000000 (sid:00000000) permanent & global
KDB(0)> c 0 remove break point at address 0
KDB(0)> b vmvcs set break point on vmvcs()
.vmvcs+000000 (sid:00000000) permanent & global
KDB(0)> b i_disable set break point on i_disable()
.i_disable+000000 (sid:00000000) permanent & global
KDB(0)> e exit debugger
...
Breakpoint
.i_disable+000000  mfmsr  r7                <start+001008> r7=DEADBEEF
KDB(0)> b display current active break points
0:      .vmvcs+000000 (sid:00000000) permanent & global
1:      .i_disable+000000 (sid:00000000) permanent & global
KDB(0)> c 1 remove break point slot 1
KDB(0)> b display current active break points
0:      .vmvcs+000000 (sid:00000000) permanent & global
KDB(0)> e exit debugger
...
Breakpoint
.vmvcs+000000     mflr   r10                <.initcom+000120>
KDB(0)> ca remove all break points
```

lb Subcommand

Note: This subcommand is only available within the **kdb** command; it is not included in the KDB Kernel Debugger.

The local break point subcommand sets permanent local break point in the code, according to the corresponding toggle (cpu or thread based). One context is associated to the local break point. Up to 8 different contexts are settable for each local break point. The context is the effective address of the thread entry, or the processor number. In the following example, we show that context is the effective

address of the **curthread**. Local break points are cleared context by context with the **lc** subcommand. By default the current context is cleared. The **c** or **ca** subcommand clears all contexts.

When the break point is hit with another context, KDB prints a warning. Using **switch** subcommand, it is possible to select another context, instead of the current one. After switching from the current context (thread or processor) to another one, the **lb** subcommand will apply on this new context.

It is possible to specify if the address is physical or virtual with **-p** or **-v** option. By default KDB chooses the current state of the machine: if the subcommand is entered before VMM initialisation, the address is physical (real address), else virtual (effective address). After VMM is setup, to set break-point in real-mode code that is not mapped V=R, **-p** must be used, else KDB expects a virtual address and translates the address.

Example

```
KDB(0)> b execv set break point on execv()
Assumed to be [External data]: 001F4200 execve
Ambiguous: [Ext func]
001F4200 .execve
.execve+000000 (sid:00000000) permanent & global
KDB(0)> e exit debugger
...
Breakpoint
.execve+000000 mflr r0 <.svc_flih+00011C>
KDB(0)> ppda print current processor data area
Per Processor Data Area [00086E40]
csa.....2FEE0000 mstack.....0037CDB0
fpowner.....00000000 curthread.....E60008C0
...
KDB(0)> lb kexit set local break point on kexit()
.kexit+000000 (sid:00000000) permanent & local
KDB(0)> b display current active break points
0: .execve+000000 (sid:00000000) permanent & global
1: .kexit+000000 (sid:00000000) permanent & local
KDB(0)> e exit debugger
...
Warning, breakpoint ignored (context mismatched):
.kexit+000000 mflr r0 <._exit+000020>
Breakpoint
.kexit+000000 mflr r0 <._exit+000020>
KDB(0)> ppda print current processor data area
Per Processor Data Area [00086E40]
csa.....2FEE0000 mstack.....0037CDB0
fpowner.....00000000 curthread.....E60008C0
...
KDB(0)> lc 1 thread+00008C0 remove local break point slot 1
```

c, lc, and ca Subcommands

Note: This subcommand is only available within the **kdb** command; it is not included in the KDB Kernel Debugger.

To clear break point, use one of these subcommands. **cat** subcommand erases all of them: global, local, permanent or not. **c** and **lc** subcommand erase only the specified break point (by slot number, symbol or address). With **lc** subcommand, by default first context is removed. If there are more than one context, it is possible to specify the context to be removed.

It is possible to specify if the address is physical or virtual with **-p** or **-v** option. By default KDB chooses the current state of the machine: if the subcommand is entered before VMM initialisation, the address is physical (real address), else virtual (effective address). After VMM is setup, to set break-point in real-mode code that is not mapped V=R, **-p** must be used, else KDB expects a virtual address and translates the address.

Note: Slot numbers are not fixed. To clear slot 1 and slot 2 enter ct 2; ct 1 or ct 1; ct 1, do not enter ct 1; ct 2

r and gt Subcommands

Note: This subcommand is only available within the **kdb** command; it is not included in the KDB Kernel Debugger.

Non permanent break point can be set by the go to address subcommands. In that case it is local break points, and when KDB is entered, this break point is cleared. The **r** subcommand sets a break point on the address found in the **lr** register. In SMP environment, it is possible to hit this break point on another processor, so it is important to have thread/process local break point.

The **gt** subcommand has the same effect but we precise the wanted address.

It is possible to specify if the address is physical or virtual with **-p** or **-v** option. By default KDB chooses the current state of the machine: if the subcommand is entered before VMM initialisation, the address is physical (real address), else virtual (effective address). After VMM is setup, to set break-point in real-mode code that is not mapped V=R, **-p** must be used, else KDB expects a virtual address and translates the address.

When the break point is hit with another context, KDB prints a warning.

Example

```
KDB(2)> b _input enable break point on _input()
_input+000000 (sid:00000000) permanent & global
KDB(2)> e exit debugger
...
Breakpoint
_input+000000 stmw r29,FFFFFFFF(stkp) <2FF3B1CC>
r29=0A4C6C20,FFFFFFFF(stkp)=2FF3B1CC
KDB(6)> f
thread+014580 STACK:
[0021632C] _input+000000 (0A4C6C20, 0571A808 [??])
[00263EF4] jfs_rele+0000B4 (??)
[00220B58] vno_rele+000018 (??)
[00232178] vno_close+000058 (??)
[002266C8] closef+0000C8 (??)
[0020C548] closefd+0000BC (??, ??)
[0020C70C] close+000174 (??)
[000037C4] .sys_call+000000 ()
[D000715C] fclose+00006C (??)
[10000580] 10000580+000000 ()
[10000174] __start+00004C ()
KDB(6)> r go to the end of the function
...
.jfs_rele+0000B8 b <.jfs_rele+00007C> r3=0
KDB(7)> e exit debugger
...
Breakpoint
_input+000000 stmw r29,FFFFFFFF(stkp) <2FF3B24C>
```

```

    r29=09D75BD0,FFFFFFF4(stkp)=2FF3B24C
KDB(3)> gt @lr go to the link register value
.jfs_rele+0000B8 (sid:00000000) step
...
.jfs_rele+0000B8      b   <.jfs_rele+00007C> r3=0
KDB(1)>

```

n s, S, and B Subcommands

Note: This subcommand is only available within the **kdb** command; it is not included in the KDB Kernel Debugger.

Step can be done by two subcommands. The **s** subcommand allows the processor to single step on the next instruction. The **n** subcommand causes step to skip over subroutine calls, and we can follow the main flow of control. A count may specify how many steps are executed before returning to KDB dialog. Enter **CR/LF** and KDB continues the last step subcommand.

The **S** subcommand single steps but stops only on **bl** and **br** instructions. With that, we can see every call and return of routines. A count is also used to specify how many times KDB continues before stopping, and **CR/LF** continues the last subcommand

On POWER RS1 machine, steps are implemented with non permanent local break points. On POWER PC machine, steps are implemented with the **SE** bit of the **msr** status register of the processor. This bit is automatically associated with the thread/process context and can migrate from one processor to another.

A too long step subcommand can be interrupted by typing **DEL** key. Every time KDB treats a step occurrence, **DEL** key is tested.

Be aware that when you single step a program, this makes an exception to the processor for each of the debugged program's instruction. One side-effect of exceptions is to break reservations (see POWER PC architecture). This is why *stcwx* *never* succeeds if any breakpoint occurred since the last *larwx*. The net effect is that lock and atomic routines are *not steppable*. If you do it anyway, the system does not break, but you loop in the lock routine. If that happens, just "return" from the lock routine to the caller, and if the lock is free, you can get it.

Be aware also that some instructions are broken by exceptions. For examples, **rfi**, move to-from **srr0 srr1**. KDB tries to prevent against that by printing a warning message.

Note that the **S** subcommand of KDB (which single-steps the program until the next sub-routine call/return) will silently and endlessly fail to go through the atomic/lock routines. To watch out for this, you will get the KDB prompt again with a warning message.

When you want to take control of a thread currently sleeping, it is possible to step in the context of this thread. To do that you just have to switch to the sleeping thread (with **sw** subcommand) and to type the **s** subcommand. The step is set inside the thread context, and when the thread runs again, the step break point occurs.

Example

```
KDB(1)> b .vno_close+00005C enable break point on vno_close+00005C
vno_close+00005C (sid:00000000) permanent & global
KDB(1)> e exit debugger
Breakpoint
.vno_close+00005C    lwz    r11,30(r4)           r11=0,30(r4)=xix_vops+000030
KDB(1)> s 10 single step 10 instructions
.vno_close+000060    lwz    r5,68(stkp)          r5=FFD00000,68(stkp)=2FF97DD0
.vno_close+000064    lwz    r4,0(r5)            r4=xix_vops,0(r5)=file+0000C0
.vno_close+000068    lwz    r5,14(r5)           r5=file+0000C0,14(r5)=file+0000D4
.vno_close+00006C    bl     <._ptrgl>           r3=05AB620C
._ptrgl+000000       lwz    r0,0(r11)           r0=.closef+0000F4,0(r11)=xix_close
._ptrgl+000004       stw    toc,14(stkp)        toc=TOC,14(stkp)=2FF97D7C
._ptrgl+000008       mtctr  r0                  <.xix_close+000000>
._ptrgl+00000C       lwz    toc,4(r11)          toc=TOC,4(r11)=xix_close+000004
._ptrgl+000010       lwz    r11,8(r11)          r11=xix_close,8(r11)=xix_close+000008
._ptrgl+000014       bcctr  <.xix_close>
KDB(1)> <CR/LF> repeat last single step command
.xix_close+000000    mflr  r0                   <.vno_close+000070>
.xix_close+000004    stw    r31,FFFFFFFC(stkp)  r31=_vno_fops$$,FFFFFFFC(stkp)=2FF97D64
.xix_close+000008    stw    r0,8(stkp)          r0=.vno_close+000070,8(stkp)=2FF97D70
.xix_close+00000C    stwu   stkp,FFFFFFA0(stkp) stkp=2FF97D68,FFFFFFA0(stkp)=2FF97D08
.xix_close+000010    lwz    r31,12B8(toc)       r31=_vno_fops$$,12B8(toc)=_xix_close$$
.xix_close+000014    stw    r3,78(stkp)         r3=05AB620C,78(stkp)=2FF97D80
.xix_close+000018    stw    r4,7C(stkp)         r4=00000020,7C(stkp)=2FF97D84
.xix_close+00001C    lwz    r3,12BC(toc)       r3=05AB620C,12BC(toc)=xclosedbg
.xix_close+000020    lwz    r3,0(r3)           r3=xclosedbg,0(r3)=xclosedbg
.xix_close+000024    lwz    r4,12C0(toc)       r4=00000020,12C0(toc)=pfsdbg
KDB(1)> r return to the end of function
.vno_close+000070    lwz    toc,14(stkp)        toc=TOC,14(stkp)=2FF97D7C
KDB(1)> S 4
.vno_close+000088    bl     <._ptrgl>           r3=05AB620C
.xix_rele+00010C     bl     <._vn_free>         r3=05AB620C
.vn_free+000140      bl     <.gpai_free>       r3=gpa_vnode
.gpai_free+00002C    br     <._vn_free+000144> <._vn_free+000144>
KDB(1)> <CR/LF> repeat last command
.vn_free+00015C     br     <.xix_rele+000110> <.xix_rele+000110>
.xix_rele+000118    bl     <._input>          r3=058F9360
.input+0000A4       bl     <._iclose>         r3=058F9360
.iclose+000148      br     <._input+0000A8>   <._input+0000A8>
KDB(1)> <CR/LF> repeat last command
.input+0001A4       bl     <._insque2>        r3=058F9360
.insque2+00004C     br     <._input+0001A8>   <._input+0001A8>
.input+0001D0       br     <.xix_rele+00011C> <.xix_rele+00011C>
.xix_rele+000164    br     <.vno_close+00008C> <.vno_close+00008C>
KDB(1)> r return to the end of function
.vno_close+00008C    lwz    toc,14(stkp)        toc=TOC,14(stkp)=2FF97D7C
KDB(1)>
```

Dumps/Display/Decode Subcommands for the KDB Kernel Debugger and kdb Command

d, dw, dd, dp, dpw, dpd Subcommands

Generally speaking, the display memory subcommands allow read or write access to be done in virtual or real mode, using an effective address or a real address as input:

- **d** subcommands: real mode access with an effective address as argument.
- **dp** subcommands: real mode access with a real address as argument.
- **ddv** subcommands: virtual mode access with an effective address as argument.
- **ddp** subcommands: virtual mode access with a real address as argument.

The **d** (display bytes) **dw** (display words) **dd** (display double words) subcommands may be used to dump memory areas, specified address is an effective address. Access is done in real mode.

The **dp** (display bytes) **dpw** (display words) **dpc** (display double words) subcommands may be used to dump memory areas, specified address is a real address.

Count argument is in hexadecimal base for display data, so 10 bytes is one line and 10 words is 4 lines. To display from symbol to symbol+0080 the subcommand is **d symbol 80** or **dw symbol 20**

Default count is one line, and <CR/LF> continues display.

Example

```

KDB(0)> d utsname 40 print utsname byte per byte
utsname+000000: 4149 5820 0000 0000 0000 0000 0000 0000  AIX.....
utsname+000010: 0000 0000 0000 0000 0000 0000 0000 0000  .....
utsname+000020: 3030 3030 3030 3030 4130 3030 0000 0000  00000000A000....
utsname+000030: 0000 0000 0000 0000 0000 0000 0000 0000  .....
KDB(0)> <CR/LF> repeat last command
utsname+000040: 3100 0000 0000 0000 0000 0000 0000 0000  1.....
utsname+000050: 0000 0000 0000 0000 0000 0000 0000 0000  .....
utsname+000060: 3400 0000 0000 0000 0000 0000 0000 0000  4.....
utsname+000070: 0000 0000 0000 0000 0000 0000 0000 0000  .....
KDB(0)> <CR/LF> repeat last command
utsname+000080: 3030 3030 3030 3030 4130 3030 0000 0000  00000000A000....
utsname+000090: 0000 0000 0000 0000 0000 0000 0000 0000  .....
xutsname+000000: 0000 0000 0000 0000 0000 0000 0000 0000  .....
devcnt+000000: 0000 0100 0000 0000 0001 239C 0001 23A8  .....#...#
KDB(0)> dw utsname 10 print utsname word per word
utsname+000000: 41495820 00000000 00000000 00000000  AIX.....
utsname+000010: 00000000 00000000 00000000 00000000  .....
utsname+000020: 30303030 30303030 41303030 00000000  00000000A000....
utsname+000030: 00000000 00000000 00000000 00000000  .....
KDB(0)> tr utsname find utsname physical address
Physical Address = 00027E98
KDB(0)> dp 00027E98 40 print utsname using physical address
00027E98: 4149 5820 0000 0000 0000 0000 0000 0000  AIX.....
00027EA8: 0000 0000 0000 0000 0000 0000 0000 0000  .....
00027EB8: 3030 3030 3030 3030 4130 3030 0000 0000  00000000A000....
00027EC8: 0000 0000 0000 0000 0000 0000 0000 0000  .....
KDB(0)> dpw 00027E98 print utsname using physical address
00027E98: 41495820 00000000 00000000 00000000  AIX.....
KDB(0)>

```

dc and dpc Subcommands

The display code subcommands may be used to decode instructions.

Breakpoints are not seen, KDB prints the real instruction.

Example

```

KDB(0)> set 4 set toggle for Power PC syntax
power_pc_syntax is true
KDB(0)> dc resume_pc 10 prints 10 instructions
.resume_pc+000000  lbz    r0,3454(0)      3454=Trconflag
.resume_pc+000004  mfsprg  r15,0
.resume_pc+000008  cmpi   cr0,r0,0
.resume_pc+00000C  lwz   toc,4208(0)      toc=TOC,4208=g_toc
.resume_pc+000010  lwz   r30,4C(r15)
.resume_pc+000014  lwz   r14,40(r15)
.resume_pc+000018  lwz   r31,8(r30)

```

```

.resume_pc+00001C    bne-   cr0.eq,<.resume_pc+0001BC>
.resume_pc+000020    lha    r28,2(r30)
.resume_pc+000024    lwz    r29,0(r14)
KDB(0)> dc mttb 5 prints mttb function
.mttb+000000        li     r0,0
.mttb+000004        mttbl  X r0 X shows that these instructions
.mttb+000008        mttbu  X r3 are not supported by the current architecture
.mttb+00000C        mttbl  X r4 POWER PC 601 processor
.mttb+000010        blr
KDB(0)> set 4 set toggle for Power RS syntax
power_pc_syntax is false
KDB(0)> dc resume_pc 10 prints 10 instructions
.resume_pc+000000    lbz    r0,3454(0)          3454=Trconflag
.resume_pc+000004    mfspr   r15,110
.resume_pc+000008    cmpi   cr0,r0,0
.resume_pc+00000C    l       toc,4208(0)      toc=TOC,4208=g_toc
.resume_pc+000010    l       r30,4C(r15)
.resume_pc+000014    l       r14,40(r15)
.resume_pc+000018    l       r31,8(r30)
.resume_pc+00001C    bne    cr0.eq,<.resume_pc+0001BC>
.resume_pc+000020    lha    r28,2(r30)
.resume_pc+000024    l       r29,0(r14)
KDB(4)> dc scdisk_pm_handler
.scdisk_pm_handler+000000    stmw   r26,FFFFFFE8(stkp)
KDB(4)> tr scdisk_pm_handler
Physical Address = 1D7CA1C0
KDB(4)> dpc 1D7CA1C0
1D7CA1C0    stmw   r26,FFFFFFE8(stkp)

```

dr Subcommand

The display registers subcommand may be used to display general purpose, segment or special registers. The current context is used to find values. After switching from current thread to another one, KDB shows registers of the new one.

For BATs registers (see “bat/Block Address Translation Subcommands” on page 409), **dbat** and **ibat** subcommands must be used.

Example

```

KDB(0)> dr ? print usage
is not a valid register name
Usage:      dr [sp|sr|gp|fp|<reg. name>]
sp reg. name: iar  msr  cr   lr   ctr  xer  mq   tid  asr
..... dsisr dar  dec  sdr0  sdr1  srr0  srr1  dabr  rtc1
..... tbu  tbl  sprg0 sprg1 sprg2 sprg3 pir  fpecr ear  pvr
..... hid0 hid1 iabr dmiss imiss dcmp icmp hash1 hash2 rpa
..... buscsr l2cr l2sr mmcr0 mmcr1 pmc1 pmc2 pmc3 pmc4 pmc5
..... pmc6 pmc7 pmc8 sia  sda
sr reg. name: s0  s1  s2  s3  s4  s5  s6  s7  s8  s9
..... s10 s11 s12 s13 s14 s15
gp reg. name: r0  r1  r2  r3  r4  r5  r6  r7  r8  r9
..... r10 r11 r12 r13 r14 r15 r16 r17 r18 r19
..... r20 r21 r22 r23 r24 r25 r26 r27 r28 r29
..... r30 r31
fp reg. name: f0  f1  f2  f3  f4  f5  f6  f7  f8  f9
..... f10 f11 f12 f13 f14 f15 f16 f17 f18 f19
..... f20 f21 f22 f23 f24 f25 f26 f27 f28 f29
..... f30 f31 fpscr
KDB(0)> dr print general purpose registers
r0 : 00003730 r1 : 2FEDFF88 r2 : 00211B6C r3 : 00000000 r4 : 00000003
r5 : 007FFFFFFF r6 : 0002F930 r7 : 2FEAFFFC r8 : 00000009 r9 : 20019CC8
r10 : 00000008 r11 : 00040B40 r12 : 0009B700 r13 : 2003FC60 r14 : DEADBEEF
r15 : 00000000 r16 : DEADBEEF r17 : 2003FD28 r18 : 00000000 r19 : 20009168
r20 : 2003FD38 r21 : 2FEAFF3C r22 : 00000001 r23 : 2003F700 r24 : 2FEE02E0
r25 : 2FEE0000 r26 : D0005454 r27 : 2A820846 r28 : E3000E00 r29 : E60008C0

```

```

r30 : 00353A6C r31 : 00000511
KDB(0)> dr sp print special registers
iar : 10001C48 msr : 0000F030 cr : 28202884 lr : 100DAF18
ctr : 100DA1D4 xer : 00000003 mq : 00000DF4
dsisr : 42000000 dar : 394A8000 dec : 007DDC00
sdr1 : 00380007 srr0 : 10001C48 srr1 : 0000F030
dabr : 00000000 rtcu : 2DC05E64 rtcl : 2E993E00
sprg0 : 000A5740 sprg1 : 00000000 sprg2 : 00000000 sprg3 : 00000000
pid : 00000000 fpecr : 00000000 ear : 00000000 pvr : 00010001
hid0 : 8101FBC1 hid1 : 00004000 iabr : 00000000
KDB(0)> dr sr print segment registers
s0 : 60000000 s1 : 60001377 s2 : 60001BDE s3 : 60001B7D s4 : 6000143D
s5 : 60001F3D s6 : 600005C9 s7 : 007FFFFFFF s8 : 007FFFFFFF s9 : 007FFFFFFF
s10 : 007FFFFFFF s11 : 007FFFFFFF s12 : 007FFFFFFF s13 : 60000A0A s14 : 007FFFFFFF
s15 : 600011D2
KDB(0)> dr fp print floating point registers
f0 : C027C28F5C28F5C3 f1 : 000333335999999A f2 : 3FE3333333333333
f3 : 3FC9999999999999 f4 : 7FF0000000000000 f5 : 00100000C0000000
f6 : 4000000000000000 f7 : 000000009A068000 f8 : 7FF8000000000000
f9 : 00000000BA411000 f10 : 0000000000000000 f11 : 0000000000000000
f12 : 0000000000000000 f13 : 0000000000000000 f14 : 0000000000000000
f15 : 0000000000000000 f16 : 0000000000000000 f17 : 0000000000000000
f18 : 0000000000000000 f19 : 0000000000000000 f20 : 0000000000000000
f21 : 0000000000000000 f22 : 0000000000000000 f23 : 0000000000000000
f24 : 0000000000000000 f25 : 0000000000000000 f26 : 0000000000000000
f27 : 0000000000000000 f28 : 0000000000000000 f29 : 0000000000000000
f30 : 0000000000000000 f31 : 0000000000000000 fpscr : BA411000
KDB(0)> dr ctr print CTR register
ctr : 100DA1D4
100DA1D4 cmpi cr0,r3,E7 r3=2FEAB008
KDB(0)> dr msr print MSR register
msr : 0000F030 bit set: EE PR FP ME IR DR
KDB(0)> dr cr
cr : 28202884 bits set in CR0 : EQ
.....CR1 : LT
.....CR2 : EQ
.....CR4 : EQ
.....CR5 : LT
.....CR6 : LT
.....CR7 : GT
KDB(0)> dr xer print XER register
xer : 00000003 comparison byte: 0 length: 3
KDB(0)> dr iar print IAR register
iar : 10001C48
10001C48 stw r12,4(stkp) r12=28202884,4(stkp)=2FEAAF4
KDB(0)> set 11 enable 64 bits display on 620 machine
64_bit is true
KDB(0)> dr display 620 general purpose registers
r0 : 000000000244CF0 r1 : 000000000259EB4 r2 : 00000000025A110
r3 : 0000000000A4B60 r4 : 000000000000001 r5 : 000000000000001
r6 : 0000000000000F0 r7 : 000000000001090 r8 : 00000000018DAD0
r9 : 00000000015AB20 r10 : 00000000018D9D0 r11 : 000000000000000
r12 : 00000000023F05C r13 : 0000000000001C8 r14 : 0000000000000BC
r15 : 000000000000040 r16 : 000000000000040 r17 : 00000000080300F0
r18 : 000000000000000 r19 : 000000000000000 r20 : 000000000225A48
r21 : 000000001FF3E00 r22 : 0000000002259D0 r23 : 00000000025A12C
r24 : 000000000000001 r25 : 000000000000001 r26 : 000000001FF42E0
r27 : 000000000000000 r28 : 000000001FF4A64 r29 : 000000001FF4000
r30 : 0000000000034CC r31 : 000000001FF4A64
KDB(0)> dr sp display 620 special registers
iar : 00000000023F288 msr : 000000000021080 cr : 42000440
lr : 000000000245738 ctr : 000000000000000 xer : 00000000
mq : 00000000 asr : 000000000000000
dsisr : 42000000 dar : 00000000000000EC dec : C3528E2F
sdr1 : 01EC0000 srr0 : 00000000023F288 srr1 : 000000000021080
dabr : 0000000000000000 tbu : 00000002 tbl : AF33287B
sprg0 : 0000000000A4C00 sprg1 : 000000000000040

```

```

sprg2 : 0000000000000000 sprg3 : 0000000000000000
pir   : 0000000000000000 ear   : 00000000 pvr   : 00140201
hid0  : 7001C080 iabr  : 0000000000000000
buscsr : 000000000008DC800 l2cr : 0000000000000421A l2sr : 0000000000000000
mmcr0 : 00000000 pmc1  : 00000000 pmc2  : 00000000
sia   : 0000000000000000 sda   : 0000000000000000
KDB(0)>

```

ddvb, ddvh, ddvw, ddvd, ddpd, ddph, and ddpw Subcommands

Note: This subcommand is only available within the **kdb** command; it is not included in the KDB Kernel Debugger.

IO space memory (Direct Store Segment (T=1)) can not be accessed when translation is disabled (see Storage model in POWER PC Operating Environment Architecture book III). **bat** mapped area must also be accessed with translation enabled, else cache controls are ignored.

Access can be done in bytes, half words, words or double words.

Address can be an effective address or a real address.

Four special subcommands **ddvb**, **ddvh**, **ddvw** and **ddvd** may be used to access these areas in translated mode, giving an effective address already mapped. On 64-bit machine, double words correctly aligned are accessed (**ddvd**) in a single load (ld) instruction.

Four special subcommands **ddpb**, **ddph**, **ddpw** and **ddpd** may be used to access these areas in translated mode, giving a physical address that will be mapped. On 64-bit machine, double words correctly aligned are accessed (**ddpd**) in a single load (ld) instruction. **DBAT** interface is used to translate this address in cache inhibited mode (POWER PC only).

Note: Interface with effective address (**ddv.**) assumes that mapping to real address is currently valid. No check is done by KDB. Interface with real address (**ddp.**) can be used to let KDB doing th mapping (attach and detach).

Example on Power PC 601

```

KDB(0)> tr fff19610 show current mapping
BAT mapping for FFF19610
DBAT0 FFC0003A FFC0005F
  bepi 7FE0 brpn 7FE0 b1 001F v 1 wim 3 ks 1 kp 0 pp 2 s 0
  eaddr = FFC00000, paddr = FFC00000 size = 4096 KBytes
KDB(0)> ddvb fff19610 10 print 10 bytes using data relocate mode enable
FFF19610: 0041 96B0 6666 CEEA 0041 A0B0 0041 AAB0 .A..ff...A...A..
KDB(0)> ddvw fff19610 4 print 4 words using data relocate mode enable
FFF19610: 004196B0 76763346 0041A0B0 0041AAB0
KDB(0)>

```

Example on Power PC, PCI machine

```

KDB(0)> ddpw 80000cfc print one word at physical address 80000cfc
80000CFC: D0000080 Read is done in relocated mode, cache inhibited
KDB(0)>

```

find and findp Subcommands

The search in memory subcommands may be used to search a specific pattern in memory.


```
Usage: find -s effective_address string [delta]
       find effective_address pattern [mask [delta]]
Usage: findp -s physical_address string [delta]
       findp physical_address pattern [mask [delta]]
string from one to 256 characters.
pattern is a 32-bits word.
mask is the mask applied on the pattern.
delta is address increment, default one char if -s or one word.
```

Example

```
KDB(0)> tpid print current thread
          SLOT NAME      STATE  TID PRI CPUID CPU FLAGS  WCHAN
thread+002F40 63*nfsd  RUN   03F8F 03C   000 00000000
KDB(0)> find lock_pinned 03F8F 00ffffff 20 search TID in the lock area
compare only 24 low bits, on cache aligned addresses (delta 0x20)
lock_pinned+00D760: 00003F8F 00000000 00000005 00000000
KDB(0)> <CR/LF> repeat last command
Invalid address E800F000, skip to (^C to interrupt)
..... E8800000
Invalid address E8840000, skip to (^C to interrupt)
..... E9000000
Invalid address E9012000, skip to (^C to interrupt)
..... F0000000
KDB(0)> findp 0 E819D200 search in physical memory
00F97C7C: E819D200 00000000 00000000 00000000
KDB(0)> <CR/LF> repeat last command
05C4FB18: E819D200 00000000 00000000 00000000
KDB(0)> <CR/LF> repeat last command
0F7550F0: E819D200 00000000 E60009C0 00000000
KDB(0)> <CR/LF> repeat last command
0F927EE8: E819D200 00000000 05E62D28 00000000
KDB(0)> <CR/LF> repeat last command
0FAE16E8: E819D200 00000000 05D3B528 00000000
KDB(0)> <CR/LF> repeat last command
kdb_get_real_memory: Out of range address 1FFFFFFF
KDB(0)>
```

The **-s** option can be used to enter string of characters. The **'.'** character is used to match any character.

Example

```
KDB(0)>find -s 01A86260 pse search "pse" in pse text code
01A86ED4: 7073 655F 6B64 6200 8062 0518 8063 0000  pse_kdb..b....c..
KDB(0)> <CR/LF> repeat last command
01A92952: 7073 6562 7566 6361 6C6C 735F 696E 6974  psebufcalls_init
KDB(0)> <CR/LF> repeat last command
01A939AE: 7073 655F 6275 6663 616C 6C00 0000 BF81  pse_bufcall.....
KDB(0)> <CR/LF> repeat last command
01A94F5A: 7073 655F 7265 766F 6B65 BEA1 FFD4 7D80  pse_revoke....}.
KDB(0)> <CR/LF> repeat last command
01A9547E: 7073 655F 7365 6C65 6374 BE41 FFC8 7D80  pse_select.A..}.
KDB(0)> find -s 01A86260 pse_....._thread how to use '.'
01A9F586: 7073 655F 626C 6F63 6B5F 7468 7265 6164  pse_block_thread
KDB(0)> <CR/LF> repeat last command
01A9F6EA: 7073 655F 736C 6565 705F 7468 7265 6164  pse_sleep_thread
```

ext and extp Subcommands

The extract from memory subcommands may be used to extract specific zone from memory.

ext will display the number of words at the start address, then display the number of words at address = address + delta, and keep doing this until 'count' loops are done.

extp will display the number of words at the start address, then display the number of words at address = *(address + delta), and keep doing this until 'count' loops are done.

```
Usage: ext    effective_address delta [size [count]]
        ext -p effective_address delta [size [count]]
Usage: extp   physical_address delta [size [count]]
        extp -p physical_address delta [size [count]]
delta is address increment or next address offset.
size is how many words to print, default one line
count is how many extractions to do, default one.
```

Example

```
(0)> ext thread+7c 0000C0 1 20 extract scheduler information from threads
thread+00007C: 00021001      ....
thread+00013C: 00024800      ..H.
thread+0001FC: 00007F01      ....
thread+0002BC: 00017F01      ....
thread+00037C: 00027F01      ....
thread+00043C: 00037F01      ....
thread+0004FC: 00021001      ....
thread+0005BC: 00012402      ..$.
thread+00067C: 00002502      ..%.
thread+00073C: 00002502      ..%.
thread+0007FC: 00002502      ..%.
thread+0008BC: 00032502      ..%.
thread+00097C: 00002502      ..%.
thread+000A3C: 00033C00      ..<.
...
KDB(0)> extp 0 4000000 4 100 extract memory using real address
00000000: 00000000 00000000 00000000 00000000      .....
04000000: 00004001 00000000 00000000 00000000      ..@.....
08000000: 00008001 00000000 00000000 00000000      .....
0C000000: D0071128 F010EA08 F010EA68 F010F028      ...(.h...
10000000: 00000000 00000000 00000000 00000000      .....
14000000: 746C2E63 2C206C69 62636673 2C20626F      tl.c, libcfs, bo
18000000: 20005924 0000031D 20001B04 20005924      .Y$. ... .Y$
1C000000: 000C000D 000E000F 00100011 00120013      .....
20000000: kdb_get_real_memory: Out of range address 20000000
```

The **-p** option specifies that **delta** is offset of the field giving the next address. A list can be printed by this way.

Example

```
(0)> ext -p proc+500 14 8 10 print siblings of a process
proc+000500: 07000000 00000303 00000000 00000000      .....
proc+000510: 00000000 E3000400 E3000500 00000000      .....
proc+000400: 07000000 00000303 00000000 00000000      .....
proc+000410: 00000000 E3000300 E3000400 00000000      .....
proc+000300: 07000000 00000303 00000000 00000000      .....
proc+000310: 00000000 E3000200 E3000300 00000000      .....
proc+000200: 07000000 00000303 00000000 00000000      .....
proc+000210: 00000000 00000000 E3000200 00000000      .....
```

Modify Memory Subcommands for the KDB Kernel Debugger and kdb Command

m, mw, md, mp, mpw, and mpd Subcommands

Note: This subcommand is only available within the **kdb** command; it is not included in the KDB Kernel Debugger.

Generally speaking, read or write access can be done in virtual or real mode, using an effective address or a real address as input:

- **m** subcommands: real mode access with an effective address as argument.
- **mp** subcommands: real mode access with a real address as argument.
- **mdv** subcommands: virtual mode access with an effective address as argument.
- **mdp** subcommands: virtual mode access with a real address as argument.

For each display subcommand **d dw dd** using an effective address, there is a memory modify subcommand **m** (modify bytes) **mw** (modify words) **md** (modify double words).

For each display subcommand **dp dpw dpd** using a real address, there is a memory modify subcommand **mp** (modify bytes) **mpw** (modify words) **mpd** (modify double words).

These subcommands are interactive, each modification is entered one by one. The first unexpected input stops modification. "." for example may be used as <eod>. In the following example, we shows how to do a patch.

If a break point is set at the same address, use the **mw** subcommand to keep break point coherency.

Note: Symbolic expressions are not allowed as input.

Example

```

KDB(0)> dc @iar print current instruction
.open+000000 mflr r0
KDB(0)> mw @iar nop current instruction
.open+000000: 7C0802A6 = 60000000
.open+000004: 93E1FFFC = . end of input
KDB(0)> dc @iar print current instruction
.open+000000 ori r0,r0,0
KDB(0)> m @iar restore current instruction byte per byte
.open+000000: 60 = 7C
.open+000001: 00 = 08
.open+000002: 00 = 02
.open+000003: 00 = A6
.open+000004: 93 = . end of input
KDB(0)> dc @iar print current instruction
.open+000000 mflr r0
KDB(0)> tr @iar physical address of current instruction
Physical Address = 001C5BA0
KDB(0)> mwp 001C5BA0 modify with physical address
001C5BA0: 7C0802A6 = <CR/LF>
001C5BA4: 93E1FFFC = <CR/LF>
001C5BA8: 90010008 = <CR/LF>
001C5BAC: 9421FF40 = 60000000
001C5BB0: 83E211C4 = . end of input
KDB(0)> dc @iar 5 print instructions
.open+000000 mflr r0
.open+000004 stw r31,FFFFFFFC(stkp)
.open+000008 stw r0,8(stkp)
.open+00000C ori r0,r0,0
.open+000010 lwz r31,11C4(toc) 11C4(toc)=_open$$
KDB(0)> mw open+c restore instruction
.open+00000C: 60000000 = 9421FF40
.open+000010: 83E211C4 = . end of input
KDB(0)> dc open+c print instruction
.open+00000C stwu stkp,FFFFFF40(stkp)
KDB(0)>

```

mr Subcommand

Each register may be altered by the **mr** subcommand. When the register is in the **mst** context, KDB alters this mst and the modification will be taken as **resume**. When the register is a special one, the processor register is altered immediately. Symbolic expressions are allowed as input.

Example

```
KDB(0)> dc @iar print current instruction
.open+000000 mflr r0
KDB(0)> mr iar modify current instruction address
iar : 001C5BA0 = @iar+4
KDB(0)> dc @iar print current instruction
.open+000004 stw r31,FFFFFFC(stkp)
KDB(0)> mr iar restore current instruction address
iar : 001C5BA4 = @iar-4
KDB(0)> dc @iar print current instruction
.open+000000 mflr r0
KDB(0)> mr sr modify first invalid segment register
s0 : 00000000 = <CR/LF>
s1 : 60000323 = <CR/LF>
s2 : 20001E1E = <CR/LF>
s3 : 007FFFFFFF = 0
s4 : 007FFFFFFF = . end of input
KDB(0)> dr s3 print segment register 3
s3 : 00000000
KDB(0)> mr s3 restore segment register 3
s3 : 00000000 = 007FFFFFFF
KDB(0)> mr f29 modify floating point register f29
f29 : 0000000000000000 = 000333335999999A
KDB(0)> dr f29
f29 : 000333335999999A
KDB(0)> u
Uthread [2FF3B400]:
    save@.....2FF3B400    fpr@.....2FF3B550
...
KDB(0)> dd 2FF3B550 20
__ublock+000150: C027C28F5C28F5C3 000333335999999A .'..\(...33Y...
__ublock+000160: 3FE3333333333333 3FC9999999999999 ?..333333?.....
__ublock+000170: 7FF0000000000000 00100000C0000000 .....
__ublock+000180: 4000000000000000 000000009A068000 @.....
__ublock+000190: 7FF8000000000000 00000000BA411000 .....A..
__ublock+0001A0: 0000000000000000 0000000000000000 .....
__ublock+0001B0: 0000000000000000 0000000000000000 .....
__ublock+0001C0: 0000000000000000 0000000000000000 .....
__ublock+0001D0: 0000000000000000 0000000000000000 .....
__ublock+0001E0: 0000000000000000 0000000000000000 .....
__ublock+0001F0: 0000000000000000 0000000000000000 .....
__ublock+000200: 0000000000000000 0000000000000000 .....
__ublock+000210: 0000000000000000 0000000000000000 .....
__ublock+000220: 0000000000000000 0000000000000000 .....
__ublock+000230: 0000000000000000 000333335999999A .....33Y...
__ublock+000240: 0000000000000000 0000000000000000 .....
KDB(0)>
```

mdvb, mdvh, mdvw, mdvd, mdpb, mdph, mdpw, mdpd Subcommands

Note: This subcommand is only available within the **kdb** command; it is not included in the KDB Kernel Debugger.

Specific subcommands are available to write in IO space memory. To avoid bad effects, memory is not read before, only the specified write is performed with translation enabled.

Access can be done in bytes, half words, words or double words.

Address can be an effective address or a real address.

Four special subcommands **mdvb**, **mdvh**, **mdvw** and **mdvd** may be used to access these areas in translated mode, giving an effective address already mapped. On 64-bit machine, double words correctly aligned are accessed (**mdvd**) in a single store instruction.

Four special subcommands **mdpb**, **mdph**, **mdp**w and **mdpd** may be used to access these areas in translated mode, giving a physical address that will be mapped. On 64-bit machine, double words correctly aligned are accessed (**mdpd**) in a single store instruction. **DBAT** interface is used to translate this address in cache inhibited mode (POWER PC only).

Note: Interface with effective address (**mdv**.) assumes that mapping to real address is currently valid. No check is done by KDB. Interface with real address (**mdp**.) can be used to let KDB doing the mapping (attach and detach).

Example on Power PC 601

```
KDB(0)> tr FFF19610 print physical mapping
BAT mapping for FFF19610
DBAT0 FFC0003A FFC0005F
  bepi 7FE0 brpn 7FE0 bl 001F v 1 wim 3 ks 1 kp 0 pp 2 s 0
  eaddr = FFC00000, paddr = FFC00000 size = 4096 KBytes
KDB(0)> mdvb fff19610 byte modify with data relocate enable
FFF19610: ?? = 00
FFF19611: ?? = 00
FFF19612: ?? = . end of input
KDB(0)> mdvw fff19610 word modify with data relocate enable
FFF19610: ???????? = 004196B0
FFF19614: ???????? = . end of input
KDB(0)>
```

Example on Power PC, PCI machine

```
KDB(0)> mdpw 80000cf8 change one word at physical address 80000cf8
80000CF8: ???????? = 84000080
80000CFC: ???????? = .Write is done in relocated mode, cache inhibited
KDB(0)> ddpw 80000cfc print one word at physical address 80000cfc
80000CFC: D2000000
KDB(0)> mdpw 80000cfc change one word at physical address 80000cfc
80000CFC: ???????? = d0000000
80000D00: ???????? = .
KDB(0)> mdpw 80000cf8 change one word at physical address 80000cf8
80000CF8: ???????? = 8c000080
80000CFC: ???????? = .
KDB(0)> ddpw 80000cfc print one word at physical address 80000cfc
80000CFC: D2000080
```

Namelist/Symbol Subcommands for the KDB Kernel Debugger and kdb Command

nm and ts Subcommands

The following subcommands may be used to swap from symbol to hexadecimal values.

Example

```
KDB(0)> nm __ublock print symbol value
Symbol Address : 2FF3B400
KDB(0)> ts E3000000 print symbol name
proc+000000
```

ns Subcommand

The `ns` toggle may be used to disable symbol printing.

Example

```
KDB(0)> set 2 do not print context
mst_wanted is false
KDB(0)> f print stack frame
thread+00D080 STACK:
[000095A4].simple_lock+0000A4 ()
[0007F4A0]v_prefreescb+000038 (??, ??)
[00017AC4]isync_vcs3+000004 (??, ??)
____ Exception (2FF40000) ____
[00009414].unlock_enable+000110 ()
[00009410].unlock_enable+00010C ()
[0000CDD0]as_det+0000A8 (??, ??)
[001B33F8]shm_freespace+000080 (??, ??)
[001F6A04]rmmaseg+0000D0 (??)
[001E41DC]vm_map_entry_delete+00023C (??, ??)
[001E4828]vm_map_delete+000158 (??, ??, ??)
[001E5034]vm_map_remove+000064 (??, ??, ??)
[001E6514]munmap+0000C0 (??, ??)
[000036FC].sys_call+000000 ()
KDB(0)> ns enable no symbol printing
Symbolic name translation off
KDB(0)> f print stack frame
E600D080 STACK:
000095A4 ()
0007F4A0 (??, ??)
00017AC4 (??, ??)
____ Exception (2FF40000) ____
00009414 ()
00009410 ()
0000CDD0 (??, ??)
001B33F8 (??, ??)
001F6A04 (??)
001E41DC (??, ??)
001E4828 (??, ??, ??)
001E5034 (??, ??, ??)
001E6514 (??, ??)
000036FC ()
KDB(0)> ns disable no symbol printing
Symbolic name translation on
KDB(0)>
```

Watch Break Points Subcommands for the KDB Kernel Debugger and kdb Command

wr, ww, wrw, cw, lwr, lww, lwrw, and lcw Subcommands

Note: This subcommand is only available within the **kdb** command; it is not included in the KDB Kernel Debugger.

On POWER PC architecture, a watch register may be used (called **DABR** Data Address Breakpoint Register or **HID5** on POWER 601) to enter KDB when a specified effective address is accessed. The register holds a double-word effective address and bits to specify load and/or store operation. See PowerPC Implementation Definition for the 601-604-620 Processor (book IV) to have more information. The **wr** subcommand may be used to stop on load instruction. The **ww** subcommand may be used to stop on store instruction. The **wrw** subcommand may be used to stop on load or store instruction. Without argument, the subcommand prints the current active watch subcommand. The **cw** subcommand may be used to clear the last watch subcommand. This subcommand is global to all processors, each processor may have different watch address with the local subcommands **lwr lww lwrw lcw**. When no size is specified, the default size is 8 bytes and the address is double word aligned. Otherwise KDB checks the faulting address with the specified range and continues execution if it does not match.

It is possible to specify if the address is physical or virtual with **-p** or **-v** option. 604 and 620 processors take care of the translation mode, and it is necessary to specify which mode is used. By default KDB chooses the current state of the machine: if the subcommand is entered before VMM initialisation, the watch address is physical (real address), else virtual (effective address).

Example

```
KDB(0)> ww -p emulate_count set a data break point (physical address, write mode)
KDB(0)> ww print current data break points
CPU 0: emulate_count+000000 paddr=00238360 size=8 hit=0 mode=W
CPU 1: emulate_count+000000 paddr=00238360 size=8 hit=0 mode=W
KDB(0)> e exit the debugger
...
Watch trap: 00238360 <emulate_count+000000>
power_asm_emulate+00013C stw r28,0(r30)
r28=0000003A,0(r30)=emulate_count
KDB(0)> ww print current data break points
CPU 0: emulate_count+000000 paddr=00238360 size=8 hit=1 mode=W
CPU 1: emulate_count+000000 paddr=00238360 size=8 hit=0 mode=W
KDB(0)> wr sysinfo set a data break point (read mode)
KDB(0)> wr print current data break points
CPU 0: sysinfo+000000 eaddr=003BA9D0 vsid=00000000 size=8 hit=0 mode=R
CPU 1: sysinfo+000000 eaddr=003BA9D0 vsid=00000000 size=8 hit=0 mode=R
KDB(0)> e exit the debugger
...
Watch trap: 003BA9D4 <sysinfo+000004>
.fetch_and_add+000008 lwarx r3,0,r6
r3=sysinfo+000004,r6=sysinfo+000004
KDB(0)> cw clear data break points
```

Miscellaneous Subcommands for the KDB Kernel Debugger and kdb Command

time and debug Subcommands

Note: This subcommand is only available within the **kdb** command; it is not included in the KDB Kernel Debugger.

debug subcommand can be used to print additional information during KDB execution, in order to debug the debugger. **time** can be used to have print elapsed time.

Example

```
KDB(4)> debug ? debug help
vmm HW lookup debug... on with arg 'dbg1++', off with arg 'dbg1-'
vmm tr/tv cmd debug... on with arg 'dbg2++', off with arg 'dbg2-'
vmm SW lookup debug... on with arg 'dbg3++', off with arg 'dbg3-'
symbol lookup debug... on with arg 'dbg4++', off with arg 'dbg4-'
stack trace debug.... on with arg 'dbg5++', off with arg 'dbg5-'
BRKPT debug (list)... on with arg 'dbg61++', off with arg 'dbg61-'
BRKPT debug (instr)... on with arg 'dbg62++', off with arg 'dbg62-'
BRKPT debug (suspend).. on with arg 'dbg63++', off with arg 'dbg63-'
BRKPT debug (phantom).. on with arg 'dbg64++', off with arg 'dbg64-'
BRKPT debug (context).. on with arg 'dbg65++', off with arg 'dbg65-'
DABR debug (address).. on with arg 'dbg71++', off with arg 'dbg71-'
DABR debug (register).. on with arg 'dbg72++', off with arg 'dbg72-'
DABR debug (status)... on with arg 'dbg73++', off with arg 'dbg73-'
BRAT debug (address).. on with arg 'dbg81++', off with arg 'dbg81-'
BRAT debug (register).. on with arg 'dbg82++', off with arg 'dbg82-'
BRAT debug (status)... on with arg 'dbg83++', off with arg 'dbg83-'
BRKPT debug (context).. on this debug feature is enable
KDB(4)> debug dbg5++ enable debug mode
stack trace debug.... on
KDB(4)> f stack frame in debug mode
thread+000180 STACK:
=== Look for traceback at 0x00015278
=== Got traceback at 0x00015280 (delta = 0x00000008)
=== has_tboff = 1, tb_off = 0xD8
=== Trying to find Stack Update Code from 0x000151A8 to 0x00015278
=== Found 0x9421FFA0 at 0x000151B8
=== Trying to find Stack Restore Code from 0x000151A8 to 0x0001527C
=== Trying to find Registers Save Code from 0x000151A8 to 0x00015278
[00015278]waitproc+0000D0 ()
=== Look for traceback at 0x00015274
=== Got traceback at 0x00015280 (delta = 0x0000000C)
=== has_tboff = 1, tb_off = 0xD8
[00015274]waitproc+0000CC ()
=== Look for traceback at 0x0002F400
=== Got traceback at 0x0002F420 (delta = 0x00000020)
=== has_tboff = 1, tb_off = 0x30
[0002F400]procentry+000010 (??, ??, ??, ??)
KDB(4)> time time report
No elapsed time to report
... later on
KDB(7)> time time report
Elapsed time since last leaving the debugger:
0 upper ticks and 669242417 lower ticks.
```

Conditional Subcommands for the KDB Kernel Debugger and kdb Command

test Subcommand

The **test** subcommand may be used to break at a specified address when a condition becomes true.

Example

```
KDB(0)> bt open "[ @sysinfo >= 3d ]" stop on open() if condition true
KDB(0)> e exit debugger
...
Enter kdb [ @sysinfo >= 3d ]
KDB(1)> bt display current active trace break points
0: .open+000000 (sid:00000000) trace {hit: 1} {script: [ @sysinfo >= 3d ]}
KDB(1)> dw sysinfo 1 print sysinfo value
sysinfo+000000: 0000004A
```

Calculator Converter Subcommands for the KDB Kernel Debugger and kdb Command

hcal and dcal Subcommands

The **hcal** subcommand may be used to convert hexadecimal values.

The **dcal** subcommand may be used to convert decimal values.

Example

```
KDB(0)> hcal 0x10000 convert a single value
Value hexa: 00010000      Value decimal: 65536
KDB(0)> dcal 1024*1024 convert an expression
Value decimal: 1048576    Value hexa: 00100000
KDB(0)> set 11 64 bits printing
64_bit is true
KDB(0)> hcal 0-1 convert -1
Value hexa: FFFFFFFF      Value decimal: -1 Unsigned: 18446744073709551615
KDB(0)> set 11 32 bits printing
64_bit is false
KDB(0)> hcal 0-1 convert -1
Value hexa: FFFFFFFF      Value decimal: -1 Unsigned: 4294967295
```

Machine Status Subcommands for the KDB Kernel Debugger and kdb Command

status Subcommand

The **stat** subcommand may be used to display the last kernel **printf()** messages, still in memory. The reason why the debugger is called is also displayed. Warning, there is one reason per processor. Last line gives information about processor that crashed:

- Processor logical number
- Current Save Area (CSA) address
- LED value

Running on a dump file the **cpu** subcommand must be used to switch to the processor that failed. It is done by default after the **stat** subcommand.

Example

```
KDB(6)> stat machine status got with kdb kernel
RS6K_SMP_MCA POWER_PC POWER_604 machine with 8 cpu(s)
SYSTEM STATUS:
sysname: AIX
nodename: jumbo32
release: 2
version: 4
machine: 00920312A000
nid: 920312A0
Illegal Trap Instruction Interrupt in Kernel
age of system: 1 day, 5 hr., 59 min., 50 sec.
SYSTEM MESSAGES
AIX Version 4.2
Starting physical processor #1 as logical #1... done.
Starting physical processor #2 as logical #2... done.
Starting physical processor #3 as logical #3... done.
Starting physical processor #4 as logical #4... done.
Starting physical processor #5 as logical #5... done.
Starting physical processor #6 as logical #6... done.
Starting physical processor #7 as logical #7... done.
<- end_of_buffer
CPU 6 CSA 00427EB0 at time of crash, error code for LEDs: 70000000
(0)> stat machine status got with kdb running on the dump file
RS6K_SMP_MCA POWER_PC POWER_604 machine with 4 cpu(s)
..... SYSTEM STATUS
sysname... AIX          nodename.. zoo22
release... 3           version... 4
machine... 00989903A6  nid..... 989903A6
time of crash: Sat Jul 12 12:34:32 1997
age of system: 1 day, 2 hr., 3 min., 49 sec.
..... SYSTEM MESSAGES
AIX Version 4.3
Starting physical processor #1 as logical #1... done.
Starting physical processor #2 as logical #2... done.
Starting physical processor #3 as logical #3... done.
<- end_of_buffer
..... CPU 0 CSA 004ADEB0 at time of crash, error code for LEDs: 30000000
thread+01B438 STACK:
[00057F64]v_sync+0000E4 (B01C876C, 0000001F [??])
[000A4FA0]v_presync+000050 (??, ??)
[0002B05C]begbt_603_patch_2+000008 (??, ??)
Machine State Save Area [2FF3B400]
iar  : 0002AF4C  msr  : 000010B0  cr   : 24224220  lr   : 0023D474
ctr  : 00000004  xer  : 20000008  mq   : 00000000
r0   : 000A4F50  r1   : 2FF3A600  r2   : 002E62B8  r3   : 00000000  r4   : 07D17B60
r5   : E601B438  r6   : 00025225  r7   : 00025225  r8   : 00000106  r9   : 00000004
r10  : 0023D474  r11  : 2FF3B400  r12  : 000010B0  r13  : 000C0040  r14  : 2FF229A0
r15  : 2FF229BC  r16  : DEADBEEF  r17  : DEADBEEF  r18  : DEADBEEF  r19  : 00000000
r20  : 0048D4C0  r21  : 0048D3E0  r22  : 07D6EE90  r23  : 00000140  r24  : 07D61360
r25  : 00000148  r26  : 0000014C  r27  : 07C75FF0  r28  : 07C75FFC  r29  : 07C75FF0
r30  : 07D17B60  r31  : 07C76000
s0   : 00000000  s1   : 007FFFFFF  s2   : 00001DD8  s3   : 007FFFFFF  s4   : 007FFFFFF
s5   : 007FFFFFF  s6   : 007FFFFFF  s7   : 007FFFFFF  s8   : 007FFFFFF  s9   : 007FFFFFF
s10  : 007FFFFFF  s11  : 00000101  s12  : 0000135B  s13  : 00000CC5  s14  : 00000404
s15  : 6000096E
prev  00000000  kjmpbuf  2FF3A700  stackfix  00000000  intpri  0B
curid 00003C60  sralloc  E01E0000  ioalloc  00000000  backt   00
flags  00 tid    00000000  excp_type 00000000
fpscr  00000000  fpau     00  fpinfo  00  fpscrx  00000000
o_iar  00000000  o_toc   00000000  o_arg1   00000000
excbranch 00000000  o_vaddr 00000000  mstext  00000000
Except :
csr  00000000  dsir  40000000  bit set: DSISR_PFT
srval 00000000  dar   07CA705C  dsirr 00000106
[0002AF4C].backt+000000 (00000000, 07D17B60 [??])
[0023D470]ilogsync+00014C (??)
```

```

[002894B8]logsync+000090 (??)
[0028899C]logmvc+000124 (??, ??, ??, ??)
[0023AB68]logafter+000100 (??, ??, ??)
[0023A46C]commit2+0001EC (??)
[0023BF50]finicom+0000BC (??, ??)
[0023C2CC]comlist+0001F0 (??, ??)
[0029391C]jfs_rename+000794 (??, ??, ??, ??, ??, ??, ??)
[00248220]vnode_rename+000038 (??, ??, ??, ??, ??, ??, ??)
[0026A168]rename+000380 (??, ??)
(0)>

```

switch Subcommand

The **switch** subcommand is very useful. By default, KDB shows the current process virtual space. But it is possible to elect another process, and to have all its virtual space on line. When KDB is exiting, the initial context is automatically restored. If local break points are process/thread attached, the switched context is taken as break point context. As kernel address space and user address space are not identical, the **switch** subcommand can be used to switch between user (**sw u**) and kernel (**sw k**) space.

Example

```

KDB(0)> sw 12 switch to thread slot 12
Switch to thread: <thread+000900>
KDB(0)> f print stack trace
thread+000900 STACK:
[000215FC]e_block_thread+000250 ()
[00021C48]e_sleep_thread+000070 (??, ??, ??)
[000200F4]errread+00009C (??, ??)
[001C89B4]rdevread+000120 (??, ??, ??, ??)
[0023A61C]cdev_rdwr+00009C (??, ??, ??, ??, ??, ??, ??)
[00216324]spec_rdwr+00008C (??, ??, ??, ??, ??, ??, ??, ??)
[001CEA3C]vnode_rdwr+000070 (??, ??, ??, ??, ??, ??, ??, ??)
[001BDB0C]rwuiio+0000CC (??, ??, ??, ??, ??, ??, ??, ??)
[001BDF40]rdwr+000184 (??, ??, ??, ??, ??, ??)
[001BDD68]kreadv+000064 (??, ??, ??, ??)
[000037D8].sys_call+000000 ()
[D0046B68]read+000028 (??, ??, ??)
[1000167C]child+000120 ()
[10001A84]main+0000E4 (??, ??)
[1000014C].__start+00004C ()
KDB(0)> dr sr display segment registers
s0 : 00000000 s1 : 007FFFFFFF s2 : 00000AB7 s3 : 007FFFFFFF s4 : 007FFFFFFF
s5 : 007FFFFFFF s6 : 007FFFFFFF s7 : 007FFFFFFF s8 : 007FFFFFFF s9 : 007FFFFFFF
s10 : 007FFFFFFF s11 : 007FFFFFFF s12 : 007FFFFFFF s13 : 6000058B s14 : 00000204
s15 : 60000CBB
KDB(0)> sw u switch to user context
KDB(0)> dr sr display segment registers
s0 : 60000000 s1 : 600009B1 s2 : 60000AB7 s3 : 007FFFFFFF s4 : 007FFFFFFF
s5 : 007FFFFFFF s6 : 007FFFFFFF s7 : 007FFFFFFF s8 : 007FFFFFFF s9 : 007FFFFFFF
s10 : 007FFFFFFF s11 : 007FFFFFFF s12 : 007FFFFFFF s13 : 6000058B s14 : 007FFFFFFF
s15 : 60000CBB
Now it is possible to look at user code
For example, find how read() is called by child()
KDB(0)> dc 1000167C print child() code (seg 1 is now valid)
1000167C b1 <1000A1BC>
KDB(0)> dc 1000A1BC 6 print child() code
1000A1BC lwz r12,244(toc)
1000A1C0 stw toc,14(stkp)
1000A1C4 lwz r0,0(r12)
1000A1C8 lwz toc,4(r12)
1000A1CC mtctr r0
1000A1D0 bcctr
... find stack pointer of child() routine with 'set 9; f'
[D0046B68]read+000028 (??, ??, ??)

```

```

=====
2FF22B50: 2FF2 2D70 2000 9910 1000 1680 F00F 3130 /.-p .....10
2FF22B60: F00F 1E80 2000 4C54 0000 0003 0000 4503 .... .LT.....E.
2FF22B70: 2FF2 2B88 0000 D030 0000 0000 6000 0000 /.+....0....'...
2FF22B80: 6000 09B1 0000 0000 0000 0002 0000 0002 '.....
=====
[1000167C]child+000120 ()
...
(0)> dw 2FF22B50+14 1 - stw toc,14(stkp)
2FF22B64: 20004C54 toc address
(0)> dw 20004C54+244 1 - lzw r12,244(toc)
20004E98: F00BF5C4 function descriptor address
(0)> dw F00BF5C4 2 - lzw r0,0(r12) - lzw toc,4(r12)
F00BF5C4: D0046B40 F00C1E9C function descriptor (code and toc)
(0)> dc D0046B40 11 - bcctr will execute:
D0046B40 mflr r0
D0046B44 stw r31,FFFFFFFC(stkp)
D0046B48 stw r0,8(stkp)
D0046B4C stwu stkp,FFFFFFB0(stkp)
D0046B50 stw r5,3C(stkp)
D0046B54 stw r4,38(stkp)
D0046B58 stw r3,40(stkp)
D0046B5C addic r4,stkp,38
D0046B60 li r5,1
D0046B64 li r6,0
D0046B68 bl <D00ADC68> read+000028
The following example shows some of the differences between kernel and user
mode for 64-bit process
(0)> sw k kernel mode
(0)> dr msr kernel machine status register
msr : 000010B0 bit set: ME IR DR
(0)> dr r1 kernel stack pointer
r1 : 2FF3B2A0 2FF3B2A0
(0)> f stack frame (kernel MST)
thread+002A98 STACK:
[00031960]e_block_thread+000224 ()
[00041738]nsleep+000124 (??, ??)
[01CFF0F4]nsleep64_+000058 (0FFFFFFF, F0000001, 00000001, 10003730, 1FFFFFF0, 1FFFFFF8)
[000038B4].sys_call+000000 ()
[80000010000867C]080000010000867C (??, ??, ??, ??)
[80000010001137C]nsleep+000094 (??, ??)
[800000100058204]sleep+000030 (??)
[100000478]main+0000CC (0000000100000001, 00000000200FEB78)
[10000023C]__start+000044 ()
(0)> sw u user mode
(0)> dr msr user machine status register
msr : 800000004000D0B0 bit set: EE PR ME IR DR
(0)> dr r1 user stack pointer
r1 : 0FFFFFFFFFFFFFF0 0FFFFFFFFFFFFFF0
(0)> f stack frame (kernel MST extension)
thread+002A98 STACK:
[8000001000581D4]sleep+000000 (0000000000000064 [??])
[100000478]main+0000CC (0000000100000001, 00000000200FEB78)
[10000023C]__start+000044 ()

```

Kernel Extension Loader Subcommands for the KDB Kernel Debugger and kdb Command

Ike, stbl, and rmst Subcommands

The following subcommands **Ike stbl** may be used to display current state of loaded kernel extensions. During boot phase, KDB is called to try to load extension symbol table. A message is printed to say what happens. In the following example, **unix** and **one** driver have symbol table. If the kernel extension is stripped, the

symbol table is not loaded in memory, but it is possible to build a new symbol table with the `traceback` table. The `Ike` subcommand can be used for this purpose. A symbol name cache is managed inside KDB, and the cache is filled with function names with `Ike entry` subcommand. This cache is a circular buffer, old entries will be removed by new ones when the cache is full.

- `Ike <slot | address>` populate the cache with loader entry as argument
- `Ike -a <address>` populate the cache with kernext address as argument
- `Ike -l` list current cached name list
- `stbl` list all symbol tables connected to KDB
- `rmst` disconnect a specified symbol table

Example

```
... during boot phase
no symbol [/etc/drivers/mddtu_load]
no symbol [/etc/drivers/fd]
Preserving 14280 bytes of symbol table [/etc/drivers/rsdd]
no symbol [/etc/drivers/posixdd]
no symbol [/etc/drivers/dtropicdd]
...
KDB(4)> stbl list symbol table entries
LDRENTY TEXT DATA TOC MODULE NAME
1 00000000 00000000 00000000 00207EF0 /unix
2 0B04C400 0156F0F0 015784F0 01578840 /etc/drivers/rsdd
KDB(4)> rmst 2 ignore second entry
KDB(4)> stbl list symbol table entries
LDRENTY TEXT DATA TOC MODULE NAME
1 00000000 00000000 00000000 00207EF0 /unix
KDB(4)> stbl 1 list a symbol table entry
LDRENTY TEXT DATA TOC MODULE NAME
1 00000000 00000000 00000000 00207EF0 /unix
st_desc addr.... 00153920
symoff..... 002A9EB8
nb_sym..... 0000551E
...
(0)> lke ? help
A KERNEXT FUNCTION NAME CACHE exists
with 1024 entries max (circular buffer)
Usage: lke <entry> to populate the cache
Usage: lke -a <address> to populate the cache
Usage: lke -l to list the cache
(0)> lke list loaded kernel extensions
ADDRESS FILE FILESIZE FLAGS MODULE NAME
1 055ADD00 014620C0 000076CC 00000262 /usr/lib/drivers/pse/psekdb
2 055AD780 05704000 000702D0 00000272 /usr/lib/drivers/nfs.ext
3 055AD880 05781000 00000D74 00000248 /unix
4 055AD380 01461D58 00000348 00000272 /usr/lib/drivers/nfs_kdes.ext
5 055AD800 056F7000 00000D20 00000248 /unix
6 055AD600 01455140 0000CC0C 00000262 /etc/drivers/ptydd
7 055AD500 01451400 00003D2C 00000272 /usr/lib/drivers/if_en
8 055AD580 05656000 00000D20 00000248 /unix
9 055AD400 055FB000 0004E038 00000272 /usr/lib/drivers/netinet
...
39 05518200 0135FA60 00006EFC 00000262 /etc/drivers/bcsidd
40 05518300 0135F5B8 0000049C 00000272 /etc/drivers/lsadd
41 05518180 04F7D000 00000CCC 00000248 /unix
42 05518280 0135E020 00001590 00000262 /etc/drivers/mca_ppc_busdd
43 04F61100 00326BF8 00000000 00000256 /unix
44 04F61158 04F62000 00000CCC 00000248 /unix
(0)> lke 40 print slot 40 and process traceback table
ADDRESS FILE FILESIZE FLAGS MODULE NAME
40 05518300 0135F5B8 0000049C 00000272 /etc/drivers/lsadd
le_flags..... TEXT KERNELEX DATAINTEXT DATA DATAEXISTS
le_next..... 05518180 le_fp..... 00000000
```

```

le_filename.... 05518358 le_file..... 0135F5B8
le_filesize.... 0000049C le_data..... 0135F988
le_tid..... 00000000 le_datasize... 000000CC
le_usecount.... 00000008 le_loadcount... 00000001
le_ndepend.... 00000001 le_maxdepend... 00000001
le_ule..... 04F86000 le_deferred.... 00000000
le_exports..... 04F86000 le_de..... 632E6100
le_searchlist.. C0000420 le_dlusecount.. 00000000
le_dlindex.... 0000622F le_lex..... 00000000
TOC@..... 0135FA10
                                <PROCESS TRACE BACKS>
                                .lsa_pos_unlock 0135F6B4                                .lsa_pos_lock 0135F6E4
                                .lsa_config 0135F738                                .lockl.glink 0135F86C
                                .pincode.glink 0135F894                                .lock_alloc.glink 0135F8BC
                                .simple_lock_init.glink 0135F8E4                                .unpincode.glink 0135F90C
                                .lock_free.glink 0135F934                                .unlockl.glink 0135F95C
(0)> lke -a 0135E51C using a kernext address as argument
ADDRESS FILE FILESIZE FLAGS MODULE NAME
  1 05518280 0135E020 00001590 00000262 /etc/drivers/mca_ppc_busdd
le_flags..... TEXT DATAINTEXT DATA DATAEXISTS
le_next..... 04F61100 le_fp..... 00000000
le_filename.... 055182D8 le_file..... 0135E020
le_filesize.... 00001590 le_data..... 0135F380
le_tid..... 00000000 le_datasize... 00000230
le_usecount.... 00000001 le_loadcount... 00000001
le_ndepend.... 00000001 le_maxdepend... 00000001
le_ule..... 00000000 le_deferred.... 00000000
le_exports..... 00000000 le_de..... 6366672E
le_searchlist.. C0000420 le_dlusecount.. 00000000
le_dlindex.... 00006C69 le_lex..... 00000000
TOC@..... 0135F4F8
                                <PROCESS TRACE BACKS>
                                .mca_ppc_businit 0135E120                                .complete_error 0135E38C
                                .d_protect_ppc 0135E51C                                .d_move_ppc 0135E608
                                .d_bflush_ppc 0135E630                                .d_cflush_ppc 0135E65C
                                .d_complete_ppc 0135E688                                .d_master_ppc 0135E7B4
                                .d_slave_ppc 0135E974                                .d_unmask_ppc 0135EBA4
                                .d_mask_ppc 0135EC40                                .d_clear_ppc 0135ECD8
                                .d_init_ppc 0135ED8C                                .vm_att.glink 0135EF88
                                .lock_alloc.glink 0135EFB0                                .simple_lock_init.glink 0135EFD8
                                .vm_det.glink 0135F000                                .pincode.glink 0135F028
                                .bcopy 0135F060                                .copystr 0135F238
                                .errsave.glink 0135F2E0                                .xmemdma_ppc.glink 0135F308
                                .xmemqra.glink 0135F330                                .xmemacc.glink 0135F358
(0)> lke -l list current name cache
                                KERNEXT FUNCTION NAME CACHE
                                .lsa_pos_unlock 0135F6B4                                .lsa_pos_lock 0135F6E4
                                .lsa_config 0135F738                                .lockl.glink 0135F86C
                                .pincode.glink 0135F894                                .lock_alloc.glink 0135F8BC
                                .simple_lock_init.glink 0135F8E4                                .unpincode.glink 0135F90C
                                .lock_free.glink 0135F934                                .unlockl.glink 0135F95C
                                .mca_ppc_businit 0135E120                                .complete_error 0135E38C
                                .d_protect_ppc 0135E51C                                .d_move_ppc 0135E608
                                .d_bflush_ppc 0135E630                                .d_cflush_ppc 0135E65C
                                .d_complete_ppc 0135E688                                .d_master_ppc 0135E7B4
                                .d_slave_ppc 0135E974                                .d_unmask_ppc 0135EBA4
                                .d_mask_ppc 0135EC40                                .d_clear_ppc 0135ECD8
                                .d_init_ppc 0135ED8C                                .vm_att.glink 0135EF88
                                .lock_alloc.glink 0135EFB0                                .simple_lock_init.glink 0135EFD8
                                .vm_det.glink 0135F000                                .pincode.glink 0135F028
                                .bcopy 0135F060                                .copystr 0135F238
                                .errsave.glink 0135F2E0                                .xmemdma_ppc.glink 0135F308
                                .xmemqra.glink 0135F330                                .xmemacc.glink 0135F358
00 KERNEXT FUNCTION range [0135F6B4 0135F974] 10 entries
01 KERNEXT FUNCTION range [0135E120 0135F370] 24 entries
(0)> dc .lsa_ if name is not unique
Ambiguous: [kernext function name cache]

```

```

0135F6B4 .lsa_pos_unlock
0135F6E4 .lsa_pos_lock
0135F738 .lsa_config
(0)> expected symbol or address
(0)> dc .lsa_config 11 display code
.lsa_config+000000 stmw r29,FFFFFFF4(stkp)
.lsa_config+000004 mflr r0
.lsa_config+000008 ori r31,r3,0
.lsa_config+00000C stw r0,8(stkp)
.lsa_config+000010 stwu stkp,FFFFFFF0(stkp)
.lsa_config+000014 li r30,0
.lsa_config+000018 lwz r3,C(toc)
.lsa_config+00001C li r4,0
.lsa_config+000020 bl <.lockl.glink>
.lsa_config+000024 lwz toc,14(stkp)
.lsa_config+000028 lwz r29,14(toc)
(0)> dc .lockl.glink 6 display glink code
.lockl.glink+000000 lwz r12,10(toc)
.lockl.glink+000004 stw toc,14(stkp)
.lockl.glink+000008 lwz r0,0(r12)
.lockl.glink+00000C lwz toc,4(r12)
.lockl.glink+000010 mtctr r0
.lockl.glink+000014 bcctr

```

export table Subcommand

The **exp** subcommand may be used to look for an exported symbol address, or to display the export list. By default all the export list is printed, but it is possible to specify a string as prefix. Notice that export tables can be paged out.

Example

```

KDB(0)> exp list export table
000814D4 pio_assist
019A7708 puthere
0007BE90 vmminfo
00081FD4 socket
01A28A50 tcp_input
01A28BFC in_pcb_hash_del
019A78E8 adjmsg
0000BAB8 execexit
00325138 loif
01980874 lvm_kp_tid
000816E4 ns_detach
019A7930 mps_wakeup
01A28C50 ip_forward
00081E60 ksettickd
000810AC uiomove
000811EC blkflush
0018D97C setpriv
01A5CD38 clntkudp_init
000820D0 soqremque
00178824 devtosth
00081984 rtinithead
01A5CD8C xdr_rmtcall_args
(0)> more (^C to quit) ? ^C interrupt
KDB(0)> exp send search in export table
00081F5C sendmsg
00081F80 sendto
00081F74 send
KDB(0)>

```

Address Translation Subcommands for the KDB Kernel Debugger and kdb Command

tr and tv Subcommands

The **tr** and **tv** subcommands may be used to decode the MMU translation. **tr** is a short format, **tv** gives all information.

The following example applies on POWER PC achitecture, see POWER PC Operating Environment Architecture book III to have more information.

On **tv** subcommand, all double hashed entries are dumped, when the entry matches the specified effective address, corresponding physical address and protections are displayed. Page protection (**K** and **PP** bits) is given according to current segment register value and current machine state register value.

Example

```
KDB(0)> tr @iar physical address of current instruction
Physical Address = 001C5BA0
KDB(0)> tv @iar physical mapping of current instruction
vaddr 1C5BA0 sid 0 vpage 1C5 hash1 1C5
pte_cur_addr B0007140 valid 1 vsid 0 hsel 0 avpi 0
rpn 1C5 refbit 1 modbit 1 wim 1 key 0
___ 001C5BA0 ___ K = 0 PP = 00 ==> read/write
pte_cur_addr B0007148 valid 1 vsid 101 hsel 0 avpi 0
rpn 3C4 refbit 0 modbit 0 wim 1 key 0
vaddr 1C5BA0 sid 0 vpage 1C5 hash2 1E3A
Physical Address = 001C5BA0
KDB(0)> tv __ublock physical mapping of current U block
vaddr 2FF3B400 sid 9BC vpage FF3B hash1 687
ppcpte_cur_addr B001A1C0 valid 1 sid 300 hsel 0 avpi 1
rpn 13F4 refbit 1 modbit 1 wimg 2 key 1
ppcpte_cur_addr B001A1C8 valid 1 sid 9BC hsel 0 avpi 3F
rpn BFD refbit 1 modbit 1 wimg 2 key 0
___ 00BFD400 ___ K = 0 PP = 00 ==> read/write
vaddr 2FF3B400 sid 9BC vpage FF3B hash2 978
ppcpte_cur_addr B0025E08 valid 1 sid 643 hsel 0 avpi 3F
rpn 18D3 refbit 1 modbit 1 wimg 2 key 0
Physical Address = 00BFD400
KDB(0)> tv fffc1960 physical mapping thru BATs
BAT mapping for FFFC1960
DBAT0 FFC0003A FFC0005F
  bepi 7FE0 brpn 7FE0 bl 001F v 1 wim 3 ks 1 kp 0 pp 2 s 0
  eaddr = FFC00000, paddr = FFC00000 size = 4096 KBytes
KDB(0)> tv abcdef00 invalid mapping
Invalid Sid = 007FFFFFFF
KDB(0)> tv eeee0000 invalid mapping
vaddr EEEE0000 sid 505 vpage EEE0 hash1 BE5
vaddr EEEE0000 sid 505 vpage EEE0 hash2 141A
Invalid Address EEEE0000 !!!
On 620 machine
KDB(0)> set 11 64 bits printing
64_bit is true
KDB(0)> tv 2FF3AC88 physical mapping of a stack address
eaddr 2FF3AC88 sid F9F vpage FF3A hash1 A5
p64pte_cur_addr B0005280 sid_h 0 sid_l 0 avpi 0 hsel 0 valid 1
rpn_h 0 rpn_l A5 refbit 1 modbit 1 wimg 2 key 0
p64pte_cur_addr B0005290 sid_h 0 sid_l 81 avpi 0 hsel 0 valid 1
rpn_h 0 rpn_l 824 refbit 1 modbit 0 wimg 2 key 0
p64pte_cur_addr B00052A0 sid_h 0 sid_l 285 avpi 0 hsel 0 valid 1
rpn_h 0 rpn_l 5BE refbit 1 modbit 1 wimg 2 key 0
p64pte_cur_addr B00052B0 sid_h 0 sid_l F9F avpi 1F hsel 0 valid 1
```



```

rpn_h 0 rpn_l 1EC2 refbit 1 modbit 1 wimg 2 key 0
_____ 0000000001EC2C88 _____ K = 0 PP = 00 ==> read/write
eaddr 2FF3AC88 sid F9F vpage FF3A hash2 F5A
Physical Address = 0000000001EC2C88

```

The following example applies on POWER RS1 architecture.

Example

```

KDB(0)> tr __ublock physical address of current U block
Physical Address = 0779F000
KDB(0)> tv __ublock physical mapping of current U block
vaddr 2FF98000 sid 4008 vpage FF98 hash BF90 hat_addr B102FE40
pft_cur_addr B00779F0 nfr 779F sidpno 20047 valid 1 refbit 1 modbit 1 key 0
Physical Address = 0779F000
K = 0 PP = 00 ==> read/write
KDB(0)>

```

Process Subcommands for the KDB Kernel Debugger and kdb Command

ppda Subcommand

The **ppda** subcommand allows to display all **ppda** area with the '*' argument. Otherwise, the current or specified processor **ppda** is printed.

Example

```

KDB(1)> ppda *
          SLT CSA          CURTHREAD      SRR1      SRR0
ppda+000000 0 004ADEB0 thread+000178 4000D030 1002DC74
ppda+000300 1 004B8EB0 thread+000234 00009030 .ld_usecount+00045C
ppda+000600 2 004C3EB0 thread+0002F0 0000D030 D00012F0
ppda+000900 3 004CEEB0 thread+0003AC 0000D030 D00012F0
ppda+000C00 4 004D9EB0 thread+000468 0000F030 D00012F0
ppda+000F00 5 004E4EB0 thread+000524 0000D030 10019870
ppda+001200 6 004EFEB0 thread+0005E0 0000D030 D00012F0
ppda+001500 7 004FAEB0 thread+00069C 0000D030 D00012F0
KDB(1)> ppda current processor data area
Per Processor Data Area [000C0300]
csa.....004B8EB0 mstack.....004B7EB0
fpowner.....00000000 curthread.....E6000234
syscall.....0001879B intr.....E0100080
i_softis.....0000 i_softpri.....4000
prlvl.....05CB1000
ppda_pal[0].....00000000 ppda_pal[1].....00000000
ppda_pal[2].....00000000 ppda_pal[3].....00000000
phy_cpuid.....0001 ppda_fp_cr.....28222881
flih_save[0].....00000000 flih_save[1].....2FF3B338
flih_save[2].....002E65E0 flih_save[3].....00000003
flih_save[4].....00000002 flih_save[5].....00000006
flih_save[6].....002E6750 flih_save[7].....00000000
dsisr.....40000000 dsi_flag.....00000003
dar.....2FF9F884
dssave[0].....2FF3B2A0 dssave[1].....002E65E0
dssave[2].....00000000 dssave[3].....002A4B1C
dssave[4].....E6001ED8 dssave[5].....00002A33
dssave[6].....00002A33 dssave[7].....00000001
dssrr0.....0027D5AC dssrr1.....00009030
dssprg1.....2FF9F880 dsctr.....00000000
dslr.....0027D4CC dsxer.....20000000
dsmq.....00000000 pmapstk.....00212C80
pmapsave64.....00000000 pmapcsa.....00000000
schedtail[0].....00000000 schedtail[1].....00000000
schedtail[2].....00000000 schedtail[3].....00000000

```

```

cpuid.....00000001 stackfix.....00000000
lru.....00000000 vmflags.....00010000
sio.....00 reservation.....01
hint.....00 lock.....00
no_vwait.....00000000
scoreboard[0].....00000000
scoreboard[1].....00000000
scoreboard[2].....00000000
scoreboard[3].....00000000
scoreboard[4].....00000000
scoreboard[5].....00000000
scoreboard[6].....00000000
scoreboard[7].....00000000
intr_res1.....00000000 intr_res2.....00000000
mpc_pend.....00000000 idonelist.....00000000
affinity.....00000000 TB_ref_u.....003DC159
TB_ref_l.....28000000 sec_ref.....33CDD7B0
nsec_ref.....13EF2000 _fict.....00000000
decompress.....00000000 ppda_qio.....00000000
cs_sync.....00000000
ppda_perfmon_sv[0].....00000000 ppda_perfmon_sv[1].....00000000
thread_private.....00000000 cpu_priv_seg.....60017017
fp flih save[0].....00000000 fp flih save[1].....00000000
fp flih save[2].....00000000 fp flih save[3].....00000000
fp flih save[4].....00000000 fp flih save[5].....00000000
fp flih save[6].....00000000 fp flih save[7].....00000000
TIMER.....
t_free.....00000000 t_active.....05CB9080
t_freecnt.....00000000 trb_called.....00000000
systemer.....05CB9080 ticks_its.....00000051
ref_time.tv_sec.....33CDD7B1 ref_time.tv_nsec.....01DCDA38
time_delta.....00000000 time_adjusted.....05CB9080
wtimer.next.....05767068 wtimer.prev.....0B30B81C
wtimer.func.....000F2F0C wtimer.count.....00000000
wtimer.restart.....00000000 w_called.....00000000
trb_lock.....000C04F0 slock/slockp 00000000
KDB.....
flih_llsave[0].....00000000 flih_llsave[1].....2FF22FB8
flih_llsave[2].....00000000 flih_llsave[3].....00000000
flih_llsave[4].....00000000 flih_llsave[5].....00000000
flih_save[0].....00000000 flih_save[1].....00000000
flih_save[2].....00000000 csa.....001D4800
KDB(3)>

```

intr Subcommand

The **intr** subcommand prints the interrupt handler table, or the specified one.

Example

```

KDB(0)> intr interrupt handler table
          SLT INTRADDR HANDLER  TYPE LEVEL  PRIO BID  FLAGS
i_data+000068  1 055DF0A0 00000000 0000 00000003 0000 00000000 0000
i_data+000068  1 00364F88 00090584 0000 00000001 0000 00000000 0000
i_data+000068  1 003685B0 00090584 0001 00000008 0000 82000000 0000
i_data+000068  1 019E7D48 019E7BF0 0000 00000001 0000 820C0020 0010
i_data+0000E0 16 055DF060 00000000 0001 00000001 0000 82000080 0000
i_data+0000E0 16 00368718 000A24D8 0001 00000000 0000 82000080 0000
i_data+0000F0 18 055DF100 00000000 0001 00000000 0001 82080060 0010
i_data+0000F0 18 05B3BC00 01A55018 0001 00000002 0001 82080060 0010
i_data+000120 24 055DF0C0 00000000 0001 00000004 0000 82000000 0000
i_data+000120 24 003685B0 00090584 0001 00000008 0000 82000000 0000
i_data+000120 24 019E7D48 019E7BF0 0000 00000001 0000 820C0020 0010
i_data+000140 28 055DF160 00000000 0001 00000001 0003 820C0060 0010
i_data+000140 28 0A145000 01A741AC 0001 0000000C 0003 820C0060 0010
i_data+000150 30 055DF0E0 00000000 0001 00000000 0003 820C0020 0010

```

```

i_data+000150 30 055FC000 019E7AA8 0001 0000000E 0003 820C0020 0010
i_data+000160 32 055DF080 00000000 0001 00000002 0000 82100080 0000
i_data+000160 32 00368734 000A24D8 0001 00000000 0000 82100080 0000
i_data+0004E0 144 055DF020 00000000 0002 00000000 0000 00000000 0011
i_data+0004E0 144 00368560 000903B0 0002 00000002 0000 00000000 0011
i_data+000530 154 055DF040 00000000 0002 FFFFFFFF 000A 00000000 0011
i_data+000530 154 00368580 000903B0 0002 00000002 000A 00000000 0011
KDB(0)> intr 1 interrupt handler slot 1
          SLT INTRADDR HANDLER TYPE LEVEL PRI0 BID FLAGS

i_data+000068 1 055DF0A0 00000000 0000 00000003 0000 00000000 0000
i_data+000068 1 00364F88 00090584 0000 00000001 0000 00000000 0000
i_data+000068 1 003685B0 00090584 0001 00000008 0000 82000000 0000
i_data+000068 1 019E7D48 019E7BF0 0000 00000001 0000 820C0020 0010
KDB(0)> intr 00368560 interrupt handler address ..
addr..... 00368560 handler..... 000903B0 i_hwassist_int+000000
bid..... 00000000 bus_type..... 00000002 PLANAR
next..... 00000000 flags..... 00000011 NOT_SHARED MPSAFE
level..... 00000002 priority..... 00000000 INTMAX
i_count..... 00000014
KDB(0)>

```

mst Subcommand

The **mst** subcommand prints the current context (Machine State Save Area), or the specified one. Argument is given to identify a process/thread **mst** context. If **-a** option is specified, argument is an effective address, not a slot entry.

Example

```

KDB(0)> mst current mst
Machine State Save Area
iar : 0002599C msr : 00009030 cr : 20000000 lr : 000259B8
ctr : 000258EC xer : 00000000 mq : 00000000
r0 : 00000000 r1 : 2FF3B338 r2 : 002E65E0 r3 : 00000003 r4 : 00000002
r5 : 00000006 r6 : 002E6750 r7 : 00000000 r8 : DEADBEEF r9 : DEADBEEF
r10 : DEADBEEF r11 : 00000000 r12 : 00009030 r13 : DEADBEEF r14 : DEADBEEF
r15 : DEADBEEF r16 : DEADBEEF r17 : DEADBEEF r18 : DEADBEEF r19 : DEADBEEF
r20 : DEADBEEF r21 : DEADBEEF r22 : DEADBEEF r23 : DEADBEEF r24 : DEADBEEF
r25 : DEADBEEF r26 : DEADBEEF r27 : DEADBEEF r28 : 000034E0 r29 : 000C6158
r30 : 000C0578 r31 : 00005004
s0 : 00000000 s1 : 007FFFFFFF s2 : 0000F00F s3 : 007FFFFFFF s4 : 007FFFFFFF
s5 : 007FFFFFFF s6 : 007FFFFFFF s7 : 007FFFFFFF s8 : 007FFFFFFF s9 : 007FFFFFFF
s10 : 007FFFFFFF s11 : 007FFFFFFF s12 : 007FFFFFFF s13 : 0000C00C s14 : 00004004
s15 : 007FFFFFFF
prev 00000000 kjmpbuf 00000000 stackfix 00000000 intpri 0B
curid 00000306 sralloc E01E0000 ioalloc 00000000 backt 00
flags 00 tid 00000000 excp_type 00000000
fpscr 00000000 fpeu 00 fpinfo 00 fpscrx 00000000
o_iar 00000000 o_toc 00000000 o_arg1 00000000
excbranch 00000000 o_vaddr 00000000 mstext 00000000
Except :
csr 2FEC6B78 dsisr 40000000 bit set: DSISR_PFT
srval 000019DD dar 2FEC6B78 dsirr 00000106
KDB(0)> mst 1 slot 1 is thread+0000A0
Machine State Save Area
iar : 00038ED0 msr : 00001030 cr : 2A442424 lr : 00038ED0
ctr : 002BCC00 xer : 00000000 mq : 00000000
r0 : 60017017 r1 : 2FF3B300 r2 : 002E65E0 r3 : 00000000 r4 : 00000002
r5 : E60000BC r6 : 00000109 r7 : 00000000 r8 : 000C0300 r9 : 00000001
r10 : 2FF3B380 r11 : 00000000 r12 : 00001030 r13 : 00000001 r14 : 2FF22F54
r15 : 2FF22F5C r16 : DEADBEEF r17 : DEADBEEF r18 : 0000040F r19 : 00000000
r20 : 00000000 r21 : 00000003 r22 : 01000001 r23 : 00000001 r24 : 00000000
r25 : E600014C r26 : 000D1A08 r27 : 00000000 r28 : E3000160 r29 : E60000BC
r30 : 00000004 r31 : 00000004
s0 : 00000000 s1 : 007FFFFFFF s2 : 0000A00A s3 : 007FFFFFFF s4 : 007FFFFFFF
s5 : 007FFFFFFF s6 : 007FFFFFFF s7 : 007FFFFFFF s8 : 007FFFFFFF s9 : 007FFFFFFF

```

```

s10 : 007FFFFF s11 : 007FFFFF s12 : 007FFFFF s13 : 6001F01F s14 : 00004004
s15 : 60004024
prev      00000000 kjmpbuf  00000000 stackfix 2FF3B300 intpri  00
curid     00000001 sralloc  E01E0000 ioalloc  00000000 backt   00
flags     00 tid          00000000 excp_type 00000000
fpscr     00000000 fpeu      00 fpinfo      00 fpscrx    00000000
o_iar     00000000 o_toc     00000000 o_arg1     00000000
excbranch 00000000 o_vaddr   00000000 mstext     00000000
Except :
  csr  30002F00 dsisr 40000000 bit set: DSISR_PFT
  srval 6000A00A dar   20022000 dsirr 00000106
KDB(0)> set 11 64-bit printing mode
64 bit is true
KDB(0)> sw u select user context
KDB(0)> mst print user context
Machine State Save Area
iar  : 08000001000581D4 msr  : 80000004000D0B0 cr   : 84002222
lr   : 000000010000047C ctr  : 08000001000581D4 xer  : 00000000
mq   : 00000000 asr   : 0000000013619001
r0   : 08000001000581D4 r1   : 0FFFFFFFFFFFFFFF0 r2   : 080000018007BC80
r3   : 0000000000000064 r4   : 0000000000989680 r5   : 0000000000000000
r6   : 800000000000D0B0 r7   : 0000000000000000 r8   : 000000002FF9E008
r9   : 0000000013619001 r10  : 000000002FF3B010 r11  : 0000000000000000
r12  : 0800000180076A98 r13  : 0000000110003730 r14  : 0000000000000001
r15  : 00000000200FEB78 r16  : 00000000200FEB88 r17  : BADC0FFEE0DDF00D
r18  : BADC0FFEE0DDF00D r19  : BADC0FFEE0DDF00D r20  : BADC0FFEE0DDF00D
r21  : BADC0FFEE0DDF00D r22  : BADC0FFEE0DDF00D r23  : BADC0FFEE0DDF00D
r24  : BADC0FFEE0DDF00D r25  : BADC0FFEE0DDF00D r26  : BADC0FFEE0DDF00D
r27  : BADC0FFEE0DDF00D r28  : BADC0FFEE0DDF00D r29  : BADC0FFEE0DDF00D
r30  : BADC0FFEE0DDF00D r31  : 0000000110000688
s0   : 60000000 s1   : 007FFFFF s2   : 60010B68 s3   : 007FFFFF s4   : 007FFFFF
s5   : 007FFFFF s6   : 007FFFFF s7   : 007FFFFF s8   : 007FFFFF s9   : 007FFFFF
s10  : 007FFFFF s11  : 007FFFFF s12  : 007FFFFF s13  : 007FFFFF s14  : 007FFFFF
s15  : 007FFFFF
prev      00000000 kjmpbuf  00000000 stackfix 2FF3B2A0 intpri  00
curid     00006FBC sralloc  A0000000 ioalloc  00000000 backt   00
flags     00 tid          00000000 excp_type 00000000
fpscr     00000000 fpeu      00 fpinfo      00 fpscrx    00000000
o_iar     00000000 o_toc     00000000 o_arg1     00000000
excbranch 00000000 o_vaddr   00000000 mstext     00062C08
Except : dar  08000001000581D4
KDB(0)>

```

proc Subcommand

The **p** subcommand prints the proc table. The **'*** argument specifies all the table, no argument specifies the current process. Output is decimal or hexadecimal, according to the **hexadecimal_wanted** toggle value. The current process is shown with a **"*"**.

Example

```

KDB(0)> p * print proc table
          SLOT NAME      STATE  PID  PPID  PGRP  UID  EUID  ADSPACE
proc+000000 0 swapper ACTIVE 00000 00000 00000 00000 00000 00001C07
proc+000100 1 init    ACTIVE 00001 00000 00000 00000 00000 00001405
proc+000200 2*wait   ACTIVE 00204 00000 00000 00000 00000 00002008
proc+000300 3 wait   ACTIVE 00306 00000 00000 00000 00000 00002409
proc+000400 4 wait   ACTIVE 00408 00000 00000 00000 00000 0000280A
proc+000500 5 wait   ACTIVE 0050A 00000 00000 00000 00000 00002C0B
proc+000600 6 wait   ACTIVE 0060C 00000 00000 00000 00000 0000300C
proc+000700 7 wait   ACTIVE 0070E 00000 00000 00000 00000 0000340D
proc+000800 8 wait   ACTIVE 00810 00000 00000 00000 00000 0000380E
proc+000900 9 wait   ACTIVE 00912 00000 00000 00000 00000 00003C0F
proc+000A00 10 lrud  ACTIVE 00A14 00000 00000 00000 00000 00004010
proc+000B00 11 netm  ACTIVE 00B16 00000 00000 00000 00000 00001806
proc+000C00 12 gil   ACTIVE 00C18 00000 00000 00000 00000 00004C13

```

```

proc+000F00 15 lvmb ACTIVE 00F70 00000 00D68 00000 00000 00004832
proc+001000 16 biod ACTIVE 01070 02066 02066 00000 00000 000021A8
proc+001100 17 biod ACTIVE 0116E 02066 02066 00000 00000 000011A4
proc+001200 18 errdemon ACTIVE 01220 00001 01220 00000 00000 00001104
proc+001300 19 dump ACTIVE 01306 00001 00ECC 00000 00000 00005C77
proc+001400 20 syncd ACTIVE 01418 00001 00ECC 00000 00000 00000D03
proc+001500 21 biod ACTIVE 0156C 02066 02066 00000 00000 000001A0
KDB(0)> p 21 print process slot 21
      SLOT NAME      STATE  PID  PPID  PGRP  UID  EUID  ADSPACE
proc+001500 21 biod ACTIVE 0156C 02066 02066 00000 00000 000001A0
NAME..... biod
STATE..... stat :07..... xstat :0000
FLAGS..... flag :00040001 LOAD ORPHANPGRP
..... int :00000000
..... atomic:00000000
LINKS..... child :00000000
..... siblings :E3001800 proc+001800
..... uid1 :E3001500 proc+001500
..... ganchor :00000000
THREAD.... threadlist :E6001200 thread+001200
..... threadcount:0001..... active :0001
..... suspended :0000..... terminating:0000
..... local :0000
SCHEDULE... nice : 20 sched_pri :127
DISPATCH... pevent :00000000
..... synch :FFFFFFFF
IDENTIFIER. uid :00000000..... suid :00000000
..... pid :0000156C..... ppid :00002066
(0)> more (^C to quit) ? continue
..... sid :00002066..... pgrp :00002066
MISC..... lock :00000000..... kstackseg :007FFFFF
..... adspace :000001A0..... ipc :00000000
..... pgrp1 :E3001800 proc+001800
..... tty1 :00000000
..... dblist :00000000
..... dbnext :00000000
SIGNAL.... pending :
..... sigignore: URG IO WINCH PWR
..... sigcatch : TERM USR1 USR2
STATISTICS. page size :00000000..... pctcpu :00000000
..... auditmask :00000000
..... minflt :00000004..... majflt :00000000
SCHEDULER.. repage :00000000..... sched_count:00000000
..... sched_next :00000000
..... sched_back :00000000
..... cpticks :0000..... msgcnt :0000
..... majfltsec :00000000

```

THE FOLLOWING EXAMPLE SHOWS HOW TO FIND A THREAD THRU THE PROCESS TABLE.

The initial problem was that many threads are waiting for ever.

This example shows how to point the failing process:

KDB(6)> th -w WPGIN threads waiting for VMM resources

```

      SLOT NAME      STATE  TID  PRI  CPUID  CPU  FLAGS  WCHAN
thread+000780 10 lrud SLEEP 00A15 010 000 00001004 vmmddseg+69C84D0
thread+0012C0 25 dtlogin SLEEP 01961 03C 000 00000000 vmmddseg+69C8670
thread+001500 28 cnsview SLEEP 01C71 03C 000 00000004 vmmddseg+69C8670
thread+00B1C0 237 jfsz SLEEP 0EDCD 032 000 00001000 vm_zqevent+000000
thread+00C240 259 jfsc SLEEP 10303 01E 000 00001000 _$STATIC+000110
thread+00E940 311 rm SLEEP 137C3 03C 000 00000000 vmmddseg+69C8670
thread+012300 388 touch SLEEP 1843B 03C 000 00000000 vmmddseg+69C8670
...
thread+0D0F80 4458 link_fil SLEEP 116A39 03C 000 00000000 vmmddseg+69C9C74
thread+0DC140 4695 sync SLEEP 1257BB 03C 000 00000000 vmmddseg+69C8670
thread+0DD280 4718 touch SLEEP 126E57 03C 000 00000000 vmmddseg+69C8670
thread+0E5A40 4899 renamer SLEEP 132315 03C 000 00000000 vmmddseg+69C8670
thread+0EE140 5079 renamer SLEEP 13D7C3 03C 000 00000000 vmmddseg+69C8670
thread+0F03C0 5125 renamer SLEEP 1405B7 03C 000 00000000 vmmddseg+69C8670
thread+0FC540 5383 renamer SLEEP 15072F 03C 000 00000000 vmmddseg+69C8670

```

```

thread+101AC0 5497 renamer SLEEP 157909 03C      000 00000000 vmmidseg+69C8670
thread+10D280 5742 rm      SLEEP 166E37 03C      000 00000000 vmmidseg+69C8670
KDB(6)> vmwait vmmidseg+69C8670 VMM resource
VMM Wait Info
Waiting on transactions to end to forward the log
KDB(6)> vmwait vmmidseg+69C9C74 VMM resource
VMM Wait Info
Waiting on transaction block number 00000057
KDB(6)> tblk 87 print transaction block number
  @tblk[87] vmmidseg +69C9C3C
logtid.... 002C77CF next..... 00000064 tid..... 00000057 flag..... 00000000
cpn..... 00000000 ceor..... 00000000 cxor..... 00000000 csn..... 00000000
waitsid... 00000000 waitline.. 00000000 locker.... 00000000 lsidx.... 00000AB3
logage.... 00B71704 gcwait.... FFFFFFFF waitors... E60D0F80 cqnext.... 00000000
TID is registered in __ublock, at page offset 0x6a0.
Search in physical memory TID 0x00000057.
The search is limited at this page offset.
KDB(6)> findp 6A0 00000057 ffffffff 1000 physical search
0AFC86A0: 00000057 00000000 00000000 00000000
KDB(6)> pft 1 print page frame information
Enter the page frame number (in hex): 0AFC8
VMM PFT Entry For Page Frame 0AFC8 of 7FF67
pte = B066F458, pvt = B202BF20, pft = B3A0F580
h/w hashed sid : 000164EA pno : 0000FF3B key : 0
source sid : 000164EA pno : 0000FF3B key : 0
> in use
> on scb list
> valid (h/w)
> referenced (pft/pvt/pte): 0/1/1
> modified (pft/pvt/pte): 0/1/1
page number in scb (pagex) : 0000FF3B
disk block number (dblock) : 00000000
next page on scb list (sidfwd) : FFFFFFFF
prev page on scb list (sidbwd) : 00051257
freefwd/waitlist (freefwd): 00000000
freebwd/logage/pincnt (freebwd): 00010000
out of order I/O (nonfifo): 0000
next frame i/o list (nextio) : 00000000
storage attributes (wimg) : 2
xmem hide count (xmencnt): 0
next page on s/w hash (next) : FFFFFFFF
List of alias entries (alist) : 0000FFFF
index in PDT (devid) : 0000
The Segment ID of __ublock is the ADSPACE of the process
KDB(6)> find proc 000164EA search this SID in the proc table
proc+10EB58: 000164EA E3173F00 00000000 00000000
KDB(6)> proc proc+10EB00 print the process entry
      SLOT NAME      STATE  PID PPID PGRP  UID EUID ADSPACE
proc+10EB00 4331 renamer ACTIVE 10EB98 D6282 065DE 00000 00000 000164EA
NAME..... renamer
STATE..... stat :07..... xstat :0000
FLAGS..... flag :00000001 LOAD
..... int :00000000
..... atomic:00000000
LINKS..... child :00000000
..... siblings :E3173F00 proc+173F00
..... uid1 :E310EB00 proc+10EB00
..... ganchor :00000000
THREAD..... threadlist :E60F2640 thread+0F2640
...
KDB(6)> sw thread+0F2640 switch to this thread
Switch to thread: <thread+0F2640>
KDB(6)> f look at the stack
thread+0F2640 STACK:
[000D4950]slock_instr_ppc+00045C (C0042BDF, 00000002 [??])
[000095AC].simple_lock+0000AC ()
[00202370]logmvc+00004C (??, ??, ??, ??)

```

```

[001C23F4]logafter+000108 (??, ??, ??)
[001C1CEC]commit2+0001FC (??)
[001C386C]finicom+0000C0 (??, ??)
[001C3BC0]comlist+0001CC (??, ??)
[0020D938]jfs_rename+0006EC (??, ??, ??, ??, ??, ??, ??)
[001CE794]vnode_rename+000038 (??, ??, ??, ??, ??, ??, ??)
[001DEFA4]rename+000398 (??, ??)
[000037D8].sys_call+000000 ()
[100004B4]main+0002DC (00000006, 2FF22A20)
[10000174].__start+00004C ()

```

thread Subcommand

The **th** subcommand prints the thread table. The **'*** argument specifies all the table, no argument specifies the current thread. Output is decimal or hexadecimal, according to the **hexadecimal_wanted** toggle value. The current thread is shown with a **"*"**.

The **'-w'** argument can be used to select threads waiting with the specified thread wtype, (WLOCK, WPGIN, ...).

Example

```

KDB(0)> th * print thread table
          SLOT NAME      STATE  TID PRI CPUID CPU FLAGS      WCHAN
thread+000000  0 swapper  SLEEP 00003 010      078 00001400
thread+0000A0  1 init     SLEEP 001F3 03C      000 00000400
thread+000140  2 wait    RUN   00205 07F 00000 078 00001004
thread+0001E0  3 wait    RUN   00307 07F 00001 078 00001004
thread+000280  4 netm    SLEEP 00409 024      000 00001004
thread+000320  5 gil     SLEEP 0050B 025      000 00001004
thread+0003C0  6 gil     SLEEP 0060D 025      000 00001004 netisr_servers+000000
thread+000460  7 gil     SLEEP 0070F 025      000 00001004 netisr_servers+000000
thread+000500  8 gil     SLEEP 00811 025      001 00001004 netisr_servers+000000
thread+0005A0  9 gil     SLEEP 00913 025      000 00001004 netisr_servers+000000
thread+0006E0 11 errdemon SLEEP 00B01 03C      000 00000000 errrc+000008
thread+000780 12 syncd   SLEEP 00CF9 03C      005 00000000
thread+000820 13 lvmb    SLEEP 00D97 03C      000 00001004
thread+0008C0 14 cpio    SLEEP 00EC3 040      007 00000000 054FB000
thread+000960 15 sh      SLEEP 00FAF 03C      000 00000400
thread+000A00 16 getty   SLEEP 01065 03C      000 00000420 0563525C
thread+000AA0 17 ksh     SLEEP 01163 03C      000 00000420 05BA0E44
thread+000B40 18 sh      SLEEP 01279 03C      000 00000400
thread+000BE0 19 find    SLEEP 013B1 041      001 00000000
thread+000C80 20 ksh     SLEEP 014FB 040      000 00000400
KDB(0)> th print current thread
          SLOT NAME      STATE  TID PRI CPUID CPU FLAGS      WCHAN
thread+0159C0 461*ksh   RUN   1CDC9 03D      003 00000000
NAME..... ksh
FLAGS.....
WTYPE..... NOWAIT
.....stackp64 :00000000 .....stackp :2FF1E5A0
.....state :00000002 .....wtype :00000000
.....suspend :00000001 .....flags :00000000
.....atomic :00000000
DATA.....
.....procp :E3014400 <proc+014400>
.....userp :2FF3B6C0 <__ublock+0002C0>
.....uthreadp :2FF3B400 <__ublock+000000>
THREAD LINK.....
.....prevthread :E60159C0 <thread+0159C0>
.....nextthread :E60159C0 <thread+0159C0>
SLEEP LOCK.....
.....ulock64 :00000000 .....ulock :00000000
.....wchan :00000000 .....wchan1 :00000000
.....wchan1sid :00000000 .....wchan1offset :00000000

```

```

(3)> more (^C to quit) ? continue
.....wchan2 :00000000 .....swchan :00000000
.....eventlist :00000000 .....result :00000000
.....polevel :00000000 .....pevent :00000000
.....wevent :00000000 .....slist :00000000
.....lockcount :00000002
DISPATCH.....
.....ticks :00000000 .....prior :E60159C0
.....next :E60159C0 .....synch :FFFFFFF
.....dispcnt :00000003 .....fpuct :00000000
SCHEDULER.....
.....cpuid :FFFFFFF .....scpuuid :FFFFFFF
.....affinity :00000001 .....pri :0000003C
.....policy :00000000 .....cpu :00000000
.....lockpri :0000003D .....wakepri :0000007F
.....time :000000FF .....sav_pri :0000003C
SIGNAL.....
.....cursig :00000000
.....(pending) sig :
.....sigmask :
.....scp64 :00000000 .....scp :00000000
MISC.....
.....graphics :00000000 .....cancel :00000000
(3)> more (^C to quit) ? continue
.....lockowner :00000000 .....boosted :00000000
.....tsleep :FFFFFFF
.....userdata64 :00000000 .....userdata :00000000
KDB(0)> th -w print -w usage
Missing wtype:
NOWAIT
WEVENT
WLOCK
WTIMER
WCPU
WPGIN
WPGOUT
WPLock
WFREEF
WMEM
WLOCKREAD
WUEXCEPT
KDB(0)> th -w WPGIN print threads waiting for page-in
      SLOT NAME      STATE  TID PRI CPUID CPU FLAGS   WCHAN
thread+000600    8 lrud    SLEEP 00811 010    000 00001004 vmdmseg+69C84D0
thread+000E40   19 syncd    SLEEP 01329 03D    003 00000000 vmdmseg+69D1630
thread+013440  411 oracle    SLEEP 19B75 03D    002 00000000 vmdmseg+69F171C
thread+013500  412 oracle    SLEEP 19C77 03F    006 00000000 vmdmseg+69F13A8
thread+022740  735 rts32    SLEEP 2DF7F 03F    007 00000000 vmdmseg+3A9A5B8
KDB(0)> vmwait vmdmseg+69C84D0 print VMM resource the thread is waiting for
VMM Wait Info
Waiting on lru daemon anchor
KDB(0)> vmwait vmdmseg+69D1630 print VMM resource the thread is waiting for
VMM Wait Info
Waiting on segment I/O level (v_iowait), sidx = 00000124
KDB(0)> vmwait vmdmseg+69F171C print VMM resource the thread is waiting for
VMM Wait Info
Waiting on segment I/O level (v_iowait), sidx = 000008AF
KDB(0)> vmwait vmdmseg+69F13A8 print VMM resource the thread is waiting for
VMM Wait Info
Waiting on segment I/O level (v_iowait), sidx = 000008A2
KDB(0)> vmwait vmdmseg+3A9A5B8 print VMM resource the thread is waiting for
VMM Wait Info
Waiting on page frame number 0000DE1E
KDB(1)> th -w WLOCK print threads waiting for locks
      SLOT NAME      STATE  TID PRI CPUID CPU FLAGS   WCHAN
thread+0000C0    1 init    SLEEP 001BD 03C    000 00000000 cred_lock+000000
                                                    lockhsque+000020

```



```

thread+000900 12 cron SLEEP 00C57 03C 000 00000000 cred_lock+000000
lockhsque+000020
thread+000B40 15 inetd SLEEP 00FB7 03C 000 00000000 cred_lock+000000
lockhsque+000020
thread+000CC0 17 mirrord SLEEP 01107 03C 000 00000000 cred_lock+000000
lockhsque+000020
thread+000F00 20 sendmail SLEEP 014A5 03C 000 00000004 cred_lock+000000
lockhsque+000020
thread+013F80 426 getty SLEEP 1AA6F 03C 000 00000000 cred_lock+000000
lockhsque+000020
thread+014340 431 diagd SLEEP 1AF8F 03C 000 00000000 proc_tbl_lock+000000
lockhsque+0000F8
thread+014400 432 pd_watch SLEEP 1B091 03C 000 00000000 proc_tbl_lock+000000
lockhsque+0000F8
thread+015000 448 stress_m SLEEP 1C08B 028 000 00000000 cred_lock+000000
lockhsque+000020
thread+018780 522 stresser SLEEP 20AF1 03C 000 00000000 cred_lock+000000
lockhsque+000020
thread+018CC0 529 pcomp SLEEP 21165 03C 000 00000000 cred_lock+000000
lockhsque+000020
thread+01B6C0 585 EXP_TEST SLEEP 24943 03C 000 00000000 cred_lock+000000
lockhsque+000020
thread+01C2C0 601 cres SLEEP 25957 03C 000 00000000 cred_lock+000000
lockhsque+000020
thread+022500 732 rsh SLEEP 2DC25 03C 000 00000000 cred_lock+000000
lockhsque+000020
thread+02A240 899 rcp SLEEP 383FB 03C 000 00000000 cred_lock+000000
lockhsque+000020
thread+02C580 946 ps SLEEP 3B223 03C 000 00000000 proc_tbl_lock+000000
lockhsque+0000F8
thread+02D900 972 rsh SLEEP 3CC29 03C 000 00000000 cred_lock+000000
lockhsque+000020
thread+02DD80 978 xlCcode SLEEP 3D227 03C 000 00000000 cred_lock+000000
lockhsque+000020
thread+02ED40 999 tty_benc SLEEP 3E7A7 03C 000 00000000 cred_lock+000000
lockhsque+000020
thread+02F100 1004 tty_benc SLEEP 3ECF3 03C 000 00000000 cred_lock+000000
lockhsque+000020

(1)> more (^C to quit) ? continue
      SLOT NAME STATE TID PRI CPUID CPU FLAGS WCHAN
thread+02F400 1008 tty_benc SLEEP 3F097 03C 000 00000000 cred_lock+000000
lockhsque+000020
thread+02F700 1012 ksh SLEEP 3F403 03C 000 00000000 cred_lock+000000
lockhsque+000020
thread+02F940 1015 tty_benc SLEEP 3F745 03C 000 00000000 cred_lock+000000
lockhsque+000020
thread+02FA00 1016 tty_benc SLEEP 3F869 03C 000 00000000 cred_lock+000000
lockhsque+000020
thread+02FE80 1022 tty_benc SLEEP 3FECB 03C 000 00000000 cred_lock+000000
lockhsque+000020
thread+02FF40 1023 tty_benc SLEEP 3FFF5 03C 000 00000000 cred_lock+000000
lockhsque+000020
thread+030240 1027 rshd SLEEP 403F3 03C 000 00000000 cred_lock+000000
lockhsque+000020
thread+030300 1028 bsh SLEEP 404FF 03C 000 00000000 cred_lock+000000
lockhsque+000020
thread+0303C0 1029 sh SLEEP 40505 03C 000 00000000 cred_lock+000000
lockhsque+000020

KDB(1)> slk cred_lock+000000 print lock information
Simple lock name: cred_lock
      _slock: 400401FD WAITING thread_owner: 00401FD
KDB(1)> slk proc_tbl_lock+000000 print lock information
Simple lock name: proc_tbl_lock
      _slock: 400401FD WAITING thread_owner: 00401FD
KDB(1)>

```

ttid and tpid Subcommands

The **ttid** subcommand prints the thread entry selected by thread id. Default is the current thread. The **tpid** subcommand prints all thread entries selected by process id. Default is the current process. Id input is decima hexadecimal, according to the **hexadecimal_wanted** toggle value.

Example

```
KDB(4)> p * print process table
          SLOT NAME      STATE   PID  PPID  PGRP   UID  EUID  ADSPACE
...
proc+000100    1 init      ACTIVE 00001 00000 00000 00000 00000 0000A005
...
proc+000C00   12 gil      ACTIVE 00C18 00000 00000 00000 00000 00026013
...
KDB(4)> tpid 1 print thread(s) of process pid 1
          SLOT NAME      STATE   TID  PRI  CPUID  CPU  FLAGS   WCHAN
thread+0000C0    1 init      SLEEP 001D9 03C      000 00000400
KDB(4)> tpid 00C18 print thread(s) of process pid 0xc18
          SLOT NAME      STATE   TID  PRI  CPUID  CPU  FLAGS   WCHAN
thread+000900   12 gil      SLEEP 00C19 025      000 00001004
thread+000C00   16 gil      SLEEP 01021 025 00000 000 00003004 netisr_servers+000000
thread+000B40   15 gil      SLEEP 00F1F 025 00000 000 00003004 netisr_servers+000000
thread+000A80   14 gil      SLEEP 00E1D 025 00000 000 00003004 netisr_servers+000000
thread+0009C0   13 gil      SLEEP 00D1B 025 00000 000 00003004 netisr_servers+000000
KDB(4)> ttid 001D9 print thread with tid 0x1d9
          SLOT NAME      STATE   TID  PRI  CPUID  CPU  FLAGS   WCHAN
thread+0000C0    1 init      SLEEP 001D9 03C      000 00000400
NAME..... init
FLAGS..... WAKEONSIG
WTYPE..... WEVENT
.....stackp64 :00000000 .....stackp :2FF22DC0
.....state :00000003 .....wtype :00000001
.....suspend :00000001 .....flags :00000400
.....atomic :00000000
DATA.....
.....procp :E3000100 <proc+000100>
.....userp :2FF3B6C0 <__ublock+0002C0>
.....uthreadp :2FF3B400 <__ublock+000000>
THREAD LINK.....
.....prevthread :E60000C0 <thread+0000C0>
.....nextthread :E60000C0 <thread+0000C0>
SLEEP LOCK.....
.....ulock64 :00000000 .....ulock :00000000
.....wchan :00000000 .....wchan1 :00000000
.....wchan1sid :00000000 .....wchan1offset :01AB5A58
(4)> more (^C to quit) ? continue
.....wchan2 :00000000 .....swchan :00000000
.....eventlist :00000000 .....result :00000000
.....polevel :000000AF .....pevent :00000000
.....wevent :00000004 .....slist :00000000
.....lockcount :00000000
DISPATCH.....
.....ticks :00000000 .....prior :E60000C0
.....next :E60000C0 .....synch :FFFFFFFF
.....dispct :000008F6 .....fpuct :00000000
SCHEDULER.....
.....cpuid :FFFFFFFF .....scpuid :FFFFFFFF
.....affinity :00000001 .....pri :0000003C
.....policy :00000000 .....cpu :00000000
.....lockpri :0000003D .....wakepri :0000007F
.....time :000000FF .....sav_pri :0000003C
SIGNAL.....
.....cursig :00000000
.....(pending) sig :
.....sigmask :
```

```

.....scp64 :00000000 .....scp :00000000
MISC.....
.....graphics :00000000 .....cancel :00000000
(4)> more (^C to quit) ? continue
.....lockowner :E60042C0 .....boosted :00000000
.....tsleep :FFFFFFF
.....userdata64 :00000000 .....userdata :00000000

```

runq, lockq, and sleepq Subcommands

The **rq** subcommand prints run queues, **lq** subcommand prints lock queues, and **sq** subcommand prints sleep queues. A specific queue may be displayed.

Example

```

KDB(0)> rq print run queues
          BUCKET HEAD          COUNT
thread_run+000100  65  thread+009060  1
thread_run+000104  66  thread+008660  4
thread_run+000108  67  thread+001D60  5
thread_run+0001FC 128  thread+000140  2
KDB(0)> rq 67 print run queue slot 67
RUNQ ENTRY( 67): thread_run+000108
          SLOT NAME          STATE  TID PRI CPUID CPU FLAGS  WCHAN
thread+001D60  47 ps          RUN   02FF7 042      001 00000000
thread+0071C0 182 diff        RUN   0B6ED 042      005 00000000
thread+005460 135 cpio         RUN   08773 042      001 00000000
thread+004060 103 ed           RUN   06761 042      001 00000000
thread+004100 104 nroff       RUN   068D7 042      002 00000000
KDB(0)> lq print lock queues
          BUCKET HEAD          COUNT
lockhsque+0000D0  53  thread+0B04C0  31
lockhsque+0001FC 128  thread+000000  1
KDB(0)> lq 53 print lock queue slot 53
LOCKHSQUE ENTRY( 53): lockhsque+0000D0
          SLOT NAME          STATE  TID PRI CPUID CPU FLAGS  WCHAN
thread+0B04C0 3761 ksh          SLEEP EB14F 04A      01D 00000000 09AB9410 lockhsque+0000D0
thread+093CC0 3153 renamer    SLEEP C5175 03D      003 00000000 09AB9410 lockhsque+0000D0
thread+074280 2478 renamer    SLEEP 9AEF9 03D      003 00000000 09AB9410 lockhsque+0000D0
...
thread+088740 2911 ksh          SLEEP B5FB5 03F      006 00000000 09AB9410 lockhsque+0000D0
thread+029DC0  893 inetd        SLEEP 37DB7 03C      000 00000000 09AB9410 lockhsque+0000D0
thread+0B4E40 3859 ksh          SLEEP F139F 04A      01D 00000000 09AB9410 lockhsque+0000D0
KDB(0)>

```

user Subcommand

The **u** subcommand prints the u-block of the current process, or the specified one. Argument is given to identify a process/thread **u** block.

Usage: u [-ad] [-cr] [-f] [-s] [-ru] [-t] [-ut] [-64] <slot/eaddr>

- **-ad** to print user adspace information only
- **-cr** to print credential information only
- **-f** to print file information only
- **-ru** to print profiling/resource/limit information only
- **-s** to print signal information only
- **-t** to print timers information only
- **>-ut** to print user thread information only
- **-64** to print 64-bit user information only

Example

```
KDB(0)> u -ut print current user thread block
User thread context [2FF3B400]:
  save.... @ 2FF3B400  fpr..... @ 2FF3B550
Uthread System call state:
  msr64.....00000000  msr.....0000D0B0
  errnopp64..00000000  errnopp....200FEFE8  error.....00
  scsave[0]..2004A474  scsave[1]..00000020  scsave[2]..20007B48
  scsave[3]..2FF22AA0  scsave[4]..00000014  scsave[5]..20006B68
  scsave[6]..2004A7B4  scsave[7]..2004A474
  kstack.....2FF3B400  audsvc.....00000000
  flags:
Uthread Miscellaneous stuff:
  fstid....00000000  ioctlr...00000000  selchn....00000000
  link.....00000000  loginfo...00000000
  fselchn...00000000  selbuc.....0000
  context64.00000000  context...00000000
  sigssz64..00000000  sigssz....00000000
  stkb64....00000000  stkb.....00000000
  jfscr....00000000
Uthread Signal management:
  sigsp64...00000000  sigsp....00000000
  code.....00000000  oldmask...0000000000000000
Thread timers:
  timer[0].....00000000
KDB(0)> u -64 print current 64-bit user part of ublock
64-bit process context [2FF7D000]:
  stab..... @ 2FF7D000
STAB:  esid          vsid          esid          vsid
      0 09000000000000B0 000000000714E000  1 0000000000000000 0000000000000000
      16 00000000200000B0 000000000AA75000  17 0000000000000000 0000000000000000
      80 09001000A00000B0 000000000CA99000  81 0000000000000000 0000000000000000
     104 00000000D00000B0 000000000D95B000  105 0000000000000000 0000000000000000
     128 00000001000000B0 0000000004288000  129 0000000000000000 0000000000000000
     136 00000001100000B0 000000000C298000  137 0000000000000000 0000000000000000
     160 09002001400000B0 000000000E15C000  161 08002001400000B0 0000000008290000
     248 09FFFFFFF00000B0 0000000002945000  249 08FFFFFFF00000B0 0000000001A83000
     250 0FFFFFFF000000B0 000000000BA97000  251 0000000000000000 0000000000000000
     254 0000000000000000 0000000000000000  255 0000000000000000 0000000000000000
  stablock..... @ 2FF7E000  stablock.....00000000
  mstext.mst64.. @ 2FF7E008  mstext.remaps. @ 2FF7E140
SNODE... @ 2FF7E3C8
  origin...28020000  freeind..FFFFFFF  nextind..00000002
  maxind...0006DD82  size.....00000094
UNODE... @ 2FF7E3E0
  origin...2BFA1000  freeind..FFFFFFF  nextind..0000000E
  maxind...000D4393  size.....0000004C
  maxbreak...00000001100005B8  minbreak...00000001100005B8
  maxdata....0000000000000000  exitexec...00000000
  brkseg....00000011  stkseg....FFFFFFF
KDB(0)> u -f 18 print file descriptor table of thread slot 18
  fdfree[0].00000000  fdfree[1].00000000  fdfree[2].00000000
  maxofile..00000008  freefile..00000000
  fd_lock...2FF3C188  slock/slockp 00000000
File descriptor table at..2FF3C1A0:
  fd 3 fp..100000C0 count..00000000 flags. ALLOCATED
  fd 4 fp..10000180 count..00000001 flags. ALLOCATED
  fd 5 fp..100003C0 count..00000000 flags. ALLOCATED
  fd 6 fp..100005A0 count..00000000 flags. ALLOCATED
  fd 7 fp..10000600 count..00000000 flags. FDLOCK ALLOCATED
Rest of File Descriptor Table empty or paged out.
```

LVM Subcommands for the KDB Kernel Debugger and kdb Command

pbuf Subcommand

The **pbuf** subcommand prints physical buffer information.

Example

```
(0)> pbuf 0ACA4500
PBUF..... 0ACA4500
pb@..... 0ACA4500 pb_lbuf..... 0A5B8318
pb_sched..... 01B64880 pb_pvol..... 05770000
pb_bad..... 00000000 pb_start..... 00133460
pb_mirror..... 00000000 pb_miravoid.... 00000000
pb_mirbad..... 00000000 pb_mirdone.... 00000000
pb_swretry..... 00000000 pb_type..... 00000000
pb_bbfixtype.... 00000000 pb_bbop..... 00000000
pb_bbstat..... 00000000 pb_whl_stop.... 00000000
pb_part..... 00000000 pb_bbcnt..... 00000000
pb_forw..... 0ACA45A0 pb_back..... 0ACA4460
stripe_next.... 0ACA4500 stripe_status.. 00000000
orig_addr..... 0C149000 orig_count.... 00001000
partial_stripe.. 00000000 first_issued... 00000001
orig_bflags.... 000C0000
(0)> buf 0A5B8318
      DEV      VNODE      BLKNO  FLAGS
0 0A5B8318 000A000B 00000000 0007A360 DONE MPSAFE MPSAFE_INITIAL
forw 0000C4C1 back 00000000 av_forw 0A5B98C0 av_back 00000000
blkno 0007A360 addr 0C149000 bcount 00001000 resid 00000000
error 00000000 work 00080000 options 00000000 event 00000000
iodone: v_pfind+0000000
start.tv_sec 00000000 start.tv_nsec 00000000
xmemd.aspace_id 00000000 xmemd.xm_flag 00000000 xmemd.xm_version 00000000
xmemd.subspace_id 0080CC5B xmemd.subspace_id2 00000000 xmemd.uaddr 00000000
(0)> pbuf * 0ACA4500
PBUF@ LBUF@ PVOL@ _DEV START STRIPE OR ADDR OR_COUNT
0ACA4500 0A5B8318 05770000 00120006 00133460 0ACA4500 0C149000 00001000
0ACA45A0 0AA64898 0A7DB000 00120000 001C71F0 0ACA45A0 0003E000 00001000
0ACA4640 0A323D10 05766000 00120004 00082FC0 0ACA4640 0A997000 00001000
0ACA46E0 0A5B97B8 05770000 00120006 001338C8 0AC95320 0C15C000 00001000
0ACB9400 0AA62630 0A7DB000 00120000 001851A0 0ACB9400 00054000 00001000
0ACB94A0 0AA65398 0A7BC000 00120001 001AD750 0ACB94A0 083E9000 00001000
0ACB9540 0AA62DC0 0A7DB000 00120000 00181150 0ACB9540 00000000 00002000
0ACA0000 0AA6CA20 0A7BC000 00120001 000F72BC 0ACA0000 00000000 00000800
0ACCD800 0AA64478 0A7DB000 00120000 001C7260 0ACCD800 00000000 00001000
0ACCD8A0 0A5B86E0 05770000 00120006 00133BA8 0ACCD8A0 0B796000 00002000
0ACCD940 0A31F210 05766000 00120004 0013B100 0ACCD940 00840000 00002000
0ACCD9E0 0AA6ADE8 0A7BC000 00120001 0006925C 0ACCD9E0 00000000 00000800
0ACCD8A0 0AA6C028 0A7BC000 00120001 000DA29C 0ACCD8A0 003FF000 00000800
0ACCD8B0 0A324DE8 05766000 00120004 0008ACE8 0ACCD8B0 0C151000 00001000
0ACCD8C0 0AA638C0 0A7DB000 00120000 00186228 0ACCD8C0 00000000 00001000
...
```

volgrp Subcommand

The **volgrp** subcommand prints volume groupe information. **volgrp** addresses are registered in the table, in DSDPTR field.

Example

```
(0)> devsw 0a
Slot address 0571E280
MAJ#00A OPEN CLOSE READ WRITE
01B44DE4 01B44470 01B43CD0 01B43C04
IOCTL STRATEGY TTY SELECT
01B42B18 .hd_strategy 00000000 .nodev
CONFIG PRINT DUMP MPX
```

```

01B413A0      .nodev      .hd_dump      .nodev
REVOKE        DSDPTR       SELPTR        OPTS
.nodev        05762000    00000000     0000000A
(0)> volgrp 05762000
VOLGRP..... 05762000
vg_lock..... FFFFFFFF partshift..... 0000000D
open_count... 00000013 flags..... 00000000
tot_io_cnt... 00000000 lvols@..... 05762010
pvols@..... 05762410 major_num..... 0000000A
vg_id..... 00920045 005BDB00 00000000 00000000
nextvg..... 00000000 opn_pin@..... 057624A8
von_pid..... 00000E78 nxtactvg..... 00000000
ca_freepw..... 00000000 ca_pvwmem..... 00000000
ca_hld@..... 057624D8 ca_pv_wrt@..... 057624E0
ca_inflt_cnt... 00000000 ca_size..... 00000000
ca_pvwblked... 00000000 mwc_rec..... 00000000
ca_part2..... 00000000 ca_lst..... 00000000
ca_hash@..... 057624F4 bcachwait..... FFFFFFFF
ecachwait..... FFFFFFFF wait_cnt..... 00000000
quorum_cnt... 00000002 wheel_idx..... 00000000
whl_seq_num... 00000000 sa_act_lst..... 00000000
sa_hld_lst... 00000000 vgsa_ptr..... 05776000
config_wait... FFFFFFFF sa_lbuf@..... 05762534
sa_pbuf@..... 0576258C sa_intlock@..... 0576262C
sa_intlock... E8003B80
conc_flags... 00000000 conc_msglock..... 00000000
vgsa_ts_prev.tv_sec.... 00000000 vgsa_ts_prev.tv_nsec... 00000000
vgsa_ts_merged.tv_sec.... 00000000 vgsa_ts_merged.tv_nsec.. 00000000
vgsa_spare_ptr..... 00000000 intr_notify..... 00000000
intr_ok..... 00000000 intr_tries..... 00000000
resv_tries... 00000000 sa_updated..... 00000000
re_lbuf@..... 05762660 re_pbuf@..... 057626B8
re_idx..... 00000000 re_finish..... 00000000
re_twice..... 00000000 re_marks..... 00000000
re_saved_marks..... 00000000 refresh_Q@..... 05762768
concsync_wd_pass@..... 05762770 concsync_wd_init@..... 05762788
concsync_wd_intr@..... 057627A0 concsync_terminate_Q@... 05762810
concsync_lockpart..... 00000000
conconconfig_lbuf@..... 0576281C conconconfig_wd@..... 05762874
conconconfig_wd_intr@..... 0576288C conconconfig_nodes..... 00000000
conconconfig_acknodes..... 00000000 conconconfig_nacknodes... 00000000
conconconfig_event..... 00000000 conconconfig_timeout..... 00000000
llc.flags..... 00000000 llc.ack..... 00000000
llc.nak..... 00000000 llc.timeout..... 00000000
llc.contention..... 00000000 llc.awakened..... 00000000
llc.wd@..... 05762920 llc.event..... 00000000
llc.arb_intlock..... 00000000 llc.arb_intlock@..... 0576293C
dd_conc_reset... 00000000 @timer_intlock..... 05762944
timer_intlock..... 00000000
@vg_intlock..... 05762948 vg_intlock..... E8003BA0
LVOL..... 05CC8400
work_Q..... 00000000 lv_status..... 00000000
lv_options..... 00000001 nparts..... 00000001
i_sched..... 00000000 nblocks..... 00040000
parts[0]..... 05706A00 pvol@ 05766000 dev 00120004 start 00000000
parts[1]..... 00000000
parts[2]..... 00000000
maxsize..... 00000000 tot_rds..... 00000000
complcnt..... 00000000 waitlist..... FFFFFFFF
stripe_exp.... 00000000 striping_width. 00000000
lv01_intlock... 00000000 lv01_intlock@.. 05CC8434
LVOL..... 05CC8440
work_Q..... 05780D00 lv_status..... 00000002
lv_options..... 00000190 nparts..... 00000001
i_sched..... 00000000 nblocks..... 00040000
parts[0]..... 05706000 pvol@ 05766000 dev 00120004 start 00065100
parts[1]..... 00000000

```

```

parts[2]..... 00000000
maxsize..... 00000200 tot_rds..... 00000000
complcnt..... 00000000 waitlist..... FFFFFFFF
stripe_exp..... 00000000 striping_width. 00000000
lvol_intlock... 00000000 lvol_intlock@.. 05CC8474
  WORK_Q@   BUF@   FLAGS   DEV   BLKNO   BADDR   BCOUNT   RESID   SID
05780D28 0A323580 000C8001 000A0001 00004A08 0FF3A000 00001000 00001000 0080C919
  WORK_Q@   BUF@   FLAGS   DEV   BLKNO   BADDR   BCOUNT   RESID   SID
05780D90 0A323738 000C0000 000A0001 00022420 0B783000 00001000 00001000 0080CC5B
05780D90 0A323D10 000C0000 000A0001 00022408 0B782000 00001000 00001000 0080CC5B
...
LVOL..... 0A752440
work_Q..... 0A82DD00 lv_status..... 00000002
lv_options..... 00000000 nparts..... 00000001
i_sched..... 00000000 nblocks..... 00002000
parts[0]..... 057222F0 pvol@ 0576C000 dev 00120005 start 000C7100
parts[1]..... 00000000
parts[2]..... 00000000
maxsize..... 00000200 tot_rds..... 00000000
complcnt..... 00000000 waitlist..... FFFFFFFF
stripe_exp..... 00000000 striping_width. 00000000
lvol_intlock... E80279C0 lvol_intlock@.. 0A752474

```

pvol Subcommand

The **pvol** subcommand prints physical volume information.

Example

```

(0)> pvol 05766000
PVOL..... 05766000
dev..... 00120004 xfcnt..... 00000003
armpos..... 00000000 pvstate..... 00000000
pvnum..... 00000000 vg_num..... 0000000A
fp..... 00429258 flags..... 00000000
num_bmdir_ent.... 00000000 fst_usr_blk..... 00001100
beg_relblk..... 001F5A7A next_relblk..... 001F5A7A
max_relblk..... 001F5B79 defect_tbl..... 05705500
ca_pv@..... 0576602C sa_area[0]@..... 05766034
sa_area[1]@..... 0576603C pv_pbuf@..... 05766044
conc_func..... 00000000 conc_msgseq..... 00000000
conc_msglen..... 00000000 conc_msgbuf@..... 057660F0
mirror_tur_cmd@.. 057660F8 mirror_wait_list. 00000000
ref_cmd@..... 057661A8 user_cmd@..... 05766254
refresh_intr@.... 05766300
concsync_cmd@.... 05766370 synchold_cmd@.... 0576641C
wd_cmd@..... 057664C8 concsync_intr.... 00000000
concsync_intr_next 00000000
config_cmd@..... 0576657C ack_cmd@..... 05766628
ack_idx..... 00000000 nak_cmd@..... 05767BAC
nak_idx..... 00000000 llc_cmd@..... 05769130
ppCmdTail..... 00000000 send_cmd_lock.... 00000000
send_cmd_lock@.... 057691E0

```

lvol Subcommand

The **lvol** subcommand prints logical volume information.

Example

```

(0)> lvol 05CC8440
LVOL..... 05CC8440
work_Q..... 05780D00 lv_status..... 00000002
lv_options..... 00000190 nparts..... 00000001
i_sched..... 00000000 nblocks..... 00044000
parts[0]..... 05706000 pvol@ 05766000 dev 00120004 start 00065100
parts[1]..... 00000000
parts[2]..... 00000000

```

```

maxsize..... 00000200 tot_rds..... 00000000
complcnt..... 00000000 waitlist..... FFFFFFFF
stripe_exp..... 00000000 striping_width. 00000000
lvol_intlock... 00000000 lvol_intlock@.. 05CC8474
WORK_Q@  BUF@  FLAGS  DEV  BLKNO  BADDR  BCOUNT  RESID  SID
05780D28 0A323580 000C8001 000A0001 00004A08 0FF3A000 00001000 00001000 0080C919
WORK_Q@  BUF@  FLAGS  DEV  BLKNO  BADDR  BCOUNT  RESID  SID
05780D90 0A323738 000C0000 000A0001 00022420 0B783000 00001000 00001000 0080CC5B
05780D90 0A323D10 000C0000 000A0001 00022408 0B782000 00001000 00001000 0080CC5B

```

SCSI Subcommands for the KDB Kernel Debugger and kdb Command

ascsi Subcommand

The `ascsi` subcommand prints adapter information.

Example

```

KDB(4)> lke 88 print kernel extension information
      ADDRESS  FILE FILESIZE  FLAGS MODULE NAME
88 05630600 01A2A640 00008680 00000262 /etc/drivers/ascsiddpin
le_flags..... TEXT DATAINTEXT DATA DATAEXISTS
le_fp..... 00000000
le_loadcount... 00000000
le_usecount.... 00000001
le_data/le_tid.. 01A32760 <-- address (adjusted to 0x27C0) needed to
le_datasize.... 00000560 initialize the ascsi command.
le_exports..... 0BC6B800
le_lex..... 00000000
le_defered..... 00000000
le_filename.... 05630644
le_ndepend..... 00000001
le_maxdepend... 00000001
le_de..... 00000000
KDB(4)> d 01A32760 80 print data
01A32760: 01A3 175C 01A3 1758 01A3 1754 01A3 1750 ... \...X...T...P
01A32770: 01A3 174C 01A3 1748 01A3 1744 01A3 1740 ... L...H...D...@
01A32780: 01A3 17A0 01A3 17E0 01A3 1820 01A3 1860 ..... /
01A32790: 01A3 18A0 01A3 18E0 01A3 1920 01A3 1960 ..... /
01A327A0: 01A3 19A0 01A3 19E0 01A3 1A20 01A3 1A60 ..... /
01A327B0: 01A3 1AA0 01A3 1AE0 01A3 1B20 01A3 1B60 ..... /
01A327C0: 0000 0000 0000 0002 0000 0002 0564 6000 ..... d'.
01A327D0: 0564 7000 0000 0000 0000 0000 0000 0000 .dp.....
KDB(4)> asc print adapter scsi table
Unable to find <adp_ctrl>
Enter the adp_ctrl address (in hex): 01A327C0
Adapter control [01A327C0]
semaphore.....00000000
num_of_opens.....00000002
num_of_cfgs.....00000002
ap_ptr[ 0] .....05646000
ap_ptr[ 1] .....05647000
ap_ptr[ 2] .....00000000
ap_ptr[ 3] .....00000000
ap_ptr[ 4] .....00000000
ap_ptr[ 5] .....00000000
ap_ptr[ 6] .....00000000
ap_ptr[ 7] .....00000000
ap_ptr[ 8] .....00000000
ap_ptr[ 9] .....00000000
ap_ptr[10] .....00000000
ap_ptr[11] .....00000000
ap_ptr[12] .....00000000
ap_ptr[13] .....00000000
ap_ptr[14] .....00000000
ap_ptr[15] .....00000000

```



```

KDB(4)> asc 0 print adapter slot 0
Adapter info [05646000]
ddi.resource_name.....        asc0
intr.next.....00000000 intr.handler.....01A329EC
intr.bus_type.....00000001 intr.flags.....00000050
intr.level.....0000000E intr.priority.....00000003
intr.bid.....820C0020 intr.i_count.....00129C8D
nnd.....0564701C
seq_number.....00000000
next.....00000000
local.eq_sf.....0565871C local.eq_ef.....05658FF7
local.eq_se.....056586E8 local.eq_top.....05658FF7
local.eq_end.....05658FFF local.dq_ee.....056591B0
local.dq_se.....056591B0 local.dq_top.....05659FF7
local.eq_wrap.....00000000 local.dq_wrap.....00000000
local.eq_status.....00000000 local.dq_status.....00000200
ddi.bus_id.....820C0020 ddi.bus_type.....00000001
ddi.slot.....00000004 ddi.base_addr.....00003540
ddi.battery_backed....00000000 ddi.dma_lvl.....00000003
ddi.int_lvl.....0000000E ddi.int_prior.....00000003
ddi.ext_bus_data_rate.0000000A ddi.tcw_start_addr...00150000
ddi.tcw_length.....00202000 ddi.tm_tcw_length....00010000
ddi.tm_tcw_start_addr.00352000 ddi.i_card_scsi_id...00000007
ddi.e_card_scsi_id...00000007 ddi.int_wide_ena.....00000001
(4)> more (C to quit) ? continue
ddi.ext_wide_ena.....00000001
active_head.....00000000 active_tail.....00000000
wait_head.....00000000 wait_tail.....00000000
num_cmds_queued.....00000000 num_cmds_active.....00000000
adp_pool.....0565B128
surr_ctl.eq_ssf.....0565B000 surr_ctl.eq_ssf_IO...00153000
surr_ctl.eq_ses.....0565B002 surr_ctl.eq_ses_IO...00153002
surr_ctl.dq_sse.....0565B004 surr_ctl.dq_sse_IO...00153004
surr_ctl.dq_sds.....0565B006 surr_ctl.dq_sds_IO...00153006
surr_ctl.dq_ssf.....0565B080 surr_ctl.dq_ssf_IO...00153080
surr_ctl.dq_ses.....0565B082 surr_ctl.dq_ses_IO...00153082
surr_ctl.eq_sse.....0565B084 surr_ctl.eq_sse_IO...00153084
surr_ctl.eq_sds.....0565B086 surr_ctl.eq_sds_IO...00153086
surr_ctl.pusa.....0565B100 surr_ctl.pusa_IO...00153100
surr_ctl.ause.....0565B104 surr_ctl.ause_IO...00153104
sta.in_use[ 0].....00000000 sta.stap[ 0].....0565A000
sta.in_use[ 1].....00000000 sta.stap[ 1].....0565A100
sta.in_use[ 2].....00000000 sta.stap[ 2].....0565A200
sta.in_use[ 3].....00000000 sta.stap[ 3].....0565A300
sta.in_use[ 4].....00000000 sta.stap[ 4].....0565A400
sta.in_use[ 5].....00000000 sta.stap[ 5].....0565A500
sta.in_use[ 6].....00000000 sta.stap[ 6].....0565A600
(4)> more (C to quit) ? continue
sta.in_use[ 7].....00000000 sta.stap[ 7].....0565A700
sta.in_use[ 8].....00000000 sta.stap[ 8].....0565A800
sta.in_use[ 9].....00000000 sta.stap[ 9].....0565A900
sta.in_use[10].....00000000 sta.stap[10].....0565AA00
sta.in_use[11].....00000000 sta.stap[11].....0565AB00
sta.in_use[12].....00000000 sta.stap[12].....0565AC00
sta.in_use[13].....00000000 sta.stap[13].....0565AD00
sta.in_use[14].....00000000 sta.stap[14].....0565AE00
sta.in_use[15].....00000000 sta.stap[15].....0565AF00
time_s.tv_sec.....00000000 time_s.tv_nsec.....00000000
tcw_table.....0565BF9C
opened.....00000001
adapter_mode.....00000001
adp_uid.....00000004 peer_uid.....00000000
system.....05658000 system_end.....0565BFAD
busmem.....00150000 busmem_end.....00154000
tm_tcw_table.....00000000
eq_raddr.....00150000 dq_raddr.....00151000
eq_vaddr.....05658000 dq_vaddr.....05659000

```

```

sta_raddr.....00152000 sta_vaddr.....0565A000
bufs.....00154000
tm_system.....00000000
(4)> more (^C to quit) ? continue
wdog.dog.next.....05646360 wdog.dog.prev.....0009A5C4
wdog.dog.func.....01A32B28 wdog.dog.count.....00000000
wdog.dog.restart.....0000001E wdog.ap.....05646000
wdog.reason.....00000004
tm.dog.next.....05647344 tm.dog.prev.....05646344
tm.dog.func.....01A32B28 tm.dog.count.....00000000
tm.dog.restart.....00000000 tm.ap.....05646000
tm.reason.....00000004
delay_trb.to_next.....00000000 delay_trb.knext.....00000000
delay_trb.kprev.....00000000 delay_trb.id.....00000000
delay_trb.cpunum.....00000000 delay_trb.flags.....00000000
delay_trb.timerid.....00000000 delay_trb.eventlist...00000000
delay_trb.timeout.it_interval.tv_sec...00000000 tv_nsec...00000000
delay_trb.timeout.it_value.tv_sec.....00000000 tv_nsec...00000000
delay_trb.func.....00000000 delay_trb.func_data...00000000
delay_trb.ipri.....00000000 delay_trb.tof.....00000000
xmem.aspace_id.....FFFFFFFF xmem.xm_flag.....FFFFFFFF
xmem.xm_version.....FFFFFFFF dma_channel.....10001000
mtu.....00141000 num_tcw_words.....00000011
shift.....0000001C tcw_word.....00000002
resvd1.....00000000 cfg_close.....00000000
vpd_close.....00000000 locate_state.....00000004
(4)> more (^C to quit) ? continue
locate_event.....FFFFFFFF rir_event.....FFFFFFFF
vpd_event.....FFFFFFFF eid_event.....FFFFFFFF
ebp_event.....FFFFFFFF eid_lock.....FFFFFFFF
recv_fn.....01A3C54C tm_recv_fn.....00000000
tm_buf_info.....00000000 tm_head.....00000000
tm_tail.....00000000 tm_recv_buf.....00000000
tm_bufs_tot.....00000000 tm_bufs_at_adp.....00000000
tm_buf.....00000000 tm_raddr.....00000000
proto_tag_e.....0565D000 proto_tag_i.....00000000
adapter_check.....00000000 eid@.....0564642C
limbo_start_time.....00000000 dev_eid.@.....056464B0
tm_dev_eid@.....056468B0 pipe_full_cnt.....00000000
dump_state.....00000000 pad.....00000000
adp_cmd_pending.....00000000 reset_pending.....00000000
epow_state.....00000000 mm_reset_in_prog....00000000
sleep_pending.....00000000 bus_reset_in_prog....00000000
first_try.....00000001 devs_in_use_I.....00000000
devs_in_use_E.....00000002 num_buf_cmds.....00000000
next_id.....000000D4 next_id_tm.....00000000
resvd4.....00000000 ebp_flag.....00000000
tm_bufs_blocked.....00000000 tm_enable_threshold..00000000
limbo.....00000000

```

vscsi Subcommand

The `vscsi` subcommand prints virtual scsi information.

Example

```

KDB(4)> lke 84 print kernel extension information
ADDRESS      FILE FILESIZE  FLAGS MODULE NAME
 84 05630780 01A36C00 00005A04 00000262 /etc/drivers/vscsiddpin
le_flags..... TEXT DATAINTEXT DATA DATAEXISTS
le_fp..... 00000000
le_loadcount... 00000000
le_usecount.... 00000001
le_data/le_tid.. 01A3C3A0 <-- address (adjusted to 0xC468) needed
le_datasize.... 00000264      to initialize the vscsi command.
le_exports..... 0565E000
le_lex..... 00000000

```

```

le_defered..... 00000000
le_filename..... 056307C4
le_ndepend..... 00000001
le_maxdepend.... 00000001
le_de..... 00000000
KDB(4)> d 01A3C3A0 100 print data
01A3C3A0: 01A3 B9DC 01A3 B9D8 01A3 B9D4 01A3 B9D0 .....
01A3C3B0: 01A3 B9CC 01A3 B9C8 01A3 B9C4 01A3 B9C0 .....
01A3C3C0: 01A3 BA20 01A3 BA60 01A3 BAA0 01A3 BAE0 ... ..'/.....
01A3C3D0: 01A3 BB20 01A3 BB60 01A3 BBA0 01A3 BBE0 ... ..'/.....
01A3C3E0: 01A3 BC20 01A3 BC60 01A3 BCA0 01A3 BCE0 ... ..'/.....
01A3C3F0: 01A3 BD20 01A3 BD60 01A3 BDA0 01A3 BDE0 ... ..'/.....
01A3C400: 7673 6373 6900 0000 0000 0000 4028 2329 vscsi.....@(#)
01A3C410: 3434 0931 2E31 3620 2073 7263 2F62 6F73 44.1.16 src/bos
01A3C420: 2F6B 6572 6E65 7874 2F73 6373 692F 7673 /kernext/scsi/vs
01A3C430: 6373 6964 6462 2E63 2C20 7379 7378 7363 csiddb.c, sysxsc
01A3C440: 7369 2C20 626F 7334 3230 2C20 3936 3133 si, bos420, 9613
01A3C450: 5420 332F 322F 3935 2031 313A 3030 3A30 T 3/2/95 11:00:0
01A3C460: 3500 0000 0000 0000 0564 F000 0565 D000 5.....d...e..
01A3C470: 0565 F000 0566 5000 0000 0000 0000 0000 .e...fP.....
01A3C480: 0000 0000 0000 0000 0000 0000 0000 0000 .....
01A3C490: 0000 0000 0000 0000 0000 0000 0000 0000 .....
KDB(4)> vsc print virtual scsi table
Unable to find <vsc_scsi_ptrs>
Enter the vsc_scsi_ptrs address (in hex): 01A3C468
Scsi pointer [01A3C468]
slot 0.....0564F000
slot 1.....0565D000
slot 2.....0565F000
slot 3.....05665000
slot 4.....00000000
slot 5.....00000000
slot 6.....00000000
slot 7.....00000000
slot 8.....00000000
slot 9.....00000000
slot 10.....00000000
slot 11.....00000000
slot 12.....00000000
slot 13.....00000000
slot 14.....00000000
slot 15.....00000000
slot 16.....00000000
slot 17.....00000000
slot 18.....00000000
slot 19.....00000000
slot 20.....00000000
(4)> more (^C to quit) ? continue
slot 21.....00000000
slot 22.....00000000
slot 23.....00000000
slot 24.....00000000
slot 25.....00000000
slot 26.....00000000
slot 27.....00000000
slot 28.....00000000
slot 29.....00000000
slot 30.....00000000
slot 31.....00000000
KDB(4)> vsc 1 print virtual scsi slot 1
Scsi info [0565D000]
ddi.resource_name..... vscsi1
ddi.parent_lname..... ascsi0
ddi.cmd_delay.....00000007 ddi.num_tm_bufs.....00000010
ddi.parent_unit_no...00000000 ddi.intr_priority....00000003
ddi.sc_im_entity_id...00000008 ddi.sc_tm_entity_id...00000009
ddi.bus_scsi_id.....00000007 ddi.wide_enabled.....00000001

```

```

ddi.location.....00000001 ddi.num_cmd_elems....00000028
cdar_wdog.dog.next...0C3AB264 cdar_wdog.dog.prev....0009AE64
cdar_wdog.dog.func...01A3C534 cdar_wdog.dog.count...00000000
cdar_wdog.dog.restart.00000007 cdar_wdog.scsi.....0565D000
cdar_wdog.index.....00000000 cdar_wdog.timer_id...00000001
cdar_wdog.save_time..00000000
reset_wdog.dog.next...0C50F000 reset_wdog.dog.prev...0009AB84
reset_wdog.dog.func...01A3C534 reset_wdog.dog.count..00000000
reset_wdog.dog.restart00000008 reset_wdog.scsi.....0565D000
reset_wdog.index.....00000000 reset_wdog.timer_id...00000004
reset_wdog.save_time..00000000
RESET_CMD_ELEM.REPLY.
header.format.....00000000 header.length.....00000000
header.options.....00000000 header.reserved.....00000000
header.src_unit.....00000000 header.src_entity....00000000
header.dest_unit.....00000000 header.dest_entity...00000000
(4)> more (^C to quit) ? continue
header.correlation_id.00000000 adap_status.....00000000
resid_count.....00000000 resid_addr.....00000000
cmd_status.....00000000 scsi_status.....00000000
cmd_error_code.....00000000 device_error_code....00000000
RESET_CMD_ELEM.CTL_ELEM
next.....00000000 prev.....00000000
flags.....00000003 key.....00000000
status.....00000000 num_pd_info.....00000000
pds_data_len.....00000000 reply_elem.....0565D07C
reply_elem_len.....0000002C ctl_elem.....0565D0D4
pd_info.....00000000
RESET_CMD_ELEM.REQUEST.
header.format.....00000000 header.length.....00000054
header.options.....00000046 header.reserved.....00000000
header.src_unit.....00000000 header.src_entity....00000000
header.dest_unit.....00000000 header.dest_entity...00000000
header.correlation_id.0565D0A8 type2_pd.desc_number..00000000
type2_pd.ctl_info....00008280 type2_pd.word1.....00000001
type2_pd.word2.....00000000 type2_pd.word3.....00000000
type1_pd.desc_number..00000000 type1_pd.ctl_info....00000180
type1_pd.word1.....00000054 type1_pd.word2.....00000000
type1_pd.word3.....00000000 scsi_cdb.next_addr1..00000000
(4)> more (^C to quit) ? continue
scsi_cdb.next_addr2..00000000 scsi_cdb.scsi_id....00000000
scsi_cdb.scsi_lun....00000000 scsi_cdb.media_flags..0000C400
RESET_CMD_ELEM.REQUEST.SCSI_CDB.
scsi_cmd_blk.scsi_op_code..00000000 scsi_cmd_blk.lun.....00000000
scsi_cmd_blk.scsi_bytes@...0565D116 scsi_extra.....00000000
scsi_data_length.....00000000
RESET_CMD_ELEM.PD_INFO1.
next.....00000000 buf_type.....00000000
pd_ctl_info.....00000000 mapped_addr.....00000000
total_len.....00000000 num_tcws.....00000000
p_buf_list.....00000000
RESET_CMD_ELEM.
bp.....00000000 scsi.....0565D000
cmd_type.....00000004 cmd_state.....00000000
preempt.....00000000 tag.....00000000
status_filter.type...00000129 status_filter.mask...0565D001
status_filter.sid....00000000
scsi_lock.....FFFFFFFF ioctl_lock.....E801AD40
devno.....00110001 open_event.....00000000
ioctl_event.....FFFFFFFF free_cmd_list@.....0565D170
shared.....05628100 dev@.....0565D194
(4)> more (^C to quit) ? continue
tm0.....0565D994 head_free.....00000000
b_pool.....00000000 read_bufs.....00000000
cmd_pool.....0C6CC000 next.....00000000
head_gw_free.....00000000 tail_gw_free.....00000000
proc_results.....00000000 proc_sleep_id.....00000000

```

```

dump_state.....00000000 opened.....00000001
num_tm_devices.....00000000 any_waiting.....00000000
pending_err.....00000000
DEV_INFO 0 [0C7A5600]
head_act.....00000000 tail_act.....00000000
head_pend.....00000000 tail_pend.....00000000
cmd_save_ptr.....00000000 async_func.....00000000
async_correlator.....00000000 dev_event.....FFFFFFFF
num_act_cmds.....00000000 trace_enabled.....00000000
qstate.....00000000 stop_pending.....00000000
dev_queuing.....00000001 need_resume_set.....00000000
cc_error_state.....00000000 waiting.....00000000
need_to_resume_queue..00000000
DEV_INFO 96 [0C50F000]
head_act.....0A048960 tail_act.....0A0488B0
head_pend.....00000000 tail_pend.....00000000
cmd_save_ptr.....00000000 async_func.....00000000
(4)> more (^C to quit) ? continue
async_correlator.....00000000 dev_event.....FFFFFFFF
num_act_cmds.....00000000 trace_enabled.....00000000
qstate.....00000000 stop_pending.....00000000
dev_queuing.....00000001 need_resume_set.....00000000
cc_error_state.....00000000 waiting.....00000000
need_to_resume_queue..00000000
KDB(4)> buf 0A048960 print head buffer (head_act)
          DEV      VNODE      BLKNO  FLAGS
0 0A048960 00100001 00000000 000DA850 MPSAFE MPSAFE_INITIAL
forw    00000000 back      00000000 av_forw 0A048800 av_back 00000000
blkno   000DA850 addr      00000000 bcount 00001000 resid 00000000
error   00000000 work      0A057424 options 00000000 event  FFFFFFFF
iodone: 018F371C
start.tv_sec      00000000 start.tv_nsec      00000000
xmcmd.aspace_id  00000000 xmcmd.xm_flag      00000000 xmcmd.xm_version  00000000
xmcmd.subspace_id 00803D0F xmcmd.subspace_id2 00000000 xmcmd.uaddr      00000000
KDB(4)> buf 0A048800 print next buffer (av_forw)
          DEV      VNODE      BLKNO  FLAGS
0 0A048800 00100001 00000000 000DAC38 MPSAFE MPSAFE_INITIAL
forw    00000000 back      00000000 av_forw 0A0488B0 av_back 0A048960
blkno   000DAC38 addr      0003A000 bcount 00001000 resid 00000000
error   00000000 work      0A0574F8 options 00000000 event  FFFFFFFF
iodone: 018F371C
start.tv_sec      00000000 start.tv_nsec      00000000
xmcmd.aspace_id  00000000 xmcmd.xm_flag      00000000 xmcmd.xm_version  00000000
xmcmd.subspace_id 00803D0F xmcmd.subspace_id2 00000000 xmcmd.uaddr      00000000
KDB(4)> buf 0A0488B0 print next buffer (av_forw)
          DEV      VNODE      BLKNO  FLAGS
0 0A0488B0 00100001 00000000 00069AE0 READ SPLIT MPSAFE MPSAFE_INITIAL
forw    00000000 back      00000000 av_forw 00000000 av_back 0A048800
blkno   00069AE0 addr      003E5000 bcount 00001000 resid 00000000
error   00000000 work      0A0575CC options 00000000 event  FFFFFFFF
iodone: 018F371C
start.tv_sec      00000000 start.tv_nsec      00000000
xmcmd.aspace_id  00000000 xmcmd.xm_flag      00000000 xmcmd.xm_version  00000000
xmcmd.subspace_id 00800802 xmcmd.subspace_id2 00000000 xmcmd.uaddr      00000000
KDB(4)> buf 0A0480B0 print next buffer (av_forw)
          DEV      VNODE      BLKNO  FLAGS
0 0A0480B0 00100001 00000000 0010BBB8 READ SPLIT MPSAFE MPSAFE_INITIAL
forw    00000000 back      00000000 av_forw 0A048160 av_back 00000000
blkno   0010BBB8 addr      0029C000 bcount 00001000 resid 00000000
error   00000000 work      0A0570D4 options 00000000 event  FFFFFFFF
iodone: 018F371C
start.tv_sec      00000000 start.tv_nsec      00000000
xmcmd.aspace_id  00000000 xmcmd.xm_flag      00000000 xmcmd.xm_version  00000000
xmcmd.subspace_id 008052D0 xmcmd.subspace_id2 00000000 xmcmd.uaddr      00000000
KDB(4)> buf 0A048160 print next buffer (av_forw)
          DEV      VNODE      BLKNO  FLAGS
0 0A048160 00100001 00000000 000ECE70 READ SPLIT MPSAFE MPSAFE_INITIAL

```

```

forw    00000000 back    00000000 av_forw  0A048000 av_back  0A0480B0
blkno   000ECE70 addr    00388000 bcount  00001000 resid  00000000
error   00000000 work    0A05727C options  00000000 event  FFFFFFFF
iodone: 018F371C
start.tv_sec    00000000 start.tv_nsec    00000000
xmcmd.aspace_id 00000000 xmcmd.xm_flag    00000000 xmcmd.xm_version 00000000
xmcmd.subspace_id 00800802 xmcmd.subspace_id2 00000000 xmcmd.uaddr    00000000
KDB(4)> buf 0A048000 print next buffer (av_forw)
          DEV    VNODE    BLKNO  FLAGS
0 0A048000 00100001 00000000 000F4D68 READ SPLIT MPSAFE MPSAFE_INITIAL
forw    00000000 back    00000000 av_forw  00000000 av_back  0A048160
blkno   000F4D68 addr    002D3000 bcount  00001000 resid  00000000
error   00000000 work    0A057350 options  00000000 event  FFFFFFFF
iodone: 018F371C
start.tv_sec    00000000 start.tv_nsec    00000000
xmcmd.aspace_id 00000000 xmcmd.xm_flag    00000000 xmcmd.xm_version 00000000
xmcmd.subspace_id 00800802 xmcmd.subspace_id2 00000000 xmcmd.uaddr    00000000
KDB(4)> buf 0A04F560 print next buffer (av_forw)
          DEV    VNODE    BLKNO  FLAGS
0 0A04F560 00100001 00000000 0017E7C0 READ SPLIT MPSAFE MPSAFE_INITIAL
forw    00000000 back    00000000 av_forw  0A04F400 av_back  00000000
blkno   0017E7C0 addr    0029C000 bcount  00001000 resid  00000000
error   00000000 work    0A057000 options  00000000 event  FFFFFFFF
iodone: 018F371C
start.tv_sec    00000000 start.tv_nsec    00000000
xmcmd.aspace_id 00000000 xmcmd.xm_flag    00000000 xmcmd.xm_version 00000000
xmcmd.subspace_id 00807F5F xmcmd.subspace_id2 00000000 xmcmd.uaddr    00000000
KDB(4)> buf 0A04F560 print next buffer (av_forw)
          DEV    VNODE    BLKNO  FLAGS
0 0A04F560 00100001 00000000 0017E7C0 READ SPLIT MPSAFE MPSAFE_INITIAL
forw    00000000 back    00000000 av_forw  0A04F400 av_back  00000000
blkno   0017E7C0 addr    0029C000 bcount  00001000 resid  00000000
error   00000000 work    0A057000 options  00000000 event  FFFFFFFF
iodone: 018F371C
start.tv_sec    00000000 start.tv_nsec    00000000
xmcmd.aspace_id 00000000 xmcmd.xm_flag    00000000 xmcmd.xm_version 00000000
xmcmd.subspace_id 00807F5F xmcmd.subspace_id2 00000000 xmcmd.uaddr    00000000
KDB(4)> buf 0A04F400 print next buffer (av_forw)
          DEV    VNODE    BLKNO  FLAGS
0 0A04F400 00100001 00000000 00172CC0 READ SPLIT MPSAFE MPSAFE_INITIAL
forw    00000000 back    00000000 av_forw  00000000 av_back  0A04F560
blkno   00172CC0 addr    0029C000 bcount  00001000 resid  00000000
error   00000000 work    0A0571A8 options  00000000 event  FFFFFFFF
iodone: 018F371C
start.tv_sec    00000000 start.tv_nsec    00000000
xmcmd.aspace_id 00000000 xmcmd.xm_flag    00000000 xmcmd.xm_version 00000000
xmcmd.subspace_id 00802CAC xmcmd.subspace_id2 00000000 xmcmd.uaddr    00000000

```

scdisk Subcommand

The `scdisk` subcommand prints disk information.

Example

```

KDB(4)> lke 80 print kernel extension information
          ADDRESS    FILE  FILESIZE    FLAGS  MODULE NAME
80 05630900 01A57E60 0000979C 00000262 /etc/drivers/scdiskpin
le_flags..... TEXT DATAINTEXT DATA DATAEXISTS
le_fp..... 00000000
le_loadcount.... 00000000
le_usecount.... 00000001
le_data/le_tid.. 01A61320 <-- address (adjusted to 0x1418) needed
le_datasize..... 000002DC      to initialize the scdisk command.
le_exports..... 0565E400
le_lex..... 00000000
le_defered..... 00000000
le_filename..... 05630944

```

```

le_ndepend..... 00000001
le_maxdepend.... 00000001
le_de..... 00000000
KDB(4)> d 01A61320 100 print data
01A61320: 0000 000B 0000 0006 FFFF FFFF 0562 7C00 .....b|.
01A61330: 0000 0000 0000 0000 0000 0000 0000 0000 .....
01A61340: 01A6 08DC 01A6 08D8 01A6 08D4 01A6 08D0 .....
01A61350: 01A6 08CC 01A6 08C8 01A6 08C4 01A6 08C0 .....
01A61360: 01A6 0920 01A6 0960 01A6 09A0 01A6 09E0 ... ..'.....
01A61370: 01A6 0A20 01A6 0A60 01A6 0AA0 01A6 0AE0 ... ..'.....
01A61380: 01A6 0B20 01A6 0B60 01A6 0BA0 01A6 0BE0 ... ..'.....
01A61390: 01A6 0C20 01A6 0C60 01A6 0CA0 01A6 0CE0 ... ..'.....
01A613A0: 7363 696E 666F 0000 6366 676C 6973 7400 scinfo..cfcglist.
01A613B0: 6F70 6C69 7374 0000 4028 2329 3435 2020 oplist..@(#)45
01A613C0: 312E 3139 2E36 2E31 3620 2073 7263 2F62 1.19.6.16 src/b
01A613D0: 6F73 2F6B 6572 6E65 7874 2F64 6973 6B2F os/kernext/disk/
01A613E0: 7363 6469 736B 622E 632C 2073 7973 7864 scdiskb.c, sysxd
01A613F0: 6973 6B2C 2062 6F73 3432 302C 2039 3631 isk, bos420, 961
01A61400: 3354 2031 2F38 2F39 3620 3233 3A34 313A 3T 1/8/96 23:41:
01A61410: 3538 0000 0000 0000 0567 4000 0567 5000 58.....g@.gP.
KDB(4)> scd print scsi disk table
Unable to find <scdisk_list>
Enter the scdisk_list address (in hex): 01A61418
Scsi pointer [01A61418]
slot 0.....05674000
slot 1.....05675000
slot 2.....0566C000
slot 3.....0566D000
slot 4.....0566E000
slot 5.....0566F000
slot 6.....05670000
slot 7.....05671000
slot 8.....05672000
slot 9.....05673000
slot 10.....0C40D000
slot 11.....00000000
slot 12.....00000000
slot 13.....00000000
slot 14.....00000000
slot 15.....00000000
KDB(4)> scd 0 print scsi disk slot 0
Scdisk info [05674000]
next.....00000000 next_open.....00000000
devno.....00120000 adapter_devno.....00100000
watchdog_timer.watch@....05674010 watchdog_timer.pointer...05674000
scsi_id.....00000000 lun_id.....00000000
reset_count.....00000000 dk_cmd_q_head.....00000000
dk_cmd_q_tail.....00000000 ioctl_cmd@.....05674034
cmd_pool.....05628400 pool_index.....00000000
open_event.....FFFFFFFF checked_cmd.....00000000
writev_err_cmd.....00000000 reassign_err_cmd.....00000000
reset_cmd@.....056740FC reqsns_cmd@.....056741AC
writev_cmd@.....0567425C q_recov_cmd@.....0567430C
reassign_cmd@.....056743BC dmp_cmd@.....0567446C
dk_bp_queue@.....0567451C mode.....00000001
disk_intrpt.....00000000 raw_io_intrpt.....00000000
ioctl_chg_mode_flg.....00000000 m_sense_status.....00000000
opened.....00000001 cmd_pending.....00000000
errno.....00000000 retain_reservation.....00000000
q_type.....00000000 q_err_value.....00000001
clr_q_on_error.....00000001 buffer_ratio.....00000000
cmd_tag_q.....00000000 q_status.....00000000
q_cTr.....00000000 timer_status.....00000000
restart_unit.....00000000 retry_flag.....00000000
(4)> more (^C to quit) ? continue
safe_relocate.....00000000 async_flag.....00000000
dump_inited.....00000001 extended_rw.....00000001

```

```

reset_delay.....00000002 starting_close.....00000000
reset_failures.....00000000 wprotected.....00000000
reserve_lock.....00000001 prevent_eject.....00000000
cfg_prevent_ej.....00000000 cfg_reserve_lck.....00000001
load_eject_alt.....00000000 pm_susp_bdr.....00000000
dev_type.....00000001 ioctl_pending.....00000000
play_audio.....00000000 override_pg_e.....00000000
cd_mode1_code.....00000000 cd_mode2_form1_code.....00000000
cd_mode2_form2_code.....00000000 cd_da_code.....00000000
current_cd_code.....00000000 current_cd_mode.....00000001
multi_session.....00000000 valid_cd_modes.....00000000
mult_of_blksize.....00000001 play_audio_started.....00000000
rw_timeout.....0000001E fmt_timeout.....00000000
start_timeout.....0000003C reassign_timeout.....00000078
queue_depth.....00000001 cmds_out.....00000000
raw_io_cmd.....00000000 currbuf.....0A0546E0
low.....0A14E3C0 block_size.....00000200
cfg_block_size.....00000200 last_ses_pvd_lba.....00000000
max_request.....00040000 max_coalesce.....00010000
lock.....FFFFFFFF fp.....00414348
(4)> more (^C to quit) ? continue
error_rec@.....05674598 stats@.....05674648
mode_data_length.....0000003D disc_info@.....0567465C
mode_buf@.....05674660 sense_buf@.....05674760
ch_data@.....05674860 df_data@.....05674960
def_list_header@.....05674A60 ioctl_buf@.....05674A64
mode_page_e@.....05674B63 dd@.....05674B6C
df@.....05674BB4 ch@.....05674BFC
cd@.....05674C44 ioctl_req_sense@.....05674C8C
capacity@.....05674CA4 def_list@.....05674CAC
dkstat@.....05674CB4
spin_lock@.....05674CF8 spin_lock.....E80039A0
pmh@.....05674CFC pm_pending.....00000000
pm_reserve@.....05674D41 pm_device_id.....00100000
pm_event.....FFFFFFFF pm_timer@.....05674D4C
KDB(4)> file 00414348 print file (fp)
          COUNT          OFFSET      DATA TYPE  FLAGS
18 file+000330      1 0000000000000000 0BC4A950 GNODE WRITE
f_flag..... 00000002 f_count..... 00000001
f_msgcount..... 0000 f_type..... 0003
f_data..... 0BC4A950 f_offset... 0000000000000000
f_dir_off..... 00000000 f_cred..... 00000000
f_lock@..... 00414368 f_lock..... E88007C0
f_offset_lock@. 0041436C f_offset_lock.. E88007E0
f_vinfo..... 00000000 f_ops..... 001F3CD0 gno_fops+000000
GNODE..... 0BC4A950
gn_seg..... 007FFFFFF gn_mwrcnt... 00000000 gn_mrdcnt... 00000000
gn_rdcnt..... 00000000 gn_wrcnt... 00000002 gn_excnt... 00000000
gn_rshcnt... 00000000 gn_ops..... 00000000 gn_vnode... 00000000
gn_reclck... 00000000 gn_rdev..... 00100000
gn_chan..... 00000000 gn_filocks.. 00000000 gn_data..... 0BC4A940
gn_type..... BLK      gn_flags.....
KDB(4)> buf 0A0546E0 print current buffer (currbuf)
          DEV      VNODE      BLKNO FLAGS
0 0A0546E0 00120000 00000000 00070A58 READ SPLIT MPSAFE MPSAFE_INITIAL
forw      00000000 back      00000000 av_forw 0A05DC60 av_back 0A14E3C0
blkno     00070A58 addr      00626000 bcount 00001000 resid 00000000
error     00000000 work      00000000 options 00000000 event  FFFFFFFF
iodone:   019057D4
start.tv_sec      00000000 start.tv_nsec      00000000
xmmd.aspace_id    00000000 xmmd.xm_flag        00000000 xmmd.xm_version    00000000
xmmd.subspace_id 00800802 xmmd.subspace_id2  00000000 xmmd.uaddr         00000000

```


Memory Allocator Subcommands for the KDB Kernel Debugger and kdb Command

heap Subcommand

Example

```
KDB(2)> hp print kernel heap information
Pinned heap 0FFC4000
sanity..... 48454150 base..... F11B7000
lock@..... 0FFC4008 lock..... 00000000
alt..... 00000001 numpages... 0000EE49
amount..... 002D2750 pinflag... 00000001
newheap.... 00000000 protect.... 00000000
limit..... 00000000 heap64.... 00000000
vmrelflag.. 00000000 rhash..... 00000000
pagtot..... 00000000 pagused.... 00000000
frtot[00].. 00000000 [01].. 00000000 [02].. 00000000 [03].. 00000000
frtot[04].. 00000000 [05].. 00000000 [06].. 00000000 [07].. 00000000
frtot[08].. 00000000 [09].. 00000000 [10].. 00000000 [11].. 00000000
frused[00]. 00000000 [01].. 00000000 [02].. 00000000 [03].. 00000000
frused[04]. 00000000 [05].. 00000000 [06].. 00000000 [07].. 00000000
frused[08]. 00000000 [09].. 00000000 [10].. 00000000 [11].. 00000000
fr[00]..... 00FFFFFF [01].. 00FFFFFF [02].. 00FFFFFF [03].. 00FFFFFF
fr[04]..... 00003C22 [05].. 00004167 [06].. 00004A05 [07].. 00004845
fr[08]..... 000043B5 [09].. 00000002 [10].. 0000443A [11].. 00004842
Kernel heap 0FFC40B8
sanity..... 48454150 base..... F11B6F48
lock@..... 0FFC40C0 lock..... 00000000
alt..... 00000000 numpages... 0000EE49
amount..... 04732CF0 pinflag... 00000000
newheap.... 00000000 protect.... 00000000
limit..... 00000000 heap64.... 00000000
vmrelflag.. 00000000 rhash..... 00000000
pagtot..... 00000000 pagused.... 00000000
frtot[00].. 00000000 [01].. 00000000 [02].. 00000000 [03].. 00000000
frtot[04].. 00000000 [05].. 00000000 [06].. 00000000 [07].. 00000000
frtot[08].. 00000000 [09].. 00000000 [10].. 00000000 [11].. 00000000
frused[00]. 00000000 [01].. 00000000 [02].. 00000000 [03].. 00000000
frused[04]. 00000000 [05].. 00000000 [06].. 00000000 [07].. 00000000
frused[08]. 00000000 [09].. 00000000 [10].. 00000000 [11].. 00000000
fr[00]..... 00FFFFFF [01].. 00FFFFFF [02].. 00FFFFFF [03].. 00FFFFFF
fr[04]..... 000049E9 [05].. 00003C26 [06].. 0000484E [07].. 00004737
fr[08]..... 00003C0A [09].. 00004A07 [10].. 00004855 [11].. 00004A11
addr..... 0000000000000000 maxpages..... 00000000
peakpage..... 00000000 limit_callout..... 00000000
newseg_callout... 00000000 pagesoffset..... 0FFC4194
pages_sid..... 00000000
Heap anchor
... 0FFC4190 pageno FFFFFFFF pages.type.. 00 allocpage offset... 00004A08
Heap Free list
... 0FFD69B4 pageno 00004A08 pages.type.. 02 freepage offset... 00004A0C
... 0FFD69C4 pageno 00004A0C pages.type.. 03 freerange offset... 00004A17
... 0FFD69C8 pageno 00004A0D pages.type.. 04 freesize size..... 00000005
... 0FFD69D4 pageno 00004A10 pages.type.. 05 freerangeend offset... 00004A0C
... 0FFD69F0 pageno 00004A17 pages.type.. 03 freerange offset... NO_PAGE
... 0FFD69F4 pageno 00004A18 pages.type.. 04 freesize size..... 0000A432
... 0FFFFAB4 pageno 0000EE48 pages.type.. 05 freerangeend offset... 00004A17
Heap Alloc list
... 0FFC41B0 pageno 00000007 pages.type.. 01 allocrange offset... NO_PAGE
... 0FFC41B4 pageno 00000008 pages.type.. 06 allocsize size..... 00001E00
... 0FFCB9AC pageno 00001E06 pages.type.. 07 allocrangeend offset... 00000007
... 0FFCB9B0 pageno 00001E07 pages.type.. 01 allocrange offset... NO_PAGE
... 0FFCB9B4 pageno 00001E08 pages.type.. 06 allocsize size..... 00001E00
... 0FFD31AC pageno 00003C06 pages.type.. 07 allocrangeend offset... 00001E07
```

```

... 0FFD31B4 pageno 00003C08 pages.type.. 01 allocrange   offset... 00003C42
... 0FFD31B8 pageno 00003C09 pages.type.. 06 allocsize    size..... 00000002
... 0FFD31C4 pageno 00003C0C pages.type.. 01 allocrange   offset... NO_PAGE
... 0FFD31C8 pageno 00003C0D pages.type.. 06 allocsize    size..... 00000009
... 0FFD31E4 pageno 00003C14 pages.type.. 07 allocrangeend offset... 00003C0C
... 0FFD31E8 pageno 00003C15 pages.type.. 01 allocrange   offset... NO_PAGE
... 0FFD31EC pageno 00003C16 pages.type.. 06 allocsize    size..... 00000009
... 0FFD3208 pageno 00003C1D pages.type.. 07 allocrangeend offset... 00003C15
... 0FFD320C pageno 00003C1E pages.type.. 01 allocrange   offset... NO_PAGE
...
KDB(3)> dw msg_heap 8 look at message heap
msg_heap+000000: 0000A02A CFFBF0B8 0000B02B CFFBF0B8 ...*.....+....
msg_heap+000010: 0000C02C CFFBF0B8 0000D02D CFFBF0B8 ...;.....-....
KDB(3)> mr s12 set SR12 with message heap SID
s12 : 007FFFFFF = 0000A02A
KDB(3)> heap CFFBF0B8 print message heap
Heap CFFBF000
sanity..... 48454150 base..... F0041000
lock@..... CFFBF008 lock..... 00000000
alt..... 00000001 numpages... 0000FFBF
amount..... 00000000 pinflag... 00000000
newheap.... 00000000 protect.... 00000000
limit..... 00000000 heap64.... 00000000
vmrelflag.. 00000000 rhash..... 00000000
pagtot..... 00000000 pagused... 00000000
frtot[00].. 00000000 [01].. 00000000 [02].. 00000000 [03].. 00000000
frtot[04].. 00000000 [05].. 00000000 [06].. 00000000 [07].. 00000000
frtot[08].. 00000000 [09].. 00000000 [10].. 00000000 [11].. 00000000
frused[00]. 00000000 [01].. 00000000 [02].. 00000000 [03].. 00000000
frused[04]. 00000000 [05].. 00000000 [06].. 00000000 [07].. 00000000
frused[08]. 00000000 [09].. 00000000 [10].. 00000000 [11].. 00000000
fr[00]..... 00FFFFFF [01].. 00FFFFFF [02].. 00FFFFFF [03].. 00FFFFFF
fr[04]..... 00FFFFFF [05].. 00FFFFFF [06].. 00FFFFFF [07].. 00FFFFFF
fr[08]..... 00FFFFFF [09].. 00FFFFFF [10].. 00FFFFFF [11].. 00FFFFFF
Heap CFFBF0B8
sanity..... 48454150 base..... F0040F48
lock@..... CFFBF0C0 lock..... 00000000
alt..... 00000000 numpages... 0000FFBF
amount..... 00000100 pinflag... 00000000
newheap.... 00000000 protect.... 00000000
limit..... 00000000 heap64.... 00000000
vmrelflag.. 00000000 rhash..... 00000000
pagtot..... 00000000 pagused... 00000000
frtot[00].. 00000000 [01].. 00000000 [02].. 00000000 [03].. 00000000
frtot[04].. 00000000 [05].. 00000000 [06].. 00000000 [07].. 00000000
frtot[08].. 00000000 [09].. 00000000 [10].. 00000000 [11].. 00000000
frused[00]. 00000000 [01].. 00000000 [02].. 00000000 [03].. 00000000
frused[04]. 00000000 [05].. 00000000 [06].. 00000000 [07].. 00000000
frused[08]. 00000000 [09].. 00000000 [10].. 00000000 [11].. 00000000
fr[00]..... 00FFFFFF [01].. 00FFFFFF [02].. 00FFFFFF [03].. 00FFFFFF
fr[04]..... 00FFFFFF [05].. 00FFFFFF [06].. 00FFFFFF [07].. 00FFFFFF
fr[08]..... 00000000 [09].. 00FFFFFF [10].. 00FFFFFF [11].. 00FFFFFF
addr..... 0000000000000000 maxpages..... 00000000
peakpage..... 00000000 limit_callout.... 00000000
newseg_callout.... 00000000 pagesoffset..... 00000194
pages_sid..... 00000000
Heap anchor
... CFFBF190 pageno FFFFFFFF pages.type.. 00 allocpage   offset... 00000001
Heap Free list
... CFFBF198 pageno 00000001 pages.type.. 03 freerange   offset... NO_PAGE
... CFFBF19C pageno 00000002 pages.type.. 04 freesize    size..... 0000FFBE
... CFFFF08C pageno 0000FFBE pages.type.. 05 freerangeend offset... 00000001
Heap Alloc list
KDB(3)> mr s12 reset SR12
s12 : 0000A02A = 007FFFFFF

```

xm Subcommand

Example

```
(0)> stat
RS6K_SMP_MCA POWER_PC POWER_604 machine with 8 cpu(s)
..... SYSTEM STATUS
sysname... AIX          nodename.. jumbo32
release... 3           version... 4
machine... 00920312A0 nid..... 920312A0
time of crash: Fri Jul 11 08:07:01 1997
age of system: 1 day, 20 hr., 31 min., 17 sec.
..... PANIC STRING
Memdbg: *w == pat
(0)> xm -? Display usage
xmalloc <addr>
    Print all available xmalloc information about <addr>.
    If debug xmalloc kernel is available, also print out
    information from xmalloc -s and xmalloc -h.
xmalloc -s <addr>
    Print debug xmalloc allocation records matching associated with <addr>
    (Debug xmalloc kernel only.)
xmalloc -h <addr>
    Print records in debug xmalloc kernel free list associated with <addr>
    (Debug xmalloc kernel only.)
xmalloc [-1] -f
    Print allocation records on free list, from first-freed to last-freed.
    If "-1" is specified, verbose records are printed
    (Debug xmalloc kernel only.)
xmalloc [-1] -a
    Print allocation record table.
    If "-1" is specified, verbose records are printed.
    (Debug xmalloc kernel only.)
xmalloc [-1] -p <pageno>
    Print page descriptor information for page <pageno>.
    If "-1" is specified, print extra info, even on failure.
xmalloc -d <addr>
    Print debug xmalloc kernel allocation record hash chain that
    is associated with the record hash value for <addr>.
xmalloc -v
    Verifies allocation trailers of allocated records,
    and free fill patterns of freed records.
    (Debug xmalloc kernel only.)
xmalloc -u
    Print xmalloc usage histogram. It tells the size the kernel heap
    and how much of it has been used. Next, it prints a list of
    allocated memory blocks sorted by allocation size, one per line.
    Memory allocations of the same size from the same routine are coalesced,
    keeping track of how many there were.
(0)> xm -s Display debug xmalloc status
Debug kernel error message: The xmfree service has found data written beyond the
end of the memory buffer that is being freed.
Address at fault was 0x09410200
(0)> xm -h 0x09410200 Display debug xmalloc records associated with addr
0B78DAB0: addr..... 09410200 req_size..... 128 freed unpinned
0B78DAB0: pid..... 00043158 comm..... bcross
Trace during xmalloc()                Trace during xmfree()
002329E4(.xmalloc+0000A8)              002328F0(.xmfree+0000FC)
00235CD4(.dlistadd+000040)            00234F04(.setbitmaps+0001BC)
00235520(.newblk+00006C)              00236894(.finicom+0001A4)
0B645120: addr..... 09410200 req_size..... 128 freed unpinned
0B645120: pid..... 0007DCAC comm..... bcross
Trace during xmalloc()                Trace during xmfree()
002329E4(.xmalloc+0000A8)              002328F0(.xmfree+0000FC)
00235CD4(.dlistadd+000040)            00236614(.logdfree+0001E8)
00236574(.logdfree+000148)            00236720(.finicom+000030)
0B7A3750: addr..... 09410200 req_size..... 128 freed unpinned
```

```

0B7A3750: pid..... 000010BA comm..... syncd
Trace during xmalloc()                               Trace during xfree()
002329E4(.xmalloc+0000A8)                             002328F0(.xfree+0000FC)
00235CD4(.dlistadd+000040)                             00234F04(.setbitmaps+0001BC)
00235520(.newblk+00006C)                               00236894(.finicom+0001A4)
0B52B330: addr..... 09410200 req_size.... 128 freed unpinned
0B52B330: pid..... 00058702 comm..... bcross
Trace during xmalloc()                               Trace during xfree()
002329E4(.xmalloc+0000A8)                             002328F0(.xfree+0000FC)
00235CD4(.dlistadd+000040)                             00236698(.logdfree+00026C)
00236510(.logdfree+0000E4)                             00236720(.finicom+000030)
07A33840: addr..... 09410200 req_size.... 133 freed unpinned
07A33840: pid..... 00042C24 comm..... ksh
Trace during xmalloc()                               Trace during xfree()
002329E4(.xmalloc+0000A8)                             002328F0(.xfree+0000FC)
00271F28(.ld_pathopen+000160)                         00271D24(.ld_pathclear+00008C)
0027FB6C(.ld_getlib+000074)                           002ABF04(.ld_execload+00075C)
0B796480: addr..... 09410200 req_size.... 133 freed unpinned
0B796480: pid..... 0005C2E0 comm..... ksh
Trace during xmalloc()                               Trace during xfree()
002329E4(.xmalloc+0000A8)                             002328F0(.xfree+0000FC)
00271F28(.ld_pathopen+000160)                         00271D24(.ld_pathclear+00008C)
0027FB6C(.ld_getlib+000074)                           002ABF04(.ld_execload+00075C)
07A31420: addr..... 09410200 req_size.... 135 freed unpinned
07A31420: pid..... 0007161A comm..... ksh
Trace during xmalloc()                               Trace during xfree()
002329E4(.xmalloc+0000A8)                             002328F0(.xfree+0000FC)
00271F28(.ld_pathopen+000160)                         00271D24(.ld_pathclear+00008C)
0027FB6C(.ld_getlib+000074)                           002ABF04(.ld_execload+00075C)
07A38630: addr..... 09410200 req_size.... 125 freed unpinned
07A38630: pid..... 0001121E comm..... ksh
Trace during xmalloc()                               Trace during xfree()
002329E4(.xmalloc+0000A8)                             002328F0(.xfree+0000FC)
00271F28(.ld_pathopen+000160)                         00271D24(.ld_pathclear+00008C)
0027FB6C(.ld_getlib+000074)                           002ABF04(.ld_execload+00075C)
07A3D240: addr..... 09410200 req_size.... 133 freed unpinned
07A3D240: pid..... 0000654C comm..... ksh
Trace during xmalloc()                               Trace during xfree()
002329E4(.xmalloc+0000A8)                             002328F0(.xfree+0000FC)
00271F28(.ld_pathopen+000160)                         00271D24(.ld_pathclear+00008C)
0027FB6C(.ld_getlib+000074)                           002ABF04(.ld_execload+00075C)

```

The **xm** subcommand can be used to find memory location of any heap record, knowing the page index (pageno) , or to find the heap record knowing the allocated memory location.

Example

```

(0)> heap
...
Heap Alloc list
... 0FFC41B0 pageno 00000007 pages.type.. 01 allocrange   offset... NO_PAGE
... 0FFC41B4 pageno 00000008 pages.type.. 06 allocsize    size..... 00001E00
... 0FFCB9AC pageno 00001E06 pages.type.. 07 allocrangeend offset... 00000007
... 0FFCB9B0 pageno 00001E07 pages.type.. 01 allocrange   offset... NO_PAGE
... 0FFCB9B4 pageno 00001E08 pages.type.. 06 allocsize    size..... 00001E00
... 0FFD31AC pageno 00003C06 pages.type.. 07 allocrangeend offset... 00001E07
... 0FFD31B4 pageno 00003C08 pages.type.. 01 allocrange   offset... 00003C42
... 0FFD31B8 pageno 00003C09 pages.type.. 06 allocsize    size..... 00000002
... 0FFD31C4 pageno 00003C0C pages.type.. 01 allocrange   offset... NO_PAGE
... 0FFD31C8 pageno 00003C0D pages.type.. 06 allocsize    size..... 00000009
... 0FFD31E4 pageno 00003C14 pages.type.. 07 allocrangeend offset... 00003C0C
...
(0)> xm -l -p 00001E07 how to find memory address of heap index 00001E07
type..... 1 (P_allocrange)
page_addr..... 02F82000 pinned..... 0
size..... 00000000 offset..... 00FFFFFF

```

```

page_descriptor_address.. 0FFCB9B0
(0)> xm -l 02F82000 how to find page index in kernel heap of 02F82000
P_allocrange (range of 2 or more allocated full pages)
page..... 00001E07 start..... 02F82000 page_cnt..... 00001E00
allocated_size. 01E00000 pinned..... unknown
(0)> xm -l -p 00003C08 how to find memory address of heap index 00003C08
type..... 1 (P_allocrange)
page_addr..... 04D83000 pinned..... 0
size..... 00000000 offset..... 00003C42
page_descriptor_address.. 0FFD31B4
(0)> xm -l 04D83000 ow to find page index in kernel heap of 04D83000
P_allocrange (range of 2 or more allocated full pages)
page..... 00003C08 start..... 04D83000 page_cnt..... 00000002
allocated_size. 00002000 pinned..... unknown

```

bucket Subcommand

The **bucket** subcommand prints kernel memory allocator buckets.

Example

```

KDB(0)> bucket ? print usage
Usage: bucket [-l] [-c cpu] [-i index] [symb/eaddr]
-l to display bucket free list
-c to display only a cpu buckets
-i to display only a bucket index
symb/eaddr to display only one bucket
KDB(0)> bucket -l -c 4 -i 13 print processor 4 8K bytes buckets
displaying kmembucket for cpu 4 offset 13 size 0x00002000
address.....00376404
b_next..(x).....0659F000
b_calls..(x).....0000AE8B
b_total..(x).....00000003
b_totalfree..(x).....00000003
b_elmpercl..(x).....00000001
b_highwat..(x).....0000000A
b_couldfree (sic)..(x)...00000000
b_failed..(x).....00000000
lock..(x).....00000000
Bucket free list....
  1 next...0659F000, kmemusage...09B57268 [000D 0001 00000004]
  2 next...0619E000, kmemusage...09B55260 [000D 0001 00000004]
  3 next...06687000, kmemusage...09B579A8 [000D 0001 00000004]
KDB(0)> bucket -c 3 print all processor 3 buckets
displaying kmembucket for cpu 3 offset 0 size 0x00000002
address.....00375F3C
b_next..(x).....00000000
b_calls..(x).....00000000
b_total..(x).....00000000
b_totalfree..(x).....00000000
b_elmpercl..(x).....00001000
b_highwat..(x).....00005000
b_couldfree (sic)..(x)...00000000
b_failed..(x).....00000000
lock..(x).....00000000
displaying kmembucket for cpu 3 offset 1 size 0x00000004
address.....00375F60
b_next..(x).....00000000
b_calls..(x).....00000000
b_total..(x).....00000000
b_totalfree..(x).....00000000
b_elmpercl..(x).....00000800
b_highwat..(x).....00002800
b_couldfree (sic)..(x)...00000000
(0)> more (^C to quit) ? continue
b_failed..(x).....00000000
lock..(x).....00000000

```

```

...
displaying kmembucket for cpu 3 offset 8 size 0x00000100
address.....0037605C
b_next..(x).....062A2700
b_calls..(x).....00B3F6EA
b_total..(x).....00000330
b_totalfree..(x).....00000031
b_elmpercl..(x).....00000010
b_highwat..(x).....00000180
b_couldfree (sic)..(x)...00000000
b_failed..(x).....00000000
lock..(x).....00000000
displaying kmembucket for cpu 3 offset 9 size 0x00000200
address.....00376080
b_next..(x).....05D30000
b_calls..(x).....0000A310
b_total..(x).....00000010
b_totalfree..(x).....0000000C
b_elmpercl..(x).....00000008
b_highwat..(x).....00000028
b_couldfree (sic)..(x)...00000000
b_failed..(x).....00000000
lock..(x).....00000000
...
displaying kmembucket for cpu 3 offset 20 size 0x00200000
(0)> more (^C to quit) ? continue
address.....0037620C
b_next..(x).....00000000
b_calls..(x).....00000000
b_total..(x).....00000000
b_totalfree..(x).....00000000
b_elmpercl..(x).....00000001
b_highwat..(x).....0000000A
b_couldfree (sic)..(x)...00000000
b_failed..(x).....00000000
lock..(x).....00000000
KDB(0)>

```

kmstats Subcommands

The **kmstats** subcommand prints kernel allocator memory statistics.

Example

```

KDB(0)> kmstats print allocator statistics
displaying kmemstats for offset 0 free
address.....0025C120
inuse..(x).....00000000
calls..(x).....00000000
memuse..(x).....00000000
limit blocks..(x).....00000000
map blocks..(x).....00000000
maxused..(x).....00000000
limit..(x).....02666680
failed..(x).....00000000
lock..(x).....00000000
displaying kmemstats for offset 1 mbuf
address.....0025C144
inuse..(x).....0000000D
calls..(x).....002C4E54
memuse..(x).....00000D00
limit blocks..(x).....00000000
map blocks..(x).....00000000
maxused..(x).....0001D700
limit..(x).....02666680
(0)> more (^C to quit) ? continue
failed..(x).....00000000

```

```

lock..(x).....00000000
displaying kmemstats for offset 2 mcluster
address.....0025C168
inuse..(x).....00000002
calls..(x).....00023D4E
memuse..(x).....00000900
limit blocks..(x).....00000000
map blocks..(x).....00000000
maxused..(x).....00079C00
limit..(x).....02666680
failed..(x).....00000000
lock..(x).....00000000
...
displaying kmemstats for offset 48 kalloc
address.....0025C7E0
inuse..(x).....00000000
calls..(x).....00000000
memuse..(x).....00000000
limit blocks..(x).....00000000
map blocks..(x).....00000000
maxused..(x).....00000000
limit..(x).....02666680
failed..(x).....00000000
lock..(x).....00000000
displaying kmemstats for offset 49 temp
address.....0025C804
inuse..(x).....00000007
calls..(x).....00000007
memuse..(x).....00003500
(0)> more (^C to quit) ? continue
limit blocks..(x).....00000000
map blocks..(x).....00000000
maxused..(x).....00003500
limit..(x).....02666680
failed..(x).....00000000
lock..(x).....00000000
KDB(0)>

```

File System Subcommands for the KDB Kernel Debugger and kdb Command

buffer Subcommand

The **buf** subcommand prints buffer cache headers.

Example

```

KDB(0)> buf print buffer pool
 1 057E4000 nodevice 00000000 00000000
 2 057E4058 nodevice 00000000 00000000
 3 057E40B0 nodevice 00000000 00000000
 4 057E4108 nodevice 00000000 00000000
 5 057E4160 nodevice 00000000 00000000
...
18 057E45D8 nodevice 00000000 00000000
19 057E4630 000A0011 00000000 00000100 READ DONE STALE MPSAFE MPSAFE_INITIAL
20 057E4688 000A0011 00000000 00000008 READ DONE STALE MPSAFE MPSAFE_INITIAL
KDB(0) buf 19 print buffer slot 19
                DEV      VNODE      BLKNO  FLAGS
19 057E4630 000A0011 00000000 00000100 READ DONE STALE MPSAFE MPSAFE_INITIAL
forw   0562F0CC back    0562F0CC av_forw 057E45D8 av_back 057E4688
blkno  00000100 addr    0580C000 bcount 00001000 resid 00000000
error  00000000 work     80000000 options 00000000 event  FFFFFFFF
iodone: biodone+000000
start.tv_sec      00000000 start.tv_nsec      00000000
xmemd.aspace_id  00000000 xmemd.xm_flag      00000000 xmemd.xm_version  00000000

```

```

xmemd.subspace_id 00000000 xmemd.subspace_id2 00000000 xmemd.uaddr      00000000
KDB(0)> pdt 17 print paging device slot 17 (the 1st FS)
PDT address B69C0440 entry 17 of 511, type: FILESYSTEM
next pdt on i/o list (nextio) : FFFFFFFF
dev_t or strategy ptr (device) : 000A0007
last frame w/pend I/O (iotail) : FFFFFFFF
free buf_struct list (bufstr) : 056B2108
total buf_structs (nbufs) : 005D
available (PAGING) (avail) : 0000
JFS disk agsize (agsize) : 0800
JFS inode agsize (iagsize) : 0800
JFS log SCB index (logsidx) : 00035
JFS fragments per page(fperpage): 1
JFS compression type (comptype): 0
JFS log2 bigalloc mult(bigexp) : 0
disk map srval (dmsrval) : 00002021
i/o's not finished (iocnt) : 00000000
lock (lock) : E8003200
KDB(0)> buf 056B2108 print paging device first free buffer
      DEV      VNODE      BLKNO  FLAGS
0 056B2108 000A0007 00000000 00000048 DONE SPLIT MPSAFE MPSAFE_INITIAL
forw 0007DAB3 back 00000000 av_forw 056B20B0 av_back 00000000
blkno 00000048 addr 00000000 bcount 00001000 resid 00000000
error 00000000 work 00400000 options 00000000 event 00000000
iodone: v_pfind+0000000
start.tv_sec 00000000 start.tv_nsec 00000000
xmemd.aspace_id 00000000 xmemd.xm_flag 00000000 xmemd.xm_version 00000000
xmemd.subspace_id 0083E01F xmemd.subspace_id2 00000000 xmemd.uaddr 00000000

```

hbuffer Subcommand

The **hb** subcommand prints buffer cache hash list of headers.

Example

```

KDB(0)> hb print buffer cache hash lists
      BUCKET HEAD      COUNT
0562F0CC 18 057E4630 1
0562F12C 26 057E4688 1
KDB(0)> hb 26 print buffer cache hash list bucket 26
      DEV      VNODE      BLKNO  FLAGS
20 057E4688 000A0011 00000000 00000008 READ DONE STALE MPSAFE MPSAFE_INITIAL

```

fbuffer Subcommand

The **fb** subcommand prints buffer cache freelist of headers.

Example

```

KDB(0)> fb print free list buffer buckets
      BUCKET      HEAD      COUNT
bfreelist+0000000 0001 057E4688 20
KDB(0)> fb 1 print free list buffer bucket 1
      DEV      VNODE      BLKNO  FLAGS
20 057E4688 000A0011 00000000 00000008 READ DONE STALE MPSAFE MPSAFE_INITIAL
19 057E4630 000A0011 00000000 00000100 READ DONE STALE MPSAFE MPSAFE_INITIAL
18 057E45D8 nodevice 00000000 00000000
17 057E4580 nodevice 00000000 00000000
...
2 057E4058 nodevice 00000000 00000000
1 057E4000 nodevice 00000000 00000000

```

gnode Subcommand

The **gno** subcommand prints the generic node structure.

Example

```
(0)> gno 09D0FD68 print gnode
GNODE..... 09D0FD68
gn_type..... 00000002 gn_flags..... 00000000 gn_seg..... 0001A3FA
gn_mwrcnt.... 00000000 gn_mrdcnt.... 00000000 gn_rdcnt..... 00000000
gn_wrcnt..... 00000000 gn_excnt..... 00000000 gn_rshcnt.... 00000000
gn_vnode..... 09D0FD28 gn_rdev..... 000A0010 gn_ops..... jfs_vops
gn_chan..... 00000000 gn_recl_k_lock. 00000000 gn_recl_k_lock@ 09D0FD9C
gn_recl_k_event FFFFFFFF gn_filocks.... 00000000 gn_data..... 09D0FD58
gn_type..... DIR
```

gfs Subcommand

The **gfs** subcommand prints the generic file system structure.

Example

```
(0)> gfs gfs print gfs slot 1
gfs_data. 00000000 gfs_flag. INIT VERSION4 VERSION42 VERSION421
gfs_ops.. jfs_vfsops   gn_ops... jfs_vops       gfs_name. jfs
gfs_init. jfs_init     gfs_rinit jfs_rootinit  gfs_type. JFS
gfs_hold. 00000012
(0)> gfs gfs+30 print gfs slot 2
gfs_data. 00000000 gfs_flag. INIT VERSION4 VERSION42 VERSION421
gfs_ops.. spec_vfsops gn_ops... spec_vnops   gfs_name. sfs
gfs_init. spec_init   gfs_rinit nodev       gfs_type. SFS
gfs_hold. 00000000
(0)> gfs gfs+60 print gfs slot 3
gfs_data. 00000000 gfs_flag. REMOTE VERSION4
gfs_ops.. 01D2ABF8   gn_ops... 01D2A328     gfs_name. nfs
gfs_init. 01D2B5F0   gfs_rinit 00000000    gfs_type. NFS
gfs_hold. 0000000E
```

file Subcommand

The **file** subcommand prints the file table. First, used files are printed (count > 0), then others.

Example

```
(0)> file print file table
          COUNT      OFFSET      DATA TYPE  FLAGS
1 file+000000      1 0000000000000100 09CD90C8 VNODE EXEC
2 file+000030      1 0000000000000100 09CC4DE8 VNODE EXEC
3 file+000060    1452 000000000019B084 09CC2B50 VNODE READ RSHARE
4 file+000090      2 0000000000000100 09CFC800 VNODE EXEC
5 file+0000C0      2 0000000000000000 056CE008 VNODE READ WRITE
6 file+0000F0      1 0000000000000000 056CE008 VNODE READ WRITE
7 file+000120      1 0000000000000680 09CFF680 VNODE READ WRITE
8 file+000150      1 0000000000000100 0B97BE0C VNODE EXEC
9 file+000180      2 0000000000000000 056CE070 VNODE READ NONBLOCK
10 file+0001B0    323 000000000000061C 09CC4F30 VNODE READ RSHARE
11 file+0001E0      1 0000000000000000 0B7E8700 READ WRITE
12 file+000210     16 000000000000061C 09CC5AB8 VNODE READ RSHARE
13 file+000240      1 0000000000000000 0B221950 GNODE WRITE
14 file+000270      1 0000000000000000 0B221A20 GNODE WRITE
15 file+0002A0      2 000000000000055C 09CFFCE8 VNODE READ RSHARE
16 file+0002D0      2 0000000000000000 09CFE9B0 VNODE WRITE
17 file+000300      1 0000000000000000 0B7E8600 READ WRITE
18 file+000330      1 0000000000000000 056CE008 VNODE READ
19 file+000360      1 0000000000000000 09CFBB90 VNODE WRITE
20 file+000390      3 00000000000284A 0B99A60C VNODE READ
(0)> more (^C to quit) ? Interrupted
(0)> file 3 print file slot 3
          COUNT      OFFSET      DATA TYPE  FLAGS
3 file+000060    1474 000000000019B084 09CC2B50 VNODE READ RSHARE
f_flag..... 00001001 f_count..... 000005C2
```

```

f_msgcount..... 0000 f_type..... 0001
f_data..... 09CC2B50 f_offset... 00000000019B084
f_dir_off..... 00000000 f_cred..... 056D0E58
f_lock@..... 004AF098 f_lock..... 00000000
f_offset_lock@. 004AF09C f_offset_lock.. 00000000
f_vinfo..... 00000000 f_ops..... 00250FC0 vnodeops+000000
VNODE..... 09CC2B50
v_flag.... 00000000 v_count... 00000002 v_vfsgen.. 00000000
v_lock.... 00000000 v_lock@... 09CC2B5C v_vfsp.... 056D18A4
v_mvfsp... 00000000 v_gnode... 09CC2B90 v_next.... 00000000
v_vfsnext. 09CC2A08 v_vfsprev. 09CC3968 v_pfsvnode 00000000
v_audit... 00000000

```

inode Subcommand

The **ino** subcommand prints the inode table. Only used (hashed) inode are printed (count > 0). Unused inodes (icache list) may be printed with the **fino** subcommand.

Example

```

(0)> ino print inode table
      DEV      NUMBER CNT      GNODE      IPMNT TYPE FLAGS
1 0A2A4968 00330003      10721 1 0A2A4978 09F79510 DIR
2 0A2A9790 00330003      10730 1 0A2A97A0 09F79510 REG
3 0A321E90 00330006       2948 1 0A321EA0 09F7A990 DIR
4 0A32ECD8 00330006       2965 1 0A32ECE8 09F7A990 DIR
5 0A38EBC8 00330006       3173 1 0A38EBD8 09F7A990 DIR
6 0A3CC280 00330006       3186 1 0A3CC290 09F7A990 REG
7 09D01570 000A0005      14417 1 09D01580 09CC1990 REG
8 09D7CE68 000A0005      47211 1 09D7CE78 09CC1990 REG ACC
9 09D1A530 000A0005       6543 1 09D1A540 09CC1990 REG
10 09D19C38 000A0005       6542 1 09D19C48 09CC1990 REG
11 09CFFD18 000A0005      71811 1 09CFFD28 09CC1990 REG
12 09D00238 000A0005      63718 1 09D00248 09CC1990 REG
13 09D70918 000A0005       6746 1 09D70928 09CC1990 REG
14 09D01800 000A0005      15184 1 09D01810 09CC1990 REG
15 09F9B450 00330003       4098 1 09F9B460 09F79510 DIR
16 09F996D8 00330003       4097 1 09F996E8 09F79510 DIR
17 0A5C6548 00330006       4110 1 0A5C6558 09F7A990 DIR
18 09FB30D8 00330005       4104 1 09FB30E8 09F79F50 DIR CHG UPD FSYNC DIRTY
19 09FAB868 00330003       4117 1 09FAB878 09F79510 REG
20 0A492AB8 00330003       4123 1 0A492AC8 09F79510 REG
(0)> more (^C to quit) ? Interrupted
(0)> ino 09F79510 print mount table inode (IPMNT)
      DEV      NUMBER CNT      GNODE      IPMNT TYPE FLAGS
09F79510 00330003          0 1 09F79520 09F79510 NON CMNEW
forw     09F78C18 back     09F7A5B8 next     09F79510 prev     09F79510
gnode@   09F79520 number  00000000 dev      00330003 ipmnt    09F79510
flag     00000000 locks  00000000 bigexp  00000000 compress 00000000
cflag   00000002 count  00000001 event   FFFFFFFF movedfrag 00000000
openevent FFFFFFFF id      000052AB hip     09C9C330 nodelock 00000000
nodelock@ 09F79590 dquot[USR]00000000 dquot[GRP]00000000 dinode@ 09F7959C
cluster  00000000 size   0000000000000000
GNODE..... 09F79520
gn_type..... 00000000 gn_flags..... 00000000 gn_seg..... 00000000
gn_mwrcnt.... 00000000 gn_mrdcnt.... 00000000 gn_rdcnt..... 00000000
gn_wrcnt..... 00000000 gn_excnt..... 00000000 gn_rshcnt..... 00000000
gn_vnode..... 09F794E0 gn_rdev..... 00000000 gn_ops..... jfs_vops
gn_chan..... 00000000 gn_recl_k_lock. 00000000 gn_recl_k_lock@ 09F79554
gn_recl_k_event FFFFFFFF gn_filocks.... 00000000 gn_data..... 09F79510
gn_type..... NON
di_gen    32B69977 di_mode    00000000 di_nlink    00000000
di_acct   00000000 di_uid     00000000 di_gid      00000000
di_nblocks 00000000 di_acl     00000000
di_mtime  00000000 di_atime   00000000 di_ctime    00000000
di_size_hi 00000000 di_size_lo 00000000

```

```

VNODE..... 09F794E0
v_flag.... 00000000 v_count... 00000000 v_vfsgen.. 00000000
v_lock.... 00000000 v_lock@... 09F794EC v_vfsp.... 00000000
v_mvfsp... 00000000 v_gnode... 09F79520 v_next.... 00000000
v_vfsnext. 00000000 v_vfsprev. 00000000 v_pfsvnode 00000000
v_audit... 00000000
di_iplog   09F77F48 di_ipinode   09F798E8 di_ipind    09F797A0
di_ipinomap 09F79A30 di_ipdmap   09F79B78 di_ipsuper  09F79658
di_ipinodex 09F79CC0 di_jmpmnt   0B8E0B00
di_agsize   00004000 di_iagsize  00000800 di_logsidx  00000547
di_fperpage 00000008 di_fsbigexp 00000000 di_fscompress 00000001
(0)> ino 09F77F48 print log inode (di_iplog)
      DEV      NUMBER CNT      GNODE      IPMNT TYPE FLAGS
09F77F48 00330001      0 5 09F77F58 09F77F48 NON CMNEW
forw      09C9C310 back      09F785B0 next      09F77F48 prev      09F77F48
gnode@    09F77F58 number  00000000 dev      00330001 ipmnt    09F77F48
flag      00000000 locks  00000000 bigexp  00000000 compress 00000000
cflag     00000002 count  00000005 event  FFFFFFFF movedfrag 00000000
openevent FFFFFFFF id      0000529A hip      09C9C310 nodelock 00000000
nodelock@ 09F77FC8 dquot[USR]00000000 dquot[GRP]00000000 dinode@ 09F77FD4
cluster   00000000 size    0000000000000000
GNODE..... 09F77F58
gn_type.... 00000000 gn_flags.... 00000000 gn_seg..... 00007547
gn_mwrcnt... 00000000 gn_mrdcnt.... 00000000 gn_rdcnt.... 00000000
gn_wrcnt.... 00000000 gn_excnt.... 00000000 gn_rshcnt.... 00000000
gn_vnode.... 09F77F18 gn_rdev..... 00000000 gn_ops..... jfs_vops
gn_chan.... 00000000 gn_reclck_lock. 00000000 gn_reclck_lock@ 09F77F8C
gn_reclck_event FFFFFFFF gn_filocks.... 00000000 gn_data..... 09F77F48
gn_type.... NON
di_gen     32B69976 di_mode     00000000 di_nlink    00000000
di_acct    00000000 di_uid      00000000 di_gid      00000000
di_nblocks 00000000 di_acl      00000000
di_mtime   00000000 di_atime    00000000 di_ctime    00000000
di_size_hi 00000000 di_size_lo  00000000
VNODE..... 09F77F18
v_flag.... 00000000 v_count... 00000000 v_vfsgen.. 00000000
v_lock.... 00000000 v_lock@... 09F77F24 v_vfsp.... 00000000
v_mvfsp... 00000000 v_gnode... 09F77F58 v_next.... 00000000
v_vfsnext. 00000000 v_vfsprev. 00000000 v_pfsvnode 00000000
v_audit... 00000000
di_logptr  0000015A di_logsize  00000C00 di_logend   00000FF8
di_logsync 0005A994 di_nextsync 0013BBFC di_logxor   6C868513
di_llgeor  00000FE0 di_lllogxor 6CE29103 di_logx     0BB13200
di_logdgp  0B7E5BC0 di_loglock  4004B9EF di_loglock@ 09F7804C
logxlock   00000000 logxlock@   0BB13200 logflag     00000001
logppong   00000195 logcq.head  B69CAB7C logcq.tail  0BB13228
logcsn     00001534 logcrtc     0000000C loglcrt     B69CA97C
logeopm    00000001 logeopmc    00000002
logeopmq[0]@ 0BB13228 logeopmq[1]@ 0BB13268

```

hinode Subcommand

The **hino** subcommand prints inode hash lists.

Example

```

(0)> hino print hash inode buckets
      BUCKET HEAD      TIMESTAMP      LOCK COUNT
09C86000 1 0A285470 00000005 00000000 4
09C86010 2 0A284E08 00000006 00000000 3
09C86020 3 0A2843C8 00000006 00000000 3
09C86030 4 0A287EB8 00000006 00000000 3
09C86040 5 0A287330 00000005 00000000 3
09C86050 6 0A2867A8 00000006 00000000 4
09C86060 7 0A285FF8 00000007 00000000 3
09C86070 8 0A289D78 00000006 00000000 4
09C86080 9 0A289858 00000006 00000000 4

```

```

09C86090 10 0A33E2D8 00000005 00000000 4
09C860A0 11 0A33E7F8 00000005 00000000 4
09C860B0 12 0A33EE60 00000005 00000000 4
09C860C0 13 0A33F758 00000005 00000000 4
09C860D0 14 0A28AE20 00000005 00000000 3
09C860E0 15 0A28A670 00000005 00000000 3
09C860F0 16 0A33CE58 00000005 00000000 4
09C86100 17 0A33D9E0 00000006 00000000 4
09C86110 18 0A5FF6D0 00000008 00000000 4
09C86120 19 0A5FD060 00000009 00000000 4
09C86130 20 0A5FC390 00000009 00000000 4
(0)> more (^C to quit) ? Interrupted
(0)> hino 18 print hash inode bucket 18
HASH ENTRY( 18): 09C86110
      DEV      NUMBER CNT      GNODE      IPMNT TYPE FLAGS
0A5FF6D0 00330003      2523  0 0A5FF6E0 09F79510 REG
0A340E68 00330004      2524  0 0A340E78 09F78090 REG
0A28CA50 00330003     10677  0 0A28CA60 09F79510 DIR
0A1AFCA0 00330006      2526  0 0A1AFCB0 09F7A990 REG

```

icache Subcommand

The `fino` subcommand prints all the inode cache list.

Example

```

(0)> fino print free inode cache
      DEV      NUMBER CNT      GNODE      IPMNT TYPE FLAGS
1 09CABFA0 DEADBEEF      0  0 09CABFB0 09CA7178 CHR CMNOLINK
2 0A8D3A70 DEADBEEF      0  0 0A8D3A80 09F7A990 REG CMNOLINK
3 0A8F2528 DEADBEEF      0  0 0A8F2538 09CC6528 REG CMNOLINK
4 0A7C66E0 DEADBEEF      0  0 0A7C66F0 09F7A990 REG CMNOLINK
5 0A7BA568 DEADBEEF      0  0 0A7BA578 09F79F50 REG CMNOLINK
6 0A78EC68 DEADBEEF      0  0 0A78EC78 09F78090 REG CMNOLINK
7 0A7AF9B8 DEADBEEF      0  0 0A7AF9C8 09F79F50 REG CMNOLINK
8 0A7B9230 DEADBEEF      0  0 0A7B9240 09F79F50 REG CMNOLINK
9 0A8BDCA8 DEADBEEF      0  0 0A8BDCB8 09F79F50 LNK CMNOLINK
10 0A8BE978 DEADBEEF      0  0 0A8BE988 09F7A990 REG CMNOLINK
11 0A7C58C8 DEADBEEF      0  0 0A7C58D8 09F7A990 REG CMNOLINK
12 0A78D6A0 DEADBEEF      0  0 0A78D6B0 09F78090 REG CMNOLINK
13 0A7C4BF8 DEADBEEF      0  0 0A7C4C08 09F7A990 REG CMNOLINK
14 0A78ADA0 DEADBEEF      0  0 0A78ADB0 09F78090 REG CMNOLINK
15 0A7B8A80 DEADBEEF      0  0 0A7B8A90 09F79F50 REG CMNOLINK
16 0A8BC970 DEADBEEF      0  0 0A8BC980 09F7A990 REG CMNOLINK
17 0A8D1CF8 DEADBEEF      0  0 0A8D1D08 09F7A990 REG CMNOLINK
18 0A7AE160 DEADBEEF      0  0 0A7AE170 09F79F50 REG CMNOLINK
19 0A8EF998 DEADBEEF      0  0 0A8EF9A8 09CC6528 REG CMNOLINK
20 0A7C41B8 DEADBEEF      0  0 0A7C41C8 09F7A990 REG CMNOLINK
(0)> more (^C to quit) ? Interrupted
(0)> fino 1 print free inode slot 1
      DEV      NUMBER CNT      GNODE      IPMNT TYPE FLAGS
09CABFA0 DEADBEEF      0  0 09CABFB0 09CA7178 CHR CMNOLINK
forw     09CABFA0 back     09CABFA0 next     0A8EF708 prev     0042AE60
gnode@   09CABFB0 number   00000000 dev     DEADBEEF ipmnt     09CA7178
flag     00000000 locks     00000000 bigexp   00000000 compress 00000000
cflag    00000004 count     00000000 event    FFFFFFFF movedfrag 00000000
openevent FFFFFFFF id       00000045 hip     00000000 nodelock 00000000
nodelock@ 09CAC020 dquot[USR]00000000 dquot[GRP]00000000 dinode@ 09CAC02C
cluster  00000000 size     0000000000000000
GNODE..... 09CABFB0
gn_type..... 00000004 gn_flags..... 00000000 gn_seg..... 00000000
gn_mwrcnt..... 00000000 gn_mrdcnt..... 00000000 gn_rdcnt..... 00000000
gn_wrcnt..... 00000000 gn_excnt..... 00000000 gn_rshcnt..... 00000000
gn_vnode..... 09CABF70 gn_rdev..... 00030000 gn_ops..... jfs_vops
gn_chan..... 00000000 gn_recl_k_lock. 00000000 gn_recl_k_lock@ 09CABFE4
gn_recl_k_event FFFFFFFF gn_filocks.... 00000000 gn_data..... 09CABFA0
gn_type..... CHR

```

```

di_gen      00000000 di_mode      00000000 di_nlink    00000000
di_acct     00000000 di_uid       00000000 di_gid     00000000
di_nblocks  00000000 di_acl       00000000
di_mtime    32B67A97 di_atime    32B67A97 di_ctime   32B67B4B
di_size_hi  00000000 di_size_lo  00000000
di_rdev     00030000
VNODE..... 09CABF70
v_flag.... 00000000 v_count... 00000000 v_vfsgen.. 00000000
v_lock.... 00000000 v_lock@... 09CABF7C v_vfsp.... 00000000
v_mvfsp... 00000000 v_gnode... 09CABFB0 v_next.... 00000000
v_vfsnext. 09CABE28 v_vfsprev. 00000000 v_pfsvnode 00000000
v_audit... 00000000

```

rnode Subcommand

The `rno` subcommand prints the remote node structure.

Example

```

KDB(0)> rno 0A55D400 print rnode
RNODE..... 0A55D400
freef..... 00000000 freeb..... 00000000
hash..... 0A59A400 @vnode..... 0A55D40C
@gnode..... 0A55D43C @fh..... 0A55D480
fh[ 0]..... 0033000300000003 000A0000381F2F54
fh[16]..... A3FA0000000A0000 08002F53C1030000
flags..... 000001A0 error..... 00000000
lastr..... 00000000 cred..... 0A5757F8
altcred.... 00000000 unlcred.... 00000000
unlname.... 00000000 unlsvp.... 00000000
size..... 001C3A90 @attr..... 0A55D4C0
@attrtime... 0A55D520 sdname..... 00000000
sdvp..... 00000000 vh..... 00000885
sid..... 00000885 acl..... 00000000
aclsz..... 00000000 pcl..... 00000000
pclsz..... 00000000 @lock..... 0A55D548
rmevent..... FFFFFFFF
flags..... RWVP ACLINVALID PCLINVALID

```

cku Subcommand

The `cku` subcommand prints the client kudp private structure.

Example

```

KDB(0)> tpid print current thread
          SLOT NAME      STATE      TID PRI CPUID CPU FLAGS      WCHAN
thread+001E00 40*biod      SLEEP 0028DD 03C      000 00000420 05F49A6C
KDB(0)> f print current stack
thread+001E00 STACK:
[000191AC]e_block_thread+000214 ()
[0004EE74]sosbwait+000190 (??, ??)
[01AB64FC]cIntkudp_callit_addr+0005A8 (0A1F3904, 00000006, 01AD8E48, 2FF3B198,
01AD8E54, 2FF3B1C8, 00000001, 00061A80)
[01AB757C]cIntkudp_callit+000020 (??, ??, ??, ??, ??, ??, ??, ??)
[01AC4888]rfscall_progvers+0002C8 (??, ??, ??, ??, ??, ??, ??, ??)
[01ACA384]nfsread+0000D8 (??, ??, ??, ??, ??, ??, ??, ??)
[01ACB680]do_bio+000268 (??, ??)
[01AC9868]async_daemon_x+000060 (??, ??)
[01AC9920]async_daemon+000044 ()
[000036F0].sys_call+000000 ()
KDB(0)> cku @r22 print client kudp information
CKU_PRIVATE..... 0A1F3900
flags..... 00000020 @client..... 0A1F3904
client.auth..... 054EE600 client.ops..... 01AC1898
client.private.... 0A1F3900
retrys..... 00000001 sock..... 05F49A00
@addr..... 0A1F3918 @err..... 0A1F3928

```

```

addr.sin_len..... 00000000 addr.sin_family.. 00000002
addr.sin_port..... 00000801 addr.sin_addr.... 96B70101
err.RE_errno..... 00000000 err.RE_why..... 00000000
@outxdr..... 0A1F3934 @inxdr..... 0A1F394C
outxdr.x_op..... 00000000 outxdr.x_ops..... 01AC1430
outxdr.x_public... 00000000 outxdr.x_private.. 0A5A6078
outxdr.x_base..... 05E9F600 outxdr.x_handy... 000021E8
inxdr.x_op..... 00000001 inxdr.x_ops..... 01AC1430
inxdr.x_public... 00000000 inxdr.x_private.. 05E0A7A6
inxdr.x_base..... 05E0A700 inxdr.x_handy.... 00000000
outpos..... 00000014 outbuf..... 0A5A6000
inbuf..... 05E0A746 inmbuf..... 05E0A700
cred..... 0A5757F8 timers..... 0A3CC788
timeall..... 0A3CC798 feedback..... 01AD90AC
xid..... 5397F6B5 trb..... 0A54E280
@buflock..... 0A1F3990 bufevent..... FFFFFFFF
flags..... INTR

```

vnode Subcommand

The `vno` subcommand prints the virtual node structure.

Example

```

(0)> vnode print vnode table
          COUNT VFSGEN   GNODE   VFSP  DATAPTR TYPE  FLAGS
106 09D227B0     3     0 09D227F0 056D183C 00000000 REG
126 09D1AB68     1     0 09D1ABA8 056D183C 00000000 REG
130 09D196E8     1     0 09D19728 056D183C 00000000 REG
135 09D18B60     1     0 09D18BA0 056D183C 05CC2D00 SOCK
140 09D17E90     1     0 09D17ED0 056D183C 05D3F300 SOCK
143 09D17970     1     0 09D179B0 056D183C 05CC2A00 SOCK
148 09D17078     1     0 09D170B8 056D183C 05CC2800 SOCK
154 09D14DE0     1     0 09D14E20 056D183C 00000000 REG
162 09D13818     1     0 09D13858 056D183C 05D30E00 SOCK
165 09D0D948     1     0 09D0D988 056D183C 00000000 DIR
166 09D0D800     1     0 09D0D840 056D183C 00000000 DIR
167 09D0D6B8     1     0 09D0D6F8 056D183C 00000000 DIR
168 09D0D570     1     0 09D0D5B0 056D183C 00000000 DIR
170 09D0D2E0     1     0 09D0D320 056D183C 00000000 DIR
171 09D0D198     1     0 09D0D1D8 056D183C 00000000 DIR
172 09D0D050     1     0 09D0D090 056D183C 00000000 DIR
173 09D0CF08     1     0 09D0CF48 056D183C 00000000 DIR
174 09D0CDC0     1     0 09D0CE00 056D183C 00000000 DIR
175 09D0CC78     1     0 09D0CCB8 056D183C 00000000 DIR
176 09D0CB30     1     0 09D0CB70 056D183C 00000000 DIR
(0)> more (^C to quit) ? Interrupted
(0)> vnode 106 print vnode slot 106
          COUNT VFSGEN   GNODE   VFSP  DATAPTR TYPE  FLAGS
106 09D227B0     3     0 09D227F0 056D183C 00000000 REG
v_flag.... 00000000 v_count... 00000003 v_vfsgen.. 00000000
v_lock.... 00000000 v_lock@... 09D227BC v_vfsp.... 056D183C
v_mvfsp... 00000000 v_gnode... 09D227F0 v_next.... 00000000
v_vfsnext. 09D22668 v_vfsprev. 09D22B88 v_pfsvnode 00000000
v_audit... 00000000

```

mount Subcommand

The `vfs` subcommand prints the virtual file system table.

Example

```

(0)> vfs print vfs table
          GFS      MNTD MNTDOVER  VNODES   DATA TYPE  FLAGS
1 056D183C 0024F268 09CC08B8 00000000 0A5AADA0 0B221F68 JFS  DEVMOUNT
... /dev/hd4 mounted over /
2 056D18A4 0024F268 09CC2258 09CC0B48 0A545270 0B221F00 JFS  DEVMOUNT
... /dev/hd2 mounted over /usr

```

```

3 056D1870 0024F268 09CC3820 09CC2DE0 09D913A8 0B221E30 JFS    DEVMOUNT
... /dev/hd9var mounted over /var
4 056D1808 0024F268 09CC6DF0 09CC6120 0A7DC1E8 0B221818 JFS    DEVMOUNT
... /dev/hd3 mounted over /tmp
5 056D18D8 0024F268 09D0BFA8 09D0B568 09D95500 0B2412F0 JFS    DEVMOUNT
... /dev/hd1 mounted over /home
6 056D190C 0024F2C8 0B243C0C 09D0C238 0B9F6A0C 0B230500 NFS    READONLY REMOTE
... /pvt/tools mounted over /pvt/tools
7 056D1940 0024F2C8 0B7E440C 09D0CB30 0B985C0C 0B230A00 NFS    READONLY REMOTE
... /pvt/base mounted over /pvt/base
8 056D1974 0024F2C8 0B7E4A0C 09D0CC78 0B7E4A0C 0B230C00 NFS    READONLY REMOTE
... /pvt/periph mounted over /pvt/periph
9 056D19A8 0024F2C8 0B7E4E0C 09D0CDC0 0B89000C 0B230E00 NFS    READONLY REMOTE
... /nfs mounted over /nfs
10 056D19DC 0024F2C8 0B89020C 09D0CF08 0B89840C 0B230000 NFS    READONLY REMOTE
... /tcp mounted over /tcp
(0)> vfs 5 print vfs slot 5
          GFS      MNTD MNTDOVER  VNODES      DATA TYPE  FLAGS
5 056D18D8 0024F268 09D0BFA8 09D0B568 09D95500 0B2412F0 JFS    DEVMOUNT
... /dev/hd1 mounted over /home
vfs_next..... 056D190C vfs_count.... 00000001 vfs_mntd..... 09D0BFA8
vfs_mntdover. 09D0B568 vfs_vnodes... 09D95500 vfs_count.... 00000001
vfs_number... 00000009 vfs_bsize.... 00001000 vfs_mdata.... 0B7E8E80
vmt_revision. 00000001 vmt_length.. 00000070 vfs_fsid.... 000A0008 00000003
vmt_vfsnumber 00000009 vfs_date..... 32B67BFF vfs_flag..... 00000004
vmt_gfstype.. 00000003 @vmt_data.... 0B7E8EA4 vfs_lock..... 00000000
vfs_lock@.... 056D1904 vfs_type..... 00000003 vfs_ops..... jfs_vfsops
VFS_GFS.. gfs+0000000
gfs_data. 00000000 gfs_flag. INIT VERSION4 VERSION42 VERSION421
gfs_ops.. jfs_vfsops gn_ops... jfs_vops gfs_name. jfs
gfs_init. jfs_init gfs_rinit jfs_rootinit gfs_type. JFS
gfs_hold. 00000013
VFS_MNTD.. 09D0BFA8
v_flag.... 00000001 v_count... 00000001 v_vfsgen.. 00000000
v_lock.... 00000000 v_lock@... 09D0BFB4 v_vfsp.... 056D18D8
v_mvfsp... 00000000 v_gnode... 09D0BFE8 v_next.... 00000000
v_vfsnext.. 00000000 v_vfsprev. 09D730A0 v_pfsvnode 00000000
v_audit... 00000000 v_flag.... ROOT
VFS_MNTDOVER.. 09D0B568
v_flag.... 00000000 v_count... 00000001 v_vfsgen.. 00000000
v_lock.... 00000000 v_lock@... 09D0B574 v_vfsp.... 056D183C
v_mvfsp... 056D18D8 v_gnode... 09D0B5A8 v_next.... 00000000
v_vfsnext.. 09D0A230 v_vfsprev. 09D0C0F0 v_pfsvnode 00000000
v_audit... 00000000
VFS_VNODES LIST...
          COUNT VFSGEN  GNODE      VFSP  DATAPTR TYPE FLAGS
1 09D95500 0 0 09D95540 056D18D8 00000000 REG
2 09D94AC0 0 0 09D94B00 056D18D8 00000000 DIR
3 09D91DE8 0 0 09D91E28 056D18D8 00000000 REG
4 09D91A10 0 0 09D91A50 056D18D8 00000000 DIR
5 09D8EFC8 0 0 09D8F008 056D18D8 00000000 REG
6 09D8EBF0 0 0 09D8EC30 056D18D8 00000000 DIR
7 09D8C580 0 0 09D8C5C0 056D18D8 00000000 REG
8 09D8C060 0 0 09D8C0A0 056D18D8 00000000 DIR
9 09D8A058 0 0 09D8A098 056D18D8 00000000 REG
10 09D89C80 0 0 09D89CC0 056D18D8 00000000 DIR
11 09D89240 0 0 09D89280 056D18D8 00000000 REG
...
          COUNT VFSGEN  GNODE      VFSP  DATAPTR TYPE FLAGS
63 09D73478 0 0 09D734B8 056D18D8 00000000 REG
64 09D730A0 0 0 09D730E0 056D18D8 00000000 DIR
65 09D0BFA8 1 0 09D0BFE8 056D18D8 00000000 DIR  ROOT

```

specnode Subcommand

The **specnode** subcommand prints special device nodes.

Example

```
(0)> file file+002880 print file entry
COUNT          OFFSET      DATA TYPE  FLAGS
217 file+002880    6 000000000002818F 056CE314 VNODE  READ WRITE
f_flag..... 00000003 f_count..... 00000006
f_msgcount..... 0000 f_type..... 0001
f_data..... 056CE314 f_offset... 000000000002818F
f_dir_off..... 00000000 f_cred..... 0B988E58
f_lock@..... 004B18B8 f_lock..... 00000000
f_offset_lock@. 004B18BC f_offset_lock.. 00000000
f_vinfo..... 00000000 f_ops..... 00250FC0 vnodefops+000000
VNODE..... 056CE314
v_flag... 00000000 v_count... 00000002 v_vfsgen.. 00000000
v_lock... 00000000 v_lock@... 056CE320 v_vfsp.... 01AC9840
v_mvfsp... 00000000 v_gnode... 0B2215C8 v_next.... 00000000
v_vfsnext. 00000000 v_vfsprev. 00000000 v_pfsvnode 09CD5D88
v_audit... 00000000
(0)> gno 0B2215C8 print gnode entry
GNODE..... 0B2215C8
gn_type..... 00000004 gn_flags..... 00000000 gn_seg..... 007FFFFF
gn_mwrcnt.... 00000000 gn_mrdcnt.... 00000000 gn_rdcnt..... 00000000
gn_wrcnt..... 00000000 gn_excnt..... 00000000 gn_rshcnt.... 00000000
gn_vnode..... 056CE314 gn_rdev..... 000E0000 gn_ops..... spec_vnops
gn_chan..... 00000000 gn_recl_k_lock. 00000000 gn_recl_k_lock@ 0B2215FC
gn_recl_k_event FFFFFFFF gn_filocks... 00000000 gn_data..... 0B2215B8
gn_type..... CHR
(0)> specno 0B2215B8 print special node entry
SPECNODE..... 0B2215B8
sn_next..... 00000000 sn_count.... 00000001 sn_lock..... 00000000
sn_gnode..... 0B2215C8 sn_pfsnode.. 09CD5DC8 sn_attr..... 00000000
sn_dev..... 000E0000 sn_chan..... 00000000 sn_vnode..... 056CE314
sn_ops..... 00275518 sn_devnode... 0B221C80 sn_type..... CHR
SN_VNODE..... 056CE314
v_flag... 00000000 v_count... 00000002 v_vfsgen.. 00000000
v_lock... 00000000 v_lock@... 056CE320 v_vfsp.... 01AC9840
v_mvfsp... 00000000 v_gnode... 0B2215C8 v_next.... 00000000
v_vfsnext. 00000000 v_vfsprev. 00000000 v_pfsvnode 09CD5D88
v_audit... 00000000
SN_GNODE..... 0B2215C8
gn_type..... 00000004 gn_flags..... 00000000 gn_seg..... 007FFFFF
gn_mwrcnt.... 00000000 gn_mrdcnt.... 00000000 gn_rdcnt..... 00000000
gn_wrcnt..... 00000000 gn_excnt..... 00000000 gn_rshcnt.... 00000000
gn_vnode..... 056CE314 gn_rdev..... 000E0000 gn_ops..... spec_vnops
gn_chan..... 00000000 gn_recl_k_lock. 00000000 gn_recl_k_lock@ 0B2215FC
gn_recl_k_event FFFFFFFF gn_filocks... 00000000 gn_data..... 0B2215B8
gn_type..... CHR
SN_PFSGNODE..... 09CD5DC8
gn_type..... 00000004 gn_flags..... 00000000 gn_seg..... 00000000
gn_mwrcnt.... 00000000 gn_mrdcnt.... 00000000 gn_rdcnt..... 00000000
gn_wrcnt..... 00000000 gn_excnt..... 00000000 gn_rshcnt.... 00000000
gn_vnode..... 09CD5D88 gn_rdev..... 000E0000 gn_ops..... jfs_vops
gn_chan..... 00000000 gn_recl_k_lock. 00000000 gn_recl_k_lock@ 09CD5DFC
gn_recl_k_event FFFFFFFF gn_filocks... 00000000 gn_data..... 09CD5DB8
gn_type..... CHR
```

devnode Subcommand

The **devno** subcommand prints device nodes.

Example

```
(0)> devno print device node table
          DEV CNT SPECNODE   GNODE   LASTR   PDATA TYPE
1 0B241758 00300000    1 0B2212E0 0B241768 00000000 05CB4E00 CHR
2 0B221C18 00100000    1 00000000 0B221C28 00000000 00000000 CHR
3 0B221940 00110000    2 00000000 0B221950 00000000 00000000 BLK
4 0B221870 00020000    1 0B221140 0B221880 00000000 00000000 CHR
```



```

5 0B7E5A10 00120001 2 00000000 0B7E5A20 00000000 00000000 BLK
6 0B241070 00020001 1 0B8A3EF0 0B241080 00000000 00000000 CHR
7 0B2219A8 00020002 1 0B221008 0B2219B8 00000000 00000000 CHR
8 0B2218D8 00130000 1 00000000 0B2218E8 00000000 00000000 CHR
9 0B7E5BB0 00330001 1 00000000 0B7E5BC0 00000000 00000000 BLK
10 0B221A10 00130001 1 00000000 0B221A20 00000000 00000000 CHR
11 0B241008 00330002 1 00000000 0B241018 00000000 00000000 BLK
12 0B7E59A8 00130002 1 00000000 0B7E59B8 00000000 00000000 CHR
13 0B7E5C18 00330003 1 00000000 0B7E5C28 00000000 00000000 BLK
14 0B7E5808 00130003 1 00000000 0B7E5818 00000000 00000000 CHR
15 0B7E5A78 00330004 1 00000000 0B7E5A88 00000000 00000000 BLK
16 0B7E5C80 00330005 1 00000000 0B7E5C90 00000000 00000000 BLK
17 0B7E5CE8 00330006 1 00000000 0B7E5CF8 00000000 00000000 BLK
18 0B2416F0 00040000 1 0B2211A8 0B241700 00000000 00000000 MPC
19 0B221BB0 00150000 3 0B221688 0B221BC0 00000000 05CC3E00 CHR
20 0B2410D8 00060000 1 0B221480 0B2410E8 00000000 00000000 CHR
(0)> more (^C to quit) ? Interrupted
(0)> devno 3 print device node slot 3
          DEV CNT SPECNODE  GNODE  LASTR  PDATA TYPE
3 0B221940 00110000  2 00000000 0B221950 00000000 00000000 BLK
forw..... 00DD6CD8 back..... 00DD6CD8 lock..... 00000000
GNODE..... 0B221950
gn_type..... 00000003 gn_flags..... 00000000 gn_seg..... 007FFFFF
gn_mwrcnt.... 00000000 gn_mrdcnt.... 00000000 gn_rdcnt..... 00000000
gn_wrcnt..... 00000002 gn_excnt..... 00000000 gn_rshcnt.... 00000000
gn_vnode..... 00000000 gn_rdev..... 00110000 gn_ops..... 00000000
gn_chan..... 00000000 gn_recl_k_lock. 00000000 gn_recl_k_lock@ 0B221984
gn_recl_k_event 00000000 gn_filocks.... 00000000 gn_data..... 0B221940
gn_type..... BLK
SPECNODES..... 00000000

```

fifonode Subcommand

The `fifono` subcommand prints fifo nodes.

Example

```

(0)> fifono print fifo node table
          PFSGNODE SPECNODE  SIZE  RCNT  WCNT TYPE FLAG
1 056D1C08 09D15EC8 0B2210D8 00000000  1  1 FIFO WVRT
2 056D1CA8 09D1BB08 0B7E5070 00000000  1  1 FIFO RBLK WVRT
(0)> fifono 1 print fifo node slot 1
          PFSGNODE SPECNODE  SIZE  RCNT  WCNT TYPE FLAG
1 056D1C08 09D15EC8 0B2210D8 00000000  1  1 FIFO WVRT
ff_forw.... 00DD6D44 ff_back.... 00DD6D44 ff_dev..... FFFFFFFF
ff_poll.... 00000001 ff_rptr.... 00000000 ff_wptr.... 00000000
ff_revent.. FFFFFFFF ff_wevent.. FFFFFFFF ff_buf..... 056D1C34
SPECNODE..... 0B2210D8
sn_next..... 00000000 sn_count.... 00000001 sn_lock..... 00000000
sn_gnode.... 0B2210E8 sn_pfsgnode.. 09D15EC8 sn_attr..... 00000000
sn_dev..... FFFFFFFF sn_chan..... 00000000 sn_vnode.... 056CE070
sn_ops..... 002751B0 sn_devnode... 056D1C08 sn_type..... FIFO
SN_VNODE..... 056CE070
v_flag.... 00000000 v_count... 00000002 v_vfsgen.. 00000000
v_lock.... 00000000 v_lock@... 056CE07C v_vfsp.... 01AC9810
v_mvfsp... 00000000 v_gnode... 0B2210E8 v_next.... 00000000
v_vfsnext. 00000000 v_vfsprev. 00000000 v_pfsvnode 09D15E88
v_audit... 00000000
SN_GNODE..... 0B2210E8
gn_type..... 00000008 gn_flags..... 00000000 gn_seg..... 007FFFFF
gn_mwrcnt.... 00000000 gn_mrdcnt.... 00000000 gn_rdcnt..... 00000000
gn_wrcnt..... 00000000 gn_excnt..... 00000000 gn_rshcnt.... 00000000
gn_vnode.... 056CE070 gn_rdev..... FFFFFFFF gn_ops..... fifo_vnops
gn_chan..... 00000000 gn_recl_k_lock. 00000000 gn_recl_k_lock@ 0B22111C
gn_recl_k_event 00000000 gn_filocks.... 00000000 gn_data..... 0B2210D8
gn_type..... FIFO
SN_PFSGNODE..... 09D15EC8

```

```

gn_type..... 00000008 gn_flags..... 00000000 gn_seg..... 00000000
gn_mwrcnt..... 00000000 gn_mrdcnt..... 00000000 gn_rdcnt..... 00000000
gn_wrcnt..... 00000000 gn_excnt..... 00000000 gn_rshcnt..... 00000000
gn_vnode..... 09D15E88 gn_rdev..... 000A0005 gn_ops..... jfs_vops
gn_chan..... 00000000 gn_recl_k_lock. 00000000 gn_recl_k_lock@ 09D15EFC
gn_recl_k_event FFFFFFFF gn_filocks.... 00000000 gn_data..... 09D15EB8
gn_type..... FIFO

```

hnode Subcommand

The **hno** subcommand prints hash node table.

Example

```

(0)> hno print hash node table
          BUCKET HEAD      LOCK      COUNT
hnodetable+000000  1  0B241758 00000000  2
hnodetable+0000C0 17  0B221940 00000000  1
hnodetable+00012C 26  056D1C08 00000000  1
hnodetable+000180 33  0B221870 00000000  1
hnodetable+00018C 34  0B7E5A10 00000000  2
hnodetable+000198 35  0B2219A8 00000000  1
hnodetable+000240 49  0B2218D8 00000000  1
hnodetable+00024C 50  0B7E5BB0 00000000  2
hnodetable+000258 51  0B241008 00000000  2
hnodetable+000264 52  0B7E5C18 00000000  2
hnodetable+000270 53  0B7E5A78 00000000  1
hnodetable+00027C 54  0B7E5C80 00000000  1
hnodetable+000288 55  0B7E5CE8 00000000  1
hnodetable+000300 65  0B2416F0 00000000  1
hnodetable+0003C0 81  0B221BB0 00000000  1
hnodetable+000480 97  0B2410D8 00000000  1
hnodetable+00048C 98  0B221B48 00000000  1
hnodetable+000540 113 0B7E5AE0 00000000  1
hnodetable+00054C 114 0B7E5EF0 00000000  1
hnodetable+000600 129 0B7E5B48 00000000  1
(0)> more (^C to quit) ? Interrupted
(0)> hno 34 print hash node bucket 34
HASH ENTRY( 34): 00DD6DA4
          DEV CNT SPECNODE      GNODE      LASTR      PDATA TYPE
  1 0B7E5A10 00120001  2 00000000 0B7E5A20 00000000 00000000 BLK
  2 0B241070 00020001  1 0B8A3EF0 0B241080 00000000 00000000 CHR

```

System Table Subcommands for the KDB Kernel Debugger and kdb Command

var Subcommand

The **var** subcommand prints the **var** structure and the system configuration of the machine.

Example

```

KDB(7)> var print var information
var_hdr.var_vers..... 00000000 var_hdr.var_gen..... 00000045
var_hdr.var_size..... 00000030
v_iostrun..... 00000001 v_leastpriv..... 00000000
v_autost..... 00000001 v_memscrub..... 00000000
v_maxup..... 200
v_bufhw..... 20 v_mbufhw..... 32768
v_maxpout..... 0 v_minpout..... 0
v_clist..... 16384 v_fullcore..... 00000000
v_ncpus..... 8 v_ncpus_cfg..... 8
v_initlvl..... 0 0 0 0
v_lock..... 200 ve_lock..... 00D3FA18 flox+003200
v_file..... 2303 ve_file..... 0042EFE8 file+01AFD0

```

```

v_proc..... 131072 ve_proc..... E305D000 proc+05D000
vb_proc..... E3000000 proc+000000
v_thread..... 262144 ve_thread..... E6046F80 thread+046F80
vb_thread..... E6000000 thread+000000
VMM Tunable Variables:
minfree..... 120 maxfree..... 128
minperm..... 12872 maxperm..... 51488
pfrsvdblks..... 13076
(7)> more (^C to quit) ? continue
npswarn..... 512 npskill..... 128
minpgahead..... 2 maxpgahead..... 8
maxpdtblks..... 4 numsched..... 4
htabscale..... FFFFFFFF aptscale..... 00000000
pd_npages..... 00080000
_SYSTEM_CONFIGURATION:
architecture.... 00000002 POWER_PC
implementation... 00000010 POWER_604
version..... 00040004
width..... 00000020 ncpus..... 00000008
cache_attrib.... 00000001 CACHE separate I and D
icache_size.... 00004000 dcache_size..... 00004000
icache_asc..... 00000004 dcache_asc..... 00000004
icache_block.... 00000020 dcache_block.... 00000020
icache_line.... 00000040 dcache_line..... 00000040
L2_cache_size... 00100000 L2_cache_asc.... 00000001
tlb_attrib..... 00000001 TLB separate I and D
itlb_size..... 00000040 dtlb_size..... 00000040
itlb_asc..... 00000002 dtlb_asc..... 00000002
priv_lck_cnt.... 00000000 prob_lck_cnt.... 00000000
resv_size..... 00000020 rtc_type..... 00000002
virt_alias..... 00000000 cach_cong..... 00000000
model_arch..... 00000001 model_impl..... 00000002
Xint..... 000000A0 Xfrac..... 00000003

```

devsw Subcommand

The `dev` subcommand shows the **device switch** table ala `lldb`.

Example

```

KDB(0)> dev
Slot address 054F5040
MAJ#001 OPEN CLOSE READ WRITE
      .syopen .nulldev .syread .sywrite
      IOCTL STRATEGY TTY SELECT
      .syioctl .nodev 00000000 .syselect
      CONFIG PRINT DUMP MPX
      .nodev .nodev .nodev .nodev
      REVOKE DSDPTR SELPTR OPTS
      .nodev 00000000 00000000 00000002

Slot address 054F5080
MAJ#002 OPEN CLOSE READ WRITE
      .nulldev .nulldev .mmread .mmwrite
      IOCTL STRATEGY TTY SELECT
      .nodev .nodev 00000000 .nodev
      CONFIG PRINT DUMP MPX
      .nodev .nodev .nodev .nodev
      REVOKE DSDPTR SELPTR OPTS
      .nodev 00000000 00000000 00000002

(0)> more (^C to quit) ? ^C quit
KDB(0)> devsw 4 device switch of major 0x4
Slot address 05640100
MAJ#004 OPEN CLOSE READ WRITE
      .conopen .conclose .conread .conwrite
      IOCTL STRATEGY TTY SELECT
      .conioctl .nodev 00000000 .conselect
      CONFIG PRINT DUMP MPX

```

```

.conconfig      .nodev      .nodev      .conmpx
REVOKE         DSDPTR     SELPTR     OPTS
.conrevoke     00000000  00000000  00000006

```

timer Subcommand

The **trb** subcommand displays timer request block ala **lldb**. Subcommand option are selected thru the menu, or directly entered.

Example

```

KDB(4)> trb timer request block subcommand usage
Usage: trb [CPU selector] [1-9]
CPU selector is '*' for all CPUs, 'cpu n' for CPU n, default is current CPU
Timer Request Block Information Menu
  1. TRB Maintenance Structure - Routine Addresses
  2. System TRB
  3. Thread Specified TRB
  4. Current Thread TRB's
  5. Address Specified TRB
  6. Active TRB Chain
  7. Free TRB Chain
  8. Clock Interrupt Handler Information
  9. Current System Time - System Timer Constants
Please enter an option number: <CR/LF>
KDB(4)> trb * 6 print all active timer request blocks
CPU #0 Active List
  CPU PRI      ID  SECS  NSECS  DATA FUNC
05689080 0000 0005 FFFFFFFE 00003BBA 23C3B080 05689080 sys_timer+000000
05689600 0000 0003 FFFFFFFE 00003BBA 27DAC680 00000000 pffastsched+000000
05689580 0000 0003 FFFFFFFE 00003BBA 2911BD80 00000000 pflowsched+000000
0B05A600 0000 0005 00001751 00003BBA 2ADBC480 0B05A618 rtsleep_end+000000
05689500 0000 0003 FFFFFFFE 00003BBB 23186B00 00000000 if_slowsched+000000
0B05A480 0000 0003 FFFFFFFE 00003BBF 2D5B4980 00000000 01B633F0
CPU #1 Active List
  CPU PRI      ID  SECS  NSECS  DATA FUNC
05689100 0001 0005 FFFFFFFE 00003BBA 23C38E80 05689100 sys_timer+000000
CPU #2 Active List
  CPU PRI      ID  SECS  NSECS  DATA FUNC
05689180 0002 0005 FFFFFFFE 00003BBA 23C37380 05689180 sys_timer+000000
0B05A500 0002 0005 00001525 00003BE6 0CFF9500 0B05A518 rtsleep_end+000000
CPU #3 Active List
  CPU PRI      ID  SECS  NSECS  DATA FUNC
05689200 0003 0005 FFFFFFFE 00003BBA 23C39F80 05689200 sys_timer+000000
(4)> more (^C to quit) ? continue
05689880 0003 0005 00000003 00003BBB 01B73180 00000000 sched_timer_post+000000
0B05A580 0003 0005 00000001 00003BBB 0BCA7300 0000000E interval_end+000000
CPU #4 Active List
  CPU PRI      ID  SECS  NSECS  DATA FUNC
05689280 0004 0005 FFFFFFFE 00003BBA 23C3A980 05689280 sys_timer+000000
CPU #5 Active List
  CPU PRI      ID  SECS  NSECS  DATA FUNC
05689300 0005 0005 FFFFFFFE 00003BBA 23C39800 05689300 sys_timer+000000
05689780 0005 0005 FFFFFFFF 00003BBF 1B052C00 05C62C40 01ADD6FC
CPU #6 Active List
  CPU PRI      ID  SECS  NSECS  DATA FUNC
05689380 0006 0005 FFFFFFFE 00003BBA 23C3C200 05689380 sys_timer+000000
CPU #7 Active List
  CPU PRI      ID  SECS  NSECS  DATA FUNC
05689400 0007 0005 FFFFFFFE 00003BBA 23C38180 05689400 sys_timer+000000
05689680 0007 0003 FFFFFFFE 00003BBA 2DDD3480 00000000 threadtimer+000000
KDB(4)> trb cpu 1 6 print active list of processor 1
CPU #1 TRB #1 on Active List
Timer address.....05689100
trb->to_next.....00000000
trb->knext.....00000000
trb->kprev.....00000000

```

```

Owner id (-1 for dev drv).....FFFFFFFFE
Owning processor.....00000001
Timer flags.....00000013    PENDING ACTIVE INCINTERVAL
trb->timerid.....00000000
trb->eventlist.....FFFFFFFFF
trb->timeout.it_interval.tv_sec...00000000
trb->timeout.it_interval.tv_nsec...00000000
Next scheduled timeout (secs).....00003BBA
Next scheduled timeout (nanosecs)..23C38E80
Completion handler.....000B3BA4    sys_timer+0000000
Completion handler data.....05689100
Int. priority .....00000005
Timeout function.....00000000    00000000
KDB(4)>

```

slk and clk Subcommands

The **slk** and **clk** subcommands print the specified simple or complex lock. If instrumentation is set at boot time, instrumentation information are displayed.

Example

```

KDB(1)> slk B69F2DF0 print simple lock
Simple Lock Instrumented: vmmseg+69F2DF0
      _slock: 00011C99    thread_owner: 0011C99
.....acquisitions number:      16
.....misses number:            0
..sleeping misses number:      0
.....lockname: 00FA097D    flox+206165
...link register of lock: 0007CFCC    .pfget+00023C
.....caller of lock: 00011C99
.....cpu id of lock: 00000002
.link register of unlock: 0007D8EC    .pfget+000B5C
.....caller of unlock: 00011C99
.....cpu id of unlock: 00000002
KDB(0)> clk ndd_lock print complex lock
Complex Lock Instrumented: ndd_lock
...._clock.status: 20001553    _cTock.flags 0000    _clock.rdepth 0000
.....status: WANT_WRITE
.....thread_owner: 0001553
.....acquisitions number:      2
.....misses number:            0
..sleeping misses number:      0
.....lockname: 00D2FFFF    file+8BDFE7
...link register of lock: 00047874    .ns_init+00002C
.....caller of lock: 00000003
.....cpu id of lock: 00000000
.link register of unlock: 00000000    00000000
.....caller of unlock: 00000000
.....cpu id of unlock: 00000000
KDB(1)>

```

iplcb Subcommand

The **ipl** subcommand prints processor info tables, or the specified one.

Example

```

KDB(4)> ipl * print ipl control blocks
      INDEX  PHYS_ID INT_AREA ARCHITEC IMPLEMEN  VERSION
0038ECD0    0 00000000 FF100000 00000002 00000008 00010005
0038ED98    1 00000001 FF100080 00000002 00000008 00010005
0038EE60    2 00000002 FF100100 00000002 00000008 00010005
0038EF28    3 00000003 FF100180 00000002 00000008 00010005
0038EFF0    4 00000004 FF100200 00000002 00000008 00010005
0038F0B8    5 00000005 FF100280 00000002 00000008 00010005
0038F180    6 00000006 FF100300 00000002 00000008 00010005
0038F248    7 00000007 FF100380 00000002 00000008 00010005

```

```

KDB(4)> ipl print current processor information
Processor Info 4 [0038EFF0]
num_of_structs.....00000008 index.....00000004
struct_size.....000000C8 per_buc_info_offset....0001D5D0
proc_int_area.....FF100200 proc_int_area_size....00000010
processor_present.....00000001 test_run.....0000006A
test_stat.....00000000 link.....00000000
link_address.....00000000 phys_id.....00000004
architecture.....00000002 implementation.....00000008
version.....00010005 width.....00000020
cache_attrib.....00000003 coherency_size.....00000020
resv_size.....00000020 icache_block.....00000020
dcache_block.....00000020 icache_size.....00008000
dcache_size.....00008000 icache_line.....00000040
dcache_line.....00000040 icache_asc.....00000008
dcache_asc.....00000008 L2_cache_size.....00100000
L2_cache_asc.....00000001 tlb_attrib.....00000003
itlb_size.....00000100 dtlb_size.....00000100
itlb_asc.....00000002 dtlb_asc.....00000002
slb_attrib.....00000000 islb_size.....00000000
dslb_size.....00000000 islb_asc.....00000000
(4)> more (^C to quit) ? continue
dslb_asc.....00000000 priv_lck_cnt.....00000000
prob_lck_cnt.....00000000 rtc_type.....00000001
rtcXint.....00000000 rtcXfrac.....00000000
busCfreq_HZ.....00000000 tbCfreq_HZ.....00000000
System info [0038E534]
num_of_procs.....00000008 coherency_size.....00000020
resv_size.....00000020 arb_cr_addr.....00000000
phys_id_reg_addr.....00000000 num_of_bsrr.....00000000
bsrr_addr.....00000000 tod_type.....00000000
todr_addr.....FF0000C0 rsr_addr.....FF62006C
pkrsr_addr.....FF620064 prcr_addr.....FF620060
sssr_addr.....FF001000 sir_addr.....FF100000
scr_addr.....00000000 dscr_addr.....00000000
nvram_size.....00022000 nvram_addr.....FF600000
vpd_rom_addr.....00000000 ipl_rom_size.....00100000
ipl_rom_addr.....07F00000 g_mfrr_addr.....FF107F80
g_tb_addr.....00000000 g_tb_type.....00000000
g_tb_mult.....00000000 SP_Error_Log_Table.....0001C000
pcccr_addr.....00000000 spocr_addr.....FF620068
pfeivr_addr.....FF00100C access_id_waddr.....00000000
loc_waddr.....00000000 access_id_raddr.....00000000
(4)> more (^C to quit) ? continue
loc_raddr.....00000000 architecture.....00000001
implementation.....00000002 pkg_descriptor.....rs6ksmp
KDB(4)>

```

trace Subcommand

This displays data in the kernel trace buffers. Data is entered into these buffers via the shell subcommand "trace", if the shell subcommand has not been invoked prior to using the debugger subcommand then the trace buffers will be empty.

A search facility has been added to trace which allows specification of certain search criteria which will be used to restrict the set of displayed trace entries. The subcommand line to trace now has the form:

```
trace [-h] [hook[:subhook]]... [#data]... [-c channel]
```

Where:

-h displays the trace headers, which are trace driver data structures.

-c selects the channel number from the subcommand line, otherwise it will be prompted for.

The **trace** subcommand displays the contents of the specified channel, based on any search criteria which was entered on the subcommand line.

Example

```
KDB(0)> trace -c 0 1b0 1b1 1b2 1b3 1b4 1b5 1b6 1b7 1b8 1b9
trace VMM hooks only
Trace Channel 0 (253 entries)
Current queue starts at 0x0A919000 and ends at 0x0A939000
Current entry is #128 of 128 at 0x0A92CDB4
Hook ID: VMM_DELETE (0x000001B1) Hook Type: HKTY_GT (0x0000000E)
ThreadId: 0x0000ECE5
Subhook ID/HookData: 0x0000
D0: 0x0000DD1B
D1: 0xA0801020
D2: 0x000000D3
D3: 0x000019AC0
D4: 0x00000000
Current queue starts at 0x0A919000 and ends at 0x0A939000
Current entry is #127 of 128 at 0x0A92CD84
Hook ID: VMM_DELETE (0x000001B1) Hook Type: HKTY_GT (0x0000000E)
ThreadId: 0x0000ECE5
Subhook ID/HookData: 0x0000
D0: 0x0000DD1B
D1: 0xA0801020
D2: 0x000000D6
D3: 0x00001BF3A
(0)> more (^C to quit) ? continue
D4: 0x00000000
Current queue starts at 0x0A919000 and ends at 0x0A939000
Current entry is #126 of 128 at 0x0A92CD04
Hook ID: VMM_DELETE (0x000001B1) Hook Type: HKTY_GT (0x0000000E)
ThreadId: 0x0000ECE5
Subhook ID/HookData: 0x0000
D0: 0x0000DD1B
D1: 0xA0801020
D2: 0x000000D8
D3: 0x000019AA2
D4: 0x00000000
Current queue starts at 0x0A919000 and ends at 0x0A939000
Current entry is #125 of 128 at 0x0A92CC74
Hook ID: VMM_DELETE (0x000001B1) Hook Type: HKTY_GT (0x0000000E)
ThreadId: 0x0000ECE5
Subhook ID/HookData: 0x0000
D0: 0x0000DD1B
D1: 0xA0801020
D2: 0x000000D7
D3: 0x00001A643
(0)> more (^C to quit) ? continue
D4: 0x00000000
Current queue starts at 0x0A919000 and ends at 0x0A939000
Current entry is #124 of 128 at 0x0A92CBF4
Hook ID: VMM_DELETE (0x000001B1) Hook Type: HKTY_GT (0x0000000E)
ThreadId: 0x0000ECE5
Subhook ID/HookData: 0x0000
D0: 0x0000DD1B
D1: 0xA0801020
D2: 0x000000BA
D3: 0x00001A947
D4: 0x00000000
Current queue starts at 0x0A919000 and ends at 0x0A939000
Current entry is #123 of 128 at 0x0A92CBD4
Hook ID: VMM_GETPARENT (0x000001B6) Hook Type: HKTY_GT (0x0000000E)
ThreadId: 0x0000CE27
```

```

Subhook ID/HookData: 0x0000
D0: 0x000023A4
D1: 0xA0801020
D2: 0x000000E0
D3: 0x0001D42E
(0)> more (^C to quit) ? continue
D4: 0x00000000
Current queue starts at 0x0A919000 and ends at 0x0A939000
Current entry is #122 of 128 at 0x0A92CBB4
Hook ID: VMM (0x000001B0) Hook Type: HKTY_GT (0x0000000E)
ThreadId: 0x0000CE27
Subhook ID/HookData: 0x0000
D0: 0x000023A4
D1: 0xA0801020
D2: 0x000000E0
D3: 0x0001D42E
D4: 0x00000000
Current queue starts at 0x0A919000 and ends at 0x0A939000
Current entry is #121 of 128 at 0x0A92CB94
Hook ID: VMM_DELETE (0x000001B1) Hook Type: HKTY_GT (0x0000000E)
ThreadId: 0x0000ECE5
Subhook ID/HookData: 0x0000
D0: 0x0000DD1B
D1: 0xA0801020
D2: 0x000000B9
D3: 0x000181B4
...
Hook ID: VMM_PGEXCT (0x000001B2) Hook Type: HKTY_GT (0x0000000E)
ThreadId: 0x000114ED
Subhook ID/HookData: 0x0000
D0: 0x00009D93
D1: 0xA1801000
D2: 0x0000FF99
D3: 0x00000000
(0)> more (^C to quit) ? continue
D4: 0x00000000
End of Trace

```

Net Subcommands for the KDB Kernel Debugger and kdb Command

ifnet Subcommand

The `ifnet` subcommand prints interface information, or the specified one.

Example

```

KDB(0)> ifnet display interface
SLOT 1 — IFNET INFO —(@0x00325138)—
  flags:0x08080009
    (UP|LOOPBACK)
  timer:00000 metric:00
    address: 127.0.0.1
  ifq_head:0x00000000 if_init():0x00000000 ipackets:00000190
  ifq_tail:0x00000000 if_output():0x00080E9C ierrors: 00000
  ifq_len:00000 if_ioctl():0x00080E90 opackets:00000195
  ifq_maxlen:00000 if_reset():0x00000000 oerrors: 00000
  ifq_drops:00050 if_watchdog():0x00000000
SLOT 2 — IFNET INFO —(@0x05583800)—
  flags:0x08080863
    (UP|BROADCAST|NOTRAILERS|RUNNING|CANTCHANGE)
  timer:00000 metric:00
    address: 129.183.67.8
  ifq_head:0x01A2CACC if_init():0x00000000 ipackets:00003456
  ifq_tail:0x00000000 if_output():0x01A2CAA8 ierrors: 00000

```



```

ifq_len:00000          if_ioctl():0x01A2CAC0      opackets:00000088
ifq_maxlen:00000      if_reset():0x00000000      oerrors: 00000
ifq_drops:00000       if_watchdog():0x00000000
KDB(0)>

```

tcb Subcommand

The **tcb** subcommand prints TCP blocks, or the specified one.

Example

```

KDB(0)> tcb display TCP blocks
SLOT 1 TCB ----- INPCB INFO ---(@0x05F4AB00)---
  next:0x05CD0E80    prev:0x01C033B8    head:0x01C033B8
  ppcb:0x05F9FF00    inp_socket:0x05FA4C00
  lport:      23      laddr:0x96B70114
  fport:      3972    faddr:0x81B7600D
--- SOCKET INFO ---(@05FA4C00)---
  typeCommandCommand.. 0001 (STREAM)
  optsCommandCommand.. 010C (REUSEADDR|KEEPALIVE|OOBINLINE)
  lingerCommandCommand 0000 stateCommandCommand. 0182 (ISCONNECTED|PRIV|NBIO)
  pcbCommand.. 05F4AB00 protoCommand 01C01F80 lockCommand. 05FB1680 headCommand.
  00000000 q0CommandCommand 00000000 qCommandCommand. 00000000 dqCommandCommand
  00000000 q0lenCommandCommand. 0000 qlenCommandCommand.. 0000 qlimitCommandCommand
  0000 dqlenCommandCommand. 0000 timeoCommandCommand. 0000 errorCommandCommand.
  0000 specialCommand.. 0808 pgidCommand. 00000000 oobmark. 00000000
  snd:ccCommandCommand 00000000 hiwatCommand 00004000 mbcntCommand 00000000
  mbmaxCommand 00010000 lowatCommand 00001000 mbCommandCommand 00000000 selCommand..
  00000000 eventsCommandCommand 0000 iodone.. 00000000 ioargs.. 00000000 lastpkt.
  05FA9D00 wakeone. FFFFFFFF timerCommand 00000000 timeoCommand 00000000
  flagsCommandCommand. 0000 () wakeup.. 00000000 wakearg. 00000000 lockCommand. 05FB1684
rcv:ccCommandCommand 00000000 hiwatCommand 00004000 mbcntCommand 00000000 mbmaxCommand
  00010000 lowatCommand 00000001 mbCommandCommand 00000000 selCommand.. 00000000
  eventsCommandCommand 0004 iodone.. 00000000 ioargs.. 00000000 lastpkt. 05FA4900
  wakeone. FFFFFFFF timerCommand 00000000 timeoCommand 00000000 flagsCommandCommand.
  0008 (SEL) wakeup.. 00000000 wakearg. 00000000 lockCommand. 05FB1688
(0)> more (^C to quit) ? ^C quit
KDB(0)>

```

udb Subcommand

The **udb** subcommand prints UDP blocks, or the specified one.

Example

```

KDB(0)> udb display UDP blocks
SLOT 1 UDB ----- INPCB INFO ---(@0x05F31300)---
  next:0x05D21A00    prev:0x01C07170    head:0x01C07170
  ppcb:0x00000000    inp_socket:0x05F2D200
  lport:      1595    laddr:0x00000000
  fport:      0      faddr:0x00000000
--- SOCKET INFO ---(@05F2D200)---
  typeCommandCommand.. 0002 (DGRAM)
  optsCommandCommand.. 0000 ()
  lingerCommandCommand 0000 stateCommandCommand. 0080 (PRIV)
  pcbCommand.. 05F31300 protoCommand 01C01F48 lockCommand. 05F2F900 headCommand.
  00000000 q0CommandCommand 00000000 qCommandCommand. 00000000 dqCommandCommand
  00000000 q0lenCommandCommand. 0000 qlenCommandCommand.. 0000 qlimitCommandCommand
  0000 dqlenCommandCommand. 0000 timeoCommandCommand. 0000 errorCommandCommand.
  0000 specialCommand.. 0808 pgidCommand. 00000000 oobmark. 00000000
  snd:ccCommandCommand 00000000 hiwatCommand 00010000 mbcntCommand 00000000 mbmaxCommand
  00020000 lowatCommand 00001000 mbCommandCommand 00000000 selCommand.. 00000000
  eventsCommandCommand 0000 iodone.. 00000000 ioargs.. 00000000 lastpkt. 00000000
  wakeone. FFFFFFFF timerCommand 00000000 timeoCommand 00000000 flagsCommandCommand.
  0000 () wakeup.. 00000000 wakearg. 00000000 lockCommand. 05F2F904
rcv:ccCommandCommand 00000000 hiwatCommand 00010000 mbcntCommand 00000000 mbmaxCommand
  00020000 lowatCommand 00000001 mbCommandCommand 00000000 selCommand.. 00000000
  eventsCommandCommand 0000 iodone.. 00000000 ioargs.. 00000000 lastpkt. 05D3DD00

```

```

wakeone. FFFFFFFF timerCommand 00000000 timeoCommand 0000005E flagsCommandCommand.
0000 () wakeup.. 00000000 wakearg. 00000000 lockCommand. 05F2F908
(0)> more (^C to quit) ? ^C quit
KDB(0)>

```

sock Subcommand

The **sock** subcommand prints TCP UDP sockets, or the specified one. It is possible to select only one protocol: **tcp** or **udp**.

Example

```

KDB(0)> sock tcp display TCP sockets
— TCP —(inpcb: @0x05F4AB00)—
— SOCKET INFO —(@05FA4C00)—
  typeCommandCommand.. 0001 (STREAM)
  optsCommandCommand.. 010C (REUSEADDR|KEEPALIVE|OOBINLINE)
  lingerCommandCommand 0000 stateCommandCommand. 0182 (ISCONNECTED|PRIV|NBIO)
  pcbCommand.. 05F4AB00 protoCommand 01C01F80 lockCommand. 05FB1680 headCommand.
  00000000 q0CommandCommand 00000000 qCommandCommand. 00000000 dqCommandCommand
  00000000 q0lenCommandCommand. 0000 qlenCommandCommand.. 0000 qlimitCommandCommand
  0000 dqlenCommandCommand. 0000 timeoCommandCommand. 0000 errorCommandCommand.
  0000 specialCommand.. 0808 pgidCommand. 00000000 oobmark. 00000000
snd:ccCommandCommand 00000002 hiwatCommand 00004000 mbcntCommand 00000100 mbmaxCommand
00010000 lowatCommand 00001000 mbCommandCommand 05F2D600 selCommand.. 00000000
eventsCommandCommand 0000 iodone.. 00000000 ioargs.. 00000000 lastpkt. 05F2D600
wakeone. FFFFFFFF timerCommand 00000000 timeoCommand 00000000 flagsCommandCommand.
0000 () wakeup.. 00000000 wakearg. 00000000 lockCommand. 05FB1684
rcv:ccCommandCommand 00000000 hiwatCommand 00004000 mbcntCommand 00000000 mbmaxCommand
00010000 lowatCommand 00000001 mbCommandCommand 00000000 selCommand.. 00000000
eventsCommandCommand 0005 iodone.. 00000000 ioargs.. 00000000 lastpkt. 05E1A200
wakeone. FFFFFFFF timerCommand 00000000 timeoCommand 00000000 flagsCommandCommand.
0008 (SEL) wakeup.. 00000000 wakearg. 00000000 lockCommand. 05FB1688
— TCP —(inpcb: @0x05CD0E80)—
— SOCKET INFO —(@05CABA00)—
  typeCommandCommand.. 0001 (STREAM)
(0)> more (^C to quit) ? ^C quit
KDB(0)>

```

tcpcb Subcommand

The **tcpcb** subcommand prints TCP control blocks, or the specified one.

Example

```

KDB(0)> tcpcb display TCB control blocks
— TCP —(inpcb: @0x05B17F80)—
— TCPCB —(@0x05B26C00)—
  seg_next 0x05B26C00 seg_prev 0x05B26C00 t_state 0x04 (ESTABLISHED)
  timers: TCPT_REXMT:3 TCPT_PERSIST:0 TCPT_KEEP:14400 TCPT_2MSL:0
  t_txtshift 0 t_txtcur 3 t_dupacks 0 t_maxseg 1460 t_force 0
  flags:0x0000 ()
  t_template 0x00000000 inpcb 0x00000000
  snd_cwnd: 0x00009448 snd_ssthresh:0x3FFFC000
  snd_una: 0x1EADFCA0 snd_nxt: 0x1EADFCA2 snd_up: 0x1EADFCA0
  snd_wll: 0xE3BDEEAF snd_wl2: 0x1EADFCA0 iss: 0x1EAD8401
  snd_wnd: 16060 rcv_wnd: 16060
  t_idle: 0x00000000 t_rtt: 0x00000001 t_rtseq: 0x1EADFCA0
  t_srtt: 0x00000007 t_rttvar: 0x00000003
  max_sndwnd:16060 t_iobc:0x00 t_oobflags:0x00 ()
— TCP —(inpcb: @0x05B2D000)—
— TCPCB —(@0x05B28300)—
  seg_next 0x05B28300 seg_prev 0x05B28300 t_state 0x04 (ESTABLISHED)
  timers: TCPT_REXMT:0 TCPT_PERSIST:0 TCPT_KEEP:4719 TCPT_2MSL:0
  t_txtshift 0 t_txtcur 3 t_dupacks 0 t_maxseg 1460 t_force 0
  flags:0x0000 ()

```

```

t_template 0x00000000 inpcb          0x00000000
snd_cwnd:  0x0000111C snd_ssthresh:0x3FFFC000
(0)> more (^C to quit) ? ^C quit
KDB(0)>

```

mbuf Subcommand

The **mbuf** subcommand prints TCP UDP message buffers, or the specified one. It is possible to select only one protocol: **tcp** or **udp**.

Example

```

KDB(0)> mbuf display message buffers
— TCP —(inpcb: @0x05F4AB00)—
— SND —(sock: @0x05FA4C00)—
m_commandCommandCommandCommandCommand 05E2E900 m_nextCommandCommandCommand. 00000000
m_nextpktCommandCommand. 00000000
m_lenCommandCommandCommand.. 00000004 m_dataCommandCommandCommand. 05E2E91C
m_typeCommand.. 0001 DATA
m_flagsCommandCommandCommand 0002 (M_PKTHDR)
m_pkthdr.lenCommand. 00000004 m_pkthdr.rcvif.. 00000000
-----
05E2E91C: 7566 0D0A                                uf..
— TCP —(inpcb: @0x05CD0E80)—
— TCP —(inpcb: @0x05CA6B80)—
— TCP —(inpcb: @0x05EB0A00)—
— TCP —(inpcb: @0x05D21E00)—
— TCP —(inpcb: @0x05CA6880)—
— TCP —(inpcb: @0x05DB1F00)—
— TCP —(inpcb: @0x05DB1F80)—
— TCP —(inpcb: @0x05DB1C80)—
— TCP —(inpcb: @0x05DB1D00)—
— TCP —(inpcb: @0x05DB1D80)—
— TCP —(inpcb: @0x05DB1E00)—
— TCP —(inpcb: @0x05F31580)—
— TCP —(inpcb: @0x05F31900)—
— TCP —(inpcb: @0x05F31980)—
(0)> more (^C to quit) ? ^C quit
KDB(0)>

```

VMM Subcommands for the KDB Kernel Debugger and kdb Command

Most of vmm subcommands can be used without argument, then all choices are printed. It is also possible to enter the selection as argument.

vmker Subcommand

The **vmker** subcommand prints virtual memory kernel data.

example:

```

KDB(4)> vmker display virtual memory kernel data
VMM Kernel Data:
vmm srval      (vmmsrval)   : 00000801
pgsp map srval (dmapsrval)  : 00001803
ram disk srval (ramdsrval)  : 00000000
kernel ext srval (kexsrval)   : 00002004
iplcb vaddr    (iplcbptr)   : 0045A000
hashbits      (hashbits)   : 00000010
hash shift amount (stoibits)  : 0000000B
rsvd pgsp blks (psrsvdblks) : 00000500
total page frames (nrpages)  : 0001FF58
bad page frames (badpages)   : 00000000
free page frames (numfrb)    : 000198AF
max perm frames (maxperm)    : 000195E0
num perm frames (numperm)    : 0000125A

```

```

total pgsp blks (numpsblks) : 00050000
free pgsp blks (psfreeblks) : 0004CE2C
base config seg (bconfsrval) : 0000580B
rsvd page frames (pfrsvdblks) : 00006644
fetch protect (nofetchprot): 00000000
shadow srval (ukernsrval) : 60000000
num client frames (numclient) : 00000014
max client frames (maxclient) : 000195E0
kernel srval (kernsrval) : 00000000
STOI/ITOS mask (stoimask) : 0000001F
STOI/ITOS sid mask (stoinio) : 00000000
max file pageout (maxpout) : 00000000
min file pageout (minpout) : 00000000
repage table size (rptsize) : 00010000
next free in rpt (rptfree) : 00000000
repage decay rate (rpdecay) : 0000005A
global repage cnt (sysrepage) : 00000000
swhashmask (swhashmask) : 0000FFFF
hashmask (hashmask) : 0000FFFF
cachealign (cachealign) : 00001000
overflows (overflows) : 00000000
reloads (reloads) : 0000078E
pmap_lock_addr (pmap_lock_addr): 00000000
compressed segs (numcompress): 00000000
compressed files (noflush) : 00000000
extended iplcb (iplcbxptr) : 00000000
alias hash mask (ahashmask) : 000000FF
max pgs to delete (pd_npages) : 00080000
vrl d xlate hits (vrl dhits) : 00000000
vrl d xlate misses (vrl dmisses) : 0000004C
vmm 1 swpft (...srval) : 00003006
vmm 2 swpft (...srval) : 00003807
vmm 3 swpft (...srval) : 00004008
vmm 4 swpft (...srval) : 00004809
vmm swhat (...srval) : 00002805
# of ptasegments (numptasegs) : 00000001
vmkerlock (vmkerlock) : E8000100
ame srval(s) (amesrval[0]) : 0000600C
ptaseg(s) (ptasegs[1]) : 00001002

```

rmap Subcommand

The **rmap** subcommand prints the real address range mapping table.

example:

```

KDB(2)> rmap * display real address range mappings
          SLOT          RADDR          SIZE          ALIGN WIMG <name>
vmrmap+000028 0001 0000000000000000 00458D51 00000000 0002 Kernel
vmrmap+000048 0002 000000001FF20000 00028000 00000000 0002 IPL control block
vmrmap+000068 0003 0000000000459000 00058000 00001000 0002 MST
vmrmap+000088 0004 00000000008BF000 001ABCE0 00000000 0002 RAMD
vmrmap+0000A8 0005 0000000000A6B000 00025001 00000000 0002 BCFG
vmrmap+0000E8 0007 0000000000C00000 00400000 00400000 0002 PFT
vmrmap+000108 0008 00000000004B1000 0007FD60 00001000 0002 PVT
vmrmap+000128 0009 0000000000531000 00200000 00001000 0002 PVLIST
vmrmap+000148 000A 0000000001000000 0067DDE0 00001000 0002 s/w PFT
vmrmap+000168 000B 0000000000731000 00040000 00001000 0002 s/w HAT
vmrmap+000188 000C 0000000000771000 00001000 00001000 0002 APT
vmrmap+0001A8 000D 0000000000772000 00000200 00001000 0002 AHAT
vmrmap+0001C8 000E 0000000000773000 00080000 00001000 0002 RPT
vmrmap+0001E8 000F 00000000007F3000 00020000 00001000 0002 RPHAT
vmrmap+000208 0010 0000000000813000 0000D000 00001000 0002 PDT
vmrmap+000228 0011 0000000000820000 00001000 00001000 0002 PTAR
vmrmap+000248 0012 0000000000821000 00002000 00001000 0002 PTAD
vmrmap+000268 0013 0000000000823000 00003000 00001000 0002 PTAI
vmrmap+000288 0014 0000000000826000 00001000 00001000 0002 DMAP

```

```

vmrmap+0002C8 0016 00000000FF000000 00000100 00000000 0005 SYSREG
vmrmap+0002E8 0017 00000000FF100000 00000600 00000000 0005 SYSINT
vmrmap+000308 0018 00000000FF600000 00022000 00000000 0005 NVRAM
vmrmap+000328 0019 000000001FD00000 00080000 00000000 0006 TCE
vmrmap+000348 001A 000000001FC00000 00080000 00000000 0006 TCE
vmrmap+000368 001B 00000000FF001000 00000014 00000000 0005 System Specific Reg.
vmrmap+000388 001C 00000000FF180000 00000004 00000000 0005 APR
KDB(2)> rmap 16 display real address range mappings of slot 16
RMAP entry 0016 of 001F: SYSREG
> valid
> range is in I/O space
Real address      : 00000000FF000000
Effective address : 00000000E0000000
Size              : 00000100
Alignment        : 00000000
WIMG bits        : 5
KDB(2)> rmap display page intervals utilized by the VMM
VMM RMAP, usage: rmap [*][<slot>]
Enter the RMAP index (0-001F): 20 out of range slot
Interval entry 0 of 5
.... Memory holes (1 intervals)
   0 : [01FF58,100000)
Interval entry 1 of 5
.... Fixed kernel memory (4 intervals)
   0 : [000000,0000F8)
   1 : [0000F7,00011A)
   2 : [000119,000125)
   3 : [0002E6,0002E9)
Interval entry 2 of 5
.... Released kernel memory (1 intervals)
   0 : [00011A,000124)
Interval entry 3 of 5
.... Fixed common memory (2 intervals)
   0 : [000488,000495)
   1 : [000494,000495)
Interval entry 4 of 5
.... Page replacement skips (6 intervals)
   0 : [000000,000827)
   1 : [000C00,00167E)
   2 : [01FC00,01FC80)
   3 : [01FD00,01FD80)
   4 : [01FF20,01FF48)
   5 : [01FF58,100000)
Interval entry 5 of 5
.... Debugger skips (3 intervals)
   0 : [0004B1,000731)
   1 : [000C00,001000)
   2 : [01FF58,100000)

```

pfhdata Subcommand

The **pfhdata** subcommand prints virtual memory control variables.

example:

```

KDB(2)> pfhdata display virtual memory control variables
VMM Control Variables: B69C8000 vmmidseg +69C8000
1st non-pinned page (firstnf)      : 00000000
1st free sid entry (sidfree)       : 000003F0
1st delete pending (sidxmem)       : 00000000
highest sid entry (hisid)          : 0000040C
fblru page-outs (numpout)          : 00000000
fblru remote pg-outs (numremote)   : 00000000
frames not pinned (pfavail)        : 0001E062
next lru candidate (lrupttr)       : 00000000
v_sync cursor (syncptr)            : 00000000
last pdt on i/o list (iotail)      : FFFFFFFF

```

```

num of paging spaces (npgspaces)      : 00000002
PDT last alloc from (pdtlast)         : 00000001
max pgsp PDT index  (pdtmaxpg)       : 00000001
PDT index of server (pdtserver)      : 00000000
fblru minfree       (minfree)        : 00000078
fblru maxfree       (maxfree)        : 00000080
scb serial num      (nxtscbnum)      : 00000338
comp repage cnt     (rpgcnt[RPCOMP])  : 00000000
file repage cnt     (rpgcnt[RPFILER]) : 00000000
num of comp replaces (nreplaced[RPCOMP]): 00000000
num of file replaces (nreplaced[RPFILER]): 00000000
num of comp repages (nreplaced[RPCOMP]): 00000000
num of file repages (nreplaced[RPFILER]): 00000000
minperm             (minperm)        : 00006578
min page-ahead     (minpgahead)      : 00000002
max page-ahead     (maxpgahead)      : 00000008
sysbr protect key  (kerkey)          : 00000000
non-ws page-outs   (numpermio)       : 00000000
free frame wait    (freewait)        : 00000000
device i/o wait    (devwait)         : 00000000
extend XPT wait    (extendwait)      : 00000000
buf struct wait    (bufwait)         : 00000000
inh/delete wait    (deletewait)      : 00000000
SIGDANGER level    (npswarn)         : 00002800
SIGKILL level      (npskill)         : 00000A00
next warn level    (nextwarn)        : 00002800
next kill level    (nextkill)        : 00000A00
adj warn level     (adjwarn)         : 00000008
adj kill level     (adjkill)         : 00000008
cur pdt alloc      (npdtblks)        : 00000003
max pdt alloc      (maxpdtblks)      : 00000004
num i/o sched      (numsched)        : 00000004
freewake           (freewake)        : 00000000
disk quota wait    (dqwait)          : 00000000
1st free ame entry (amefree)         : FFFFFFFF
1st del pending ame (amexmem)        : 00000000
highest ame entry   (hiame)          : 00000000
pag space free wait (pgspwait)       : 00000000
index in int array  (lruidx)         : 00000000
next memory hole    (skiplru)        : 00000000
first free apt entry (aptfree)       : 00000056
next apt entry      (aptrlu)         : 00000000
sid index of logs   (logsid)         @ B01C80CC
lru request         (lrurequested)    : 00000000
lru daemon wait anchor (lrudaemon)   : E6000758
global vmap         lock @ B01C8514 E80001C0
global ame          lock @ B01C8554 E8000200
global rpt          lock @ B01C8594 E8000240
global alloc        lock @ B01C85D4 E8000280
apt freelist        lock @ B01C8614 E80002C0

```

vmstat Subcommand

The **vmstat** subcommand prints virtual memory statistics.

example:

```

KDB(6)> vmstat display virtual memory statistics
VMM Statistics:
page faults           (pgexct)   : 0CE0A83D
page reclaims        (pgrclm)   : 00000000
lockmisses           (lockexct) : 00000000
backtracks           (backtrks) : 0025D779
pages paged in       (pageins)  : 002D264A
pages paged out      (pageouts) : 00E229D1
paging space page ins (pgspgins) : 0001F9C8
paging space page outs (pgspgouts): 0003B20E

```

```

start I/Os                (numsios) : 00B4786A
iodones                   (numiodone): 00B478F7
zero filled pages        (zerofills): 0225E1A4
executable filled pages  (exfills) : 000090C4
pages examined by clock  (scans) : 008F32DF
clock hand cycles        (cycles) : 0000008F
page steals              (pgsteals) : 004E986F
free frame waits         (freewts) : 023449E5
extend XPT waits         (extendwts): 000008C9
pending I/O waits        (pendiowts): 0022C5E3
VMM Statistics:
ping-pongs: source => alias (pings) : 00000000
ping-pongs: alias => source (pongs) : 00000000
ping-pongs: alias => alias (pangs) : 00000000
ping-pongs: alias page del (dpongs): 00000000
ping-pongs: alias page write(wpongs): 00000000
ping-pong cache flushes (cachef): 00000000
ping-pong cache invalidates (cachei): 00000000

```

vmaddr Subcommand

The **vmaddr** subcommand prints virtual memory addresses.

example:

```

KDB(1)> vmaddr display virtual memory addresses
VMM Addresses
H/W PTE : 00C00000 [real address]
H/W PVT : 004B1000 [real address]
H/W PVLIST : 00531000 [real address]
S/W HAT : A0000000 A0000000
S/W PFT : 60000000 60000000
AHAT : B0000000 vmmseg +000000
APT : B0020000 vmmseg +020000
RPHAT : B0120000 vmmseg +120000
RPT : B0140000 vmmseg +140000
PDT : B01C0000 vmmseg +1C0000
PFHDATA : B01C8000 vmmseg +1C8000
LOCKANCH : B01C8654 vmmseg +1C8654
SCBs : B01CC87C vmmseg +1CC87C
LOCKWORDS : B45CC87C vmmseg +45CC87C
AMEs : D0000000 ameseg +000000
LOCK:
PMAP : 00000000 00000000

```

pdt Subcommand

The **pdt** subcommand prints paging device table. To print all pdts, enter the subcommand without argument and select the first to be displayed.

example:

```

KDB(3)> pdt * display paging device table
      SLOT  NEXTIO  DEVICE  IOTAIL  DMSRVAL  IOCNT <name>
vmmseg+1C0000 0000 FFFFFFFF 000A0001 FFFFFFFF 00000000 00000000 paging
vmmseg+1C0040 0001 FFFFFFFF 000A000E FFFFFFFF 00000000 00000000 paging
vmmseg+1C0080 0002 FFFFFFFF 000A000F FFFFFFFF 00000000 00000000 paging
vmmseg+1C0440 0011 FFFFFFFF 000A0007 FFFFFFFF 0001B07B 00000000 filesystem
vmmseg+1C0480 0012 FFFFFFFF 000A0003 FFFFFFFF 00000000 00000000 log
vmmseg+1C04C0 0013 FFFFFFFF 000A0004 FFFFFFFF 00005085 00000000 filesystem
vmmseg+1C0500 0014 FFFFFFFF 000A0005 FFFFFFFF 0000B08B 00000000 filesystem
vmmseg+1C0540 0015 FFFFFFFF 000A0006 FFFFFFFF 0000E0AE 00000000 filesystem
vmmseg+1C0580 0016 FFFFFFFF 000A0008 FFFFFFFF 0000F14F 00000000 filesystem
vmmseg+1C05C0 0017 FFFFFFFF 0B5C7308 FFFFFFFF 00000000 00000000 remote
vmmseg+1C0600 0018 FFFFFFFF 0B5C75B4 FFFFFFFF 00000000 00000000 remote
KDB(3)> pdt 13 display paging device table slot 13
PDT address B01C04C0 entry 0013 of 01FF, type: FILESYSTEM

```

```

next pdt on i/o list (nextio) : FFFFFFFF
dev_t or strategy ptr (device) : 000A0004
last frame w/pend I/O (iotail) : FFFFFFFF
free buf_struct list (bufstr) : 0B23A0B0
total buf structs (nbufs) : 005D
available (PAGING) (avail) : 0000
JFS disk agsize (agsize) : 0400
JFS inode agsize (iagsize) : 0800
JFS log SCB index (logsidx) : 0007A
JFS fragments per page(fperpage): 1
JFS compression type (comptype): 0
JFS log2 bigalloc mult(bigexp) : 0
disk map srval (dmsrval) : 00005085
i/o's not finished (iocnt) : 00000000
logical volume lock (lock) :0B01C04E4 00000000

```

scb Subcommand

The `scb` subcommand displays VMM segment control blocks.

example:

```

KDB(2)> scb display VMM segment control block
VMM SCBs
Select the scb to display by:
  1) index
  2) sid
  3) srval
  4) search on sibits
  5) search on npsblks
  6) search on npages
  7) search on npseablks
  8) search on lock
  9) search on segment type
Enter your choice: 2 sid
Enter the sid (in hex): 00000401 value
VMM SCB Addr B69CC8C0 Index 00000001 of 00003A2F Segment ID: 00000401
WORKING STORAGE SEGMENT
parent sid (parent) : 00000000
left child sid (left) : 00000000
right child sid (right) : 00000000
extent of growing down (minvpn) : 0000ABBD
last page user region (sysbr) : FFFFFFFF
up limit (uplim) : 00007FFF
down limit (downlim) : 00008000
number of pgsp blocks (npsblks) : 00000008
number of epsa blocks (npseablks): 00000000
segment info bits (_sibits) : A004A000
default storage key (_defkey) : 2
> (_segtype)..... working segment
> (_segtype)..... segment is valid
> (_system)..... system segment
> (_chgbit)..... segment modified
> (_compseg)..... computational segment
next free list/mmap cnt (free) : 00000000
non-fblu pageout count (npopages): 0000
xmem attach count (xmencnt) : 0000
address of XPT root (vxpto) : C00C0400
pages in real memory (npages) : 0000080E
page frame at head (sidlist) : 00006E66
max assigned page number (maxvpn) : 00006AC3
lock (lock) : E80001C0
KDB(2)> scb display VMM segment control block
VMM SCBs
Select the scb to display by:
  1) index
  2) sid

```



```

3) srval
4) search on sibits
5) search on npsblks
6) search on npages
7) search on npseablks
8) search on lock
9) search on segment type
Enter your choice: 8 search on lock
Find all scbs currently locked
    sidx 00000012 locked: 00044EEF
    sidx 00000D63 locked: 000412F7
    sidx 00000FB5 locked: 00044EEF
    sidx 00001072 locked: 000280E7
    sidx 000034B4 locked: 0002EC61
5 (dec) scb locked
KDB(2)> scb 1 display VMM segment control block by index
Enter the index (in hex): 000034B4 index
VMM SCB Addr B6AAC84C Index 000034B4 of 00003A2F Segment ID: 000064B4
WORKING STORAGE SEGMENT
parent sid          (parent)   : 00000000
left child sid      (left)     : 00000000
right child sid     (right)    : 00000000
extent of growing down (minvpn) : 00010000
last page user region (sysbr)  : 00010000
up limit            (uplim)    : 0000FFFF
down limit          (downlim)  : 00010000
number of pgsp blocks (npsblks) : 0000000A
number of epsa blocks (npseablks): 00000000
segment info bits   (_sibits)  : A0002080
default storage key (_defkey)  : 2
> (_segtype)..... working segment
> (_segtype)..... segment is valid
> (_compseg)..... computational segment
> (_sparse)..... sparse segment
next free list/mmap cnt (free)  : 00000000
non-fblu pageout count (npopages): 0000
xmem attach count      (xmencnt) : 0000
address of XPT root    (vxpto)   : C0699C00
pages in real memory   (npages)   : 00000011
page frame at head     (sidlist)  : 00004C5C
max assigned page number (maxvpn)  : 000001C1
lock                   (lock)     : E80955E0

```

pft Subcommand

The **pft** subcommand displays VMM page frame table.

example:

```

KDB(5)> pft display VMM page frame
VMM PFT
Select the PFT entry to display by:
1) page frame #
2) h/w hash (sid,pno)
3) s/w hash (sid,pno)
4) search on swbits
5) search on pincount
6) search on xmencnt
7) scb list
8) io list
Enter your choice: 7 scb list
Enter the sid (in hex): 00005555 sid value
VMM PFT Entry For Page Frame 0EB87 of 0FF67
pte = B0155520, pvt = B203AE1C, pft = B3AC2950
h/w hashed sid : 00005555 pno : 00000001 key : 1
source sid : 00005555 pno : 00000001 key : 1
> in use

```

```

> on scb list
> valid (h/w)
> referenced (pft/pvt/pte): 0/0/1
> modified (pft/pvt/pte): 0/0/0
page number in scb (pagex) : 00000001
disk block number (dblock) : 00000AC6
next page on scb list (sidfwd) : 0000E682
prev page on scb list (sidbwd) : FFFFFFFF
freefwd/waitlist (freefwd): 00000000
freebwd/logage/pincnt (freebwd): 00000000
out of order I/O (nonfifo): 0000
next frame i/o list (nextio) : 00000000
storage attributes (wimg) : 2
xmem hide count (xmemcnt): 0
next page on s/w hash (next) : FFFFFFFF
List of alias entries (alist) : 0000FFFF
index in PDT (devid) : 0014
VMM PFT Entry For Page Frame 0E682 of 0FF67
pte = B01555F0, pvt = B2039A08, pft = B3AB3860
h/w hashed sid : 00005555 pno : 00000002 key : 1
source sid : 00005555 pno : 00000002 key : 1
> in use
> on scb list
> valid (h/w)
> referenced (pft/pvt/pte): 0/0/1
> modified (pft/pvt/pte): 0/0/0
page number in scb (pagex) : 00000002
disk block number (dblock) : 00000AC7
next page on scb list (sidfwd) : 0000EB7B
prev page on scb list (sidbwd) : 0000EB87
freefwd/waitlist (freefwd): 00000000
freebwd/logage/pincnt (freebwd): 00000000
out of order I/O (nonfifo): 0000
next frame i/o list (nextio) : 00000000
storage attributes (wimg) : 2
xmem hide count (xmemcnt): 0
next page on s/w hash (next) : FFFFFFFF
List of alias entries (alist) : 0000FFFF
index in PDT (devid) : 0014
VMM PFT Entry For Page Frame 0EB7B of 0FF67
pte = B0155558, pvt = B203ADEC, pft = B3AC2710
h/w hashed sid : 00005555 pno : 00000000 key : 1
source sid : 00005555 pno : 00000000 key : 1
> in use
> on scb list
> valid (h/w)
> referenced (pft/pvt/pte): 0/0/1
> modified (pft/pvt/pte): 0/0/0
page number in scb (pagex) : 00000000
disk block number (dblock) : 00000AC5
next page on scb list (sidfwd) : FFFFFFFF
prev page on scb list (sidbwd) : 0000E682
freefwd/waitlist (freefwd): 00000000
freebwd/logage/pincnt (freebwd): 00000000
out of order I/O (nonfifo): 0000
next frame i/o list (nextio) : 00000000
storage attributes (wimg) : 2
xmem hide count (xmemcnt): 0
next page on s/w hash (next) : FFFFFFFF
List of alias entries (alist) : 0000FFFF
index in PDT (devid) : 0014
Pages on SCB list
npages..... 00000003
on sidlist..... 00000003
pageout_pagein.. 00000000
free..... 00000000
KDB(0)> pft 8 io list

```

```

Enter the page frame number (in hex): 00002749 first page frame
VMM PFT Entry For Page Frame 02749 of 0FF67
pte = B00C9280, pvt = B2009D24, pft = B3875DB0
h/w hashed sid : 0080324A pno : 00000000 key : 1
source sid : 0000324A pno : 00000000 key : 1
> page out
> on scb list
> ok to write to home
> valid (h/w)
> referenced (pft/pvt/pte): 0/1/0
> modified (pft/pvt/pte): 1/1/0
page number in scb (pagex) : 00000000
disk block number (dblock) : 0000420D
next page on scb list (sidfwd) : 0000EE94
prev page on scb list (sidbwd) : 00002E11
freefwd/waitlist (freefwd): E6096C00
freebwd/logage/pincnt (freebwd): 00000000
out of order I/O (nonfifo): 0001
index in PDT (devid) : 0033
next frame i/o list (nextio) : 000043EB
storage attributes (wimg) : 2
xmem hide count (xmemcnt): 0
next page on s/w hash (next) : FFFFFFFF
List of alias entries (alist) : 0000FFFF
VMM PFT Entry For Page Frame 043EB of 0FF67 next frame i/o list
pte = B01580C0, pvt = B2010FAC, pft = B38CBC10
h/w hashed sid : 008055FC pno : 000003FF key : 1
source sid : 000055FC pno : 000003FF key : 1
> page out
> on scb list
> ok to write to home
> valid (h/w)
> referenced (pft/pvt/pte): 0/1/0
> modified (pft/pvt/pte): 1/1/0
page number in scb (pagex) : 000003FF
disk block number (dblock) : 00044D47
next page on scb list (sidfwd) : 00005364
prev page on scb list (sidbwd) : 000043EB
freefwd/waitlist (freefwd): 00000000
freebwd/logage/pincnt (freebwd): 00000000
out of order I/O (nonfifo): 0001
index in PDT (devid) : 0031
next frame i/o list (nextio) : 00004405
storage attributes (wimg) : 2
xmem hide count (xmemcnt): 0
next page on s/w hash (next) : 00002789
List of alias entries (alist) : 0000FFFF
...
VMM PFT Entry For Page Frame 02E11 of 0FF67
pte = B00C90C0, pvt = B200B844, pft = B388A330
h/w hashed sid : 0080324A pno : 00000009 key : 1
source sid : 0000324A pno : 00000009 key : 1
> page out
> on scb list
> ok to write to home
> valid (h/w)
> referenced (pft/pvt/pte): 0/1/0
> modified (pft/pvt/pte): 1/1/0
page number in scb (pagex) : 00000009
disk block number (dblock) : 000042C0
next page on scb list (sidfwd) : 00002749
prev page on scb list (sidbwd) : 00002FCB
freefwd/waitlist (freefwd): 00000000
freebwd/logage/pincnt (freebwd): 00000000
out of order I/O (nonfifo): 0001
index in PDT (devid) : 0033
next frame i/o list (nextio) : 00002749

```

```

storage attributes (wimg) : 2
xmem hide count (xmemcnt): 0
next page on s/w hash (next) : FFFFFFFF
List of alias entries (alist) : 0000FFFF
Pages on iolist..... 00000091

```

pte Subcommand

The **pte** subcommand displays VMM page table entries.

example:

```

KDB(1)> pte display VMM page table entry
VMM PTE
Select the PTE to display by:
 1) index
 2) sid,pno
 3) page frame
 4) PTE group
Enter your choice: 2          sid,pno
Enter the sid (in hex): 802 sid value
Enter the pno (in hex): 0   pno value
  PTEX v SID  h avpi RPN  r c wimg pp
004010 1 000802 0  00 007CD 1 1 0002 00
KDB(1)> pte 4 display VMM page table group
Enter the sid (in hex): 802 sid value
Enter the pno (in hex): 0   pno value
  PTEX v SID  h avpi RPN  r c wimg pp
004010 1 000802 0  00 007CD 1 1 0002 00
004011 1 000803 0  00 090FF 0 0 0002 03
004012 0 000000 0  00 00000 0 0 0000 00
004013 0 000000 0  00 00000 0 0 0000 00
004014 0 000000 0  00 00000 0 0 0000 00
004015 0 000000 0  00 00000 0 0 0000 00
004016 0 000000 0  00 00000 0 0 0000 00
004017 0 000000 0  00 00000 0 0 0000 00
  PTEX v SID  h avpi RPN  r c wimg pp
03BFEB 1 00729E 0  01 0DC55 0 0 0002 01
03BFEB 1 007659 0  00 07BC6 1 0 0002 02
03BFEB 0 000000 0  00 00000 0 0 0000 00
03BFEB 0 000000 0  00 00000 0 0 0000 00
03BFEB 0 000000 0  00 00000 0 0 0000 00
03BFEB 0 000000 0  00 00000 0 0 0000 00
03BFEB 0 000000 0  00 00000 0 0 0000 00
03BFEB 0 000000 0  00 00000 0 0 0000 00
03BFEB 0 000000 0  00 00000 0 0 0000 00

```

pta Subcommand

The **pta** subcommand displays VMM PTA segment.

example:

```

KDB(3)> pta ? display usage
VMM PTA segment @ C0000000
Usage: pta
      pta -r[oot] [sid] to print XPT root
      pta -d[blk] [sid] to print XPT direct blocks
      pta -a[pm]  [idx] to print Area Page Map
      pta -v[map] [idx] to print map blocks
      pta -x[pt]  xpt  to print XPT fields
KDB(3)> pta display PTA information
VMM PTA segment @ C0000000
pta_root..... @ C0000000 pta_hiapm..... : 00000200
pta_vmmapfree... : 00010FCB pta_usecount... : 0004D000
pta_anchor[0].. : 00000107 pta_anchor[1].. : 00000000
pta_anchor[2].. : 00000102 pta_anchor[3].. : 00000000
pta_anchor[4].. : 00000000 pta_anchor[5].. : 00000000

```

```

pta_freecnt.... : 0000000A pta_freetail... : 000001FF
pta_apm(1rst).. @ C0000600 pta_xptdbl... @ C0080000
KDB(1)> pta -a 2 display area page map for 1K bucket
VMM PTA segment @ C0000000
INDEX XPT1K
pta_apm @ C0000810 pmap... : D0000000 fwd.... : 00F7 bwd.... : 0000
pta_apm @ C00007B8 pmap... : B0000000 fwd.... : 00EE bwd.... : 0102
pta_apm @ C0000770 pmap... : E0000000 fwd.... : 00FA bwd.... : 00F7
pta_apm @ C00007D0 pmap... : 30000000 fwd.... : 0112 bwd.... : 00EE
pta_apm @ C0000890 pmap... : B0000000 fwd.... : 010A bwd.... : 00FA
pta_apm @ C0000850 pmap... : B0000000 fwd.... : 0111 bwd.... : 0112
pta_apm @ C0000888 pmap... : 50000000 fwd.... : 00F5 bwd.... : 010A
pta_apm @ C00007A8 pmap... : A0000000 fwd.... : 010E bwd.... : 0111
pta_apm @ C0000870 pmap... : 10000000 fwd.... : 00F6 bwd.... : 00F5
pta_apm @ C00007B0 pmap... : D0000000 fwd.... : 010C bwd.... : 010E
pta_apm @ C0000860 pmap... : 30000000 fwd.... : 0114 bwd.... : 00F6
pta_apm @ C00008A0 pmap... : 10000000 fwd.... : 0108 bwd.... : 010C
pta_apm @ C0000840 pmap... : E0000000 fwd.... : 010D bwd.... : 0114
pta_apm @ C0000868 pmap... : D0000000 fwd.... : 0106 bwd.... : 0108
pta_apm @ C0000830 pmap... : 50000000 fwd.... : 0000 bwd.... : 010D

```

ste Subcommand

The **ste** subcommand displays segment table entry for 64-bit process.

example:

```

KDB(0)> ste display segment table
Segment Table (STAB)
Select the STAB entry to display by:
  1) esid
  2) sid
  3) dump hash class (input=esid)
  4) dump entire stab
Enter your choice: 4 display entire stab
000000002FF9D000: ESID 0000000080000000 VSID 0000000000024292 V Ks Kp
000000002FF9D010: ESID 0000000000000000 VSID 0000000000000000 V Ks Kp
000000002FF9D020: ESID 0000000000000000 VSID 0000000000000000
000000002FF9D030: ESID 0000000000000000 VSID 0000000000000000
000000002FF9D040: ESID 0000000000000000 VSID 0000000000000000
...
(0)> f stack frame
thread+002A98 STACK:
[00031960]e_block_thread+000224 ()
[00041738]nsleep+000124 (??, ??)
[01CFF0F4]nsleep64 +000058 (0FFFFFFF, F0000001, 00000001, 10003730,
 1FFFFFF0, 1FFFFFF8)
[000038B4].sys_call+000000 ()
[80000010000867C]080000010000867C (??, ??, ??, ??)
[80000010001137C]nsleep+000094 (??, ??)
[800000100058204]sleep+000030 (??)
[100000478]main+0000CC (0000000100000001, 00000000200FEB78)
[10000023C]__start+000044 ()
(0)> ste display segment table
Segment Table (STAB)
Select the STAB entry to display by:
  1) esid
  2) sid
  3) dump hash class (input=esid)
  4) dump entire stab
Enter your choice: 3 hash class
Hash Class to dump (in hex) [esid ok here]: 08000010 input=esid
PRIMARY HASH GROUP
000000002FF9D800: ESID 0000000000000010 VSID 000000000002BC1 V Ks Kp
000000002FF9D810: ESID 0000000080000010 VSID 0000000000014AEA V Ks Kp
000000002FF9D820: ESID 0000000000000000 VSID 0000000000000000
000000002FF9D830: ESID 0000000000000000 VSID 0000000000000000

```

```

000000002FF9D840: ESID 0000000000000000 VSID 0000000000000000
000000002FF9D850: ESID 0000000000000000 VSID 0000000000000000
000000002FF9D860: ESID 0000000000000000 VSID 0000000000000000
000000002FF9D870: ESID 0000000000000000 VSID 0000000000000000
SECONDARY HASH GROUP
000000002FF9D780: ESID 0000000000000000 VSID 0000000000000000
000000002FF9D790: ESID 0000000000000000 VSID 0000000000000000
000000002FF9D7A0: ESID 0000000000000000 VSID 0000000000000000
000000002FF9D7B0: ESID 0000000000000000 VSID 0000000000000000
000000002FF9D7C0: ESID 0000000000000000 VSID 0000000000000000
000000002FF9D7D0: ESID 0000000000000000 VSID 0000000000000000
000000002FF9D7E0: ESID 0000000000000000 VSID 0000000000000000
000000002FF9D7F0: ESID 0000000000000000 VSID 0000000000000000
000000002FF9DF00: ESID 0000000000000000 VSID 0000000000000000
(0)> ste 1 display esid entry in segment table
Enter the esid (in hex): 0FFFFFFF
000000002FF9DF80: ESID 00000000FFFFFF VSID 0000000000325F9 V Ks Kp

```

sr64 Subcommand

The **sr64** subcommand displays segment registers for 64-bit process.

example:

```

KDB(0)> sr64 ? display help
Usage: sr64 [-p pid] [esid] [size]
KDB(0)> sr64 display all segment registers
SR00000000: 60000000 SR00000002: 60002B45 SR0000000D: 6000614C
SR00000010: 6000520A SR00000011: 6000636C
SR8001000A: 60003B47
SR80020014: 6000B356
SR8FFFFFFF: 60000340
SR90000000: 60001142
SR9FFFFFFF: 60004148
SRFFFFFFF: 6000B336
KDB(0)> sr64 11 display up to 16 SRs from 10
Segment registers for address space of Pid: 000048CA
SR00000010: 6000E339 SR00000011: 6000B855
KDB(0)> sr64 0 100 display up to 256 SRs from 0
Segment registers for address space of Pid: 000048CA
SR00000000: 60000000 SR00000002: 60002B45 SR0000000D: 6000614C
SR00000010: 6000520A SR00000011: 6000636C

```

segst64 Subcommand

The **segst64** subcommand displays segment state for 64-bit process.

example:

```

KDB(0)> segst64 display
snode base last nvalid sfwd sbwd
00000000 00000003 FFFFFFFE 00000010 00000001 FFFFFFFF
ESID segstate segflag num_segs fno/shmp/srval/nsegs
SR00000003>[ 0] SEG_AVAIL 00000000 0000000A
SR0000000D>[ 1] SEG_OTHER 00000001 00000001
SR0000000E>[ 2] SEG_AVAIL 00000000 00000001
SR0000000F>[ 3] SEG_OTHER 00000001 00000001
SR00000010>[ 4] SEG_TEXT 00000001 00000001
SR00000011>[ 5] SEG_WORKING 00000001 00000000
SR00000012>[ 6] SEG_AVAIL 00000000 8000FFF8
SR8001000A>[ 7] SEG_WORKING 00000001 00000000
SR8001000B>[ 8] SEG_AVAIL 00000000 00010009
SR80020014>[ 9] SEG_WORKING 00000001 00000000
SR80020015>[10] SEG_AVAIL 00000000 0FFDFFEA
SR8FFFFFFF>[11] SEG_WORKING 00000001 00000000
SR90000000>[12] SEG_TEXT 00000001 00000001
SR90000001>[13] SEG_AVAIL 00000000 0FFFFFFE

```

```

SR9FFFFFFF>[14]      SEG_TEXT 00000001 00000001
SRA0000000>[15]      SEG_AVAIL 00000000 5FFFFFFF
snode   base   last   nvalid  sfwd   sbwd
00000001 FFFFFFFF FFFFFFFF 00000001 FFFFFFFF 00000000
ESID      segstate segflag num_segs fno/shmp/srval/nsegs
SRFFFFFFF>[ 0]      SEG_WORKING 00000001 00000000

```

apt Subcommand

The **apt** subcommand displays alias page table.

example:

```

KDB(4)> apt display alias page table entry
VMM APT
Select the APT to display by:
  1) index
  2) sid,pno
  3) page frame
Enter your choice: 1      index
Enter the index (in hex): 0 value
VMM APT Entry 00000000 of 0000FF67
> valid
> pinned
segment identifier (sid) : 00001004
page number (pno) : 0000
page frame (nfr) : FF000
protection key (key) : 0
storage control attr (wimg) : 5
next on hash (next) : FFFF
next on alias list (anext): 0000
next on free list (free) : FFFF
KDB(4)> apt 2 display alias page table entry
Enter the sid (in hex): 1004 sid value
Enter the pno (in hex): 100 pno value
VMM APT Entry 00000001 of 0000FF67
> valid
> pinned
segment identifier (sid) : 00001004
page number (pno) : 0100
page frame (nfr) : FF100
protection key (key) : 0
storage control attr (wimg) : 5
next on hash (next) : 0000
next on alias list (anext): 0000
next on free list (free) : FFFF

```

vmwait Subcommand

The **vmwait** subcommand displays VMM wait status.

example:

```

KDB(6)> th -w WPGIN display threads waiting for VMM
          SLOT NAME  STATE  TID PRI  CPUID CPU  FLAGS  WCHAN
thread+000780  10 lrud  SLEEP 00A15 010  000 00001004 vmmldseg+69C84D0
thread+0012C0  25 dtlogin SLEEP 01961 03C  000 00000000 vmmldseg+69C8670
thread+001500  28 cnsview SLEEP 01C71 03C  000 00000004 vmmldseg+69C8670
thread+00B1C0  237 jfsz  SLEEP 0EDCD 032  000 00001000 vm_zqevent+0000000
thread+00C240  259 jfsc  SLEEP 10303 01E  000 00001000 _$STATIC+000110
thread+00E940  311 rm  SLEEP 137C3 03C  000 00000000 vmmldseg+69C8670
thread+012300  388 touch  SLEEP 1843B 03C  000 00000000 vmmldseg+69C8670
thread+014700  436 rm  SLEEP 1B453 03C  000 00000000 vmmldseg+69C8670
thread+0165C0  477 rm  SLEEP 1DD8D 03C  000 00000000 vmmldseg+69C8670
thread+0177C0  501 cres  SLEEP 1F529 03C  000 00000000 vmmldseg+69C8670
thread+01C980  610 lslv  SLEEP 262AF 028  000 00000000 vmmldseg+69C8670
thread+01D7C0  629 touch  SLEEP 27555 03C  000 00000000 vmmldseg+69C8670

```

```

thread+021840 715 vmmmp9 SLEEP 2CBC7 03C 000 00400000 vmmmdseg+69C8670
thread+023640 755 cres1 SLEEP 2F3DF 03C 000 00000000 vmmmdseg+69C8670
thread+027540 839 xlC SLEEP 34779 03C 000 00000000 vmmmdseg+69C8670
thread+032B80 1082 rm SLEEP 43AAB 03C 000 00000000 vmmmdseg+69C8670
thread+033900 1100 rm SLEEP 44CD9 03C 000 00000000 vmmmdseg+69C8670
thread+038D00 1212 ksh SLEEP 4BC45 029 000 00000000 vmmmdseg+69C8670
thread+03FA80 1358 cres SLEEP 54EDD 03C 000 00000000 vmmmdseg+69C8670
thread+049140 1559 touch SLEEP 617F7 03C 000 00000000 vmmmdseg+69C8670
thread+04A880 1590 rm SLEEP 6365D 03C 000 00000000 vmmmdseg+69C8670
thread+053AC0 1785 rm SLEEP 6F9A5 03C 000 00000000 vmmmdseg+69C8670
thread+05BA40 1955 rm SLEEP 7A3BB 03C 000 00000000 vmmmdseg+69C8670
thread+05FC40 2043 cres SLEEP 7FBB5 03C 000 00000000 vmmmdseg+69C8670
thread+065DC0 2173 touch SLEEP 87D35 03C 000 00000000 vmmmdseg+69C8670
thread+0951C0 3181 ksh SLEEP C6DE9 03C 000 00000000 vmmmdseg+69C8670
thread+0AD040 3691 renamer SLEEP E6B93 03C 000 00000000 vmmmdseg+69C8670
thread+0AD7C0 3701 renamer SLEEP E751F 03C 000 00000000 vmmmdseg+69C8670
thread+0B8E00 3944 ksh SLEEP F6839 03C 000 00000000 vmmmdseg+69C8670
thread+0C1B00 4132 touch SLEEP 10243D 03C 000 00000000 vmmmdseg+69C8670
thread+0C2E80 4158 renamer SLEEP 103EA9 03C 000 00000000 vmmmdseg+69C8670
thread+0CF480 4422 renamer SLEEP 1146F1 03C 000 00000000 vmmmdseg+69C8670
thread+0D0F80 4458 link_fil SLEEP 116A39 03C 000 00000000 vmmmdseg+69C9C74
thread+0DC140 4695 sync SLEEP 1257BB 03C 000 00000000 vmmmdseg+69C8670
thread+0DD280 4718 touch SLEEP 126E57 03C 000 00000000 vmmmdseg+69C8670
thread+0E5A40 4899 renamer SLEEP 132315 03C 000 00000000 vmmmdseg+69C8670
thread+0EE140 5079 renamer SLEEP 13D7C3 03C 000 00000000 vmmmdseg+69C8670
thread+0F03C0 5125 renamer SLEEP 1405B7 03C 000 00000000 vmmmdseg+69C8670
thread+0FC540 5383 renamer SLEEP 15072F 03C 000 00000000 vmmmdseg+69C8670
thread+101AC0 5497 renamer SLEEP 157909 03C 000 00000000 vmmmdseg+69C8670
thread+10D280 5742 rm SLEEP 166E37 03C 000 00000000 vmmmdseg+69C8670
KDB(6)> sw 4458 switch to thread slot 4458
Switch to thread: <thread+0D0F80>
KDB(6)> f display stack frame
thread+0D0F80 STACK:
[00017380].bactt+000000 (0000EA07, C00C2A00 [??])
[000524F4]vm_gettlock+000020 (??, ??)
[001C0D28]iwrite+0001E4 (??)
[001C3860]finicom+0000B4 (??, ??)
[001C3BC0]comlist+0001CC (??, ??)
[001C3C8C]_commit+000030 (00000000, 00000002, 0A1A06C0, 0A1ACFE8,
2FF3B400, E88C7C80, 34EF6655, 2FF3AE20)
[0020BD60]jfs_link+0000C4 (??, ??, ??, ??)
[001CED6C]vnop_link+00002C (??, ??, ??, ??)
[001D5F7C]link+0000270 (??, ??)
[000037D8].sys_call+000000 ()
[10000270]main+000098 (0000000C, 2FF229A4)
[10000174]__start+00004C ()
KDB(6)> vmwait vmmmdseg+69C9C74 display waiting channel
VMM Wait Info
Waiting on transaction block number 00000057
KDB(6)> tblk 87 display transaction block
@tblk[87] vmmmdseg +69C9C3C
logtid.... 002C77CF next..... 00000064 tid..... 00000057 flag..... 00000000
cpn..... 00000000 ceor..... 00000000 cxor..... 00000000 csn..... 00000000
waitsid... 00000000 waitline.. 00000000 locker.... 00000000 lsidx..... 00000AB3
logage.... 00B71704 gcwait.... FFFFFFFF waitors... E60D0F80 cqnext.... 00000000

```

ames Subcommand

The **ames** subcommand prints the specified process address map.

example:

```

KDB(4)> ames display current process address map
VMM AMEs
Select the ame to display by:
1) current process
2) specified process

```



```

Enter your choice: 1 current process
VMM address map, address BADCD23C
previous entry      (vme_prev)      : BADCC9FC
next entry          (vme_next)      : BADCC9FC
minimum offset     (min_offset)     : 30000000
maximum offset     (max_offset)     : D0000000
number of entries  (nentries)       : 00000001
size               (size)           : 00001000
reference count    (ref_count)      : 00000001
hint              (hint)           : BADCC9FC
first free hint    (first_free)     : BADCC9FC
entries pageable   (entries_pageable): 00000000
VMM map entry, address BADCC9FC
> copy-on-write
> needs-copy
previous entry      (vme_prev)      : BADCD23C
next entry          (vme_next)      : BADCD23C
start address       (vme_start)     : 60000000
end address         (vme_end)       : 60001000
object (vnode ptr) (object)        : 09D7EB88
page num in object (obj_pno)       : 00000000
cur protection      (protection)    : 00000003
max protection      (max_protection): 00000007
inheritance         (inheritance)   : 00000001
wired_count        (wired_count)   : 00000000
source sid          (source_sid)    : 0000272A
mapping sid         (mapping_sid)   : 000040B4
paging sid         (paging_sid)     : 000029CE
original page num  (orig_obj_pno)  : 00000000
xmem attach count  (xmattach_count): 00000000
KDB(4)> scb 2 display mapping sid
Enter the sid (in hex): 000040B4 sid value
VMM SCB Addr B6A1384C Index 000010B4 of 00003A2F Segment ID: 000040B4
MAPPING SEGMENT
ame start address  (start): 60000000
ame hint           (ame) : BADCC9FC
segment info bits  (_sibits) : 10000000
default storage key (_defkey) : 0
> (_segtype)..... mapping segment
> (_segtype)..... segment is valid
next free list/mmap cnt (free) : 00000001
non-fblu pageout count (npopages): 0000
xmem attach count      (xmencnt) : 0000
address of XPT root    (vxpto)   : 00000000
pages in real memory   (npages)  : 00000000
page frame at head    (sidlist)  : FFFFFFFF
max assigned page number (maxvpn) : FFFFFFFF
lock                  (lock)     : E8038520

```

zproc Subcommand

The **zproc** subcommand prints the VMM zeroing kproc.

example:

```

KDB(1)> zproc display VMM zeroing kproc
VMM zkproc pid = 63CA tid = 63FB
Current queue info
Queue resides at 0x0009E3E8 with 10 elements
Requests 16800 processed 16800 failed 0
Elements
  sid      pno      npg      pno      npg
0 - 007FFFFF FFFFFFFF 00000000 FFFFFFFF 00000000
1 - 007FFFFF FFFFFFFF 00000000 FFFFFFFF 00000000
2 - 007FFFFF FFFFFFFF 00000000 FFFFFFFF 00000000
3 - 007FFFFF FFFFFFFF 00000000 FFFFFFFF 00000000
4 - 007FFFFF FFFFFFFF 00000000 FFFFFFFF 00000000

```

```

5 - 007FFFFF FFFFFFFF 00000000 FFFFFFFF 00000000
6 - 007FFFFF FFFFFFFF 00000000 FFFFFFFF 00000000
7 - 007FFFFF FFFFFFFF 00000000 FFFFFFFF 00000000
8 - 007FFFFF FFFFFFFF 00000000 FFFFFFFF 00000000
9 - 007FFFFF FFFFFFFF 00000000 FFFFFFFF 00000000

```

vmlog Subcommand

The **vmlog** subcommand prints the current VMM error log entry.

example:

```

KDB(0)> vmlog display VMM error log entry
Most recent VMM errorlog entry
Error id           = DSI_PROC
Exception DSISR/ISISR = 40000000
Exception srval    = 007FFFFF
Exception virt addr = FFFFFFFF
Exception value    = 0000000E
KDB(0)> dr iar display current instruction
iar : 01913DF0
01913DF0    lwz    r0,0(r3)           r0=00001030,0(r3)=FFFFFFF
KDB(0)>

```

vrld Subcommand

The **vrld** subcommand prints the VMM reload xlate table. This information is only used on SMP POWER PC machine, to prevent VMM reload dead-lock.

example:

```

KDB(0)> vrld
freepno: 0A, initobj: 0008DAA8, *initobj: FFFFFFFF
[00] sid: 00000000, anch: 00
    {00} spno:00000000, epno:00000097, nfr:00000000, next:01
    {01} spno:00000098, epno:000000AB, nfr:00000098, next:02
    {02} spno:FFFFFFF, epno:000001F6, nfr:000001DD, next:03
    {03} spno:000001F7, epno:000001FA, nfr:000001F7, next:04
    {04} spno:0000038C, epno:000003E3, nfr:00000323, next:FF
[01] sid: 00000041, anch: 06
    {06} spno:00003400, epno:0000341F, nfr:000006EF, next:05
    {05} spno:00003800, epno:00003AFE, nfr:000003F0, next:08
    {08} spno:00006800, epno:00006800, nfr:0000037C, next:07
    {07} spno:00006820, epno:00006820, nfr:0000037B, next:09
    {09} spno:000069C0, epno:000069CC, nfr:0000072F, next:FF
[02] sid: FFFFFFFF, anch: FF
[03] sid: FFFFFFFF, anch: FF
KDB(0)>

```

ipc Subcommand

The **ipc** subcommand reports interprocess communication facility information.

example:

```

KDB(0)> ipc
IPC info
Select the display:
  1) Message Queues
  2) Shared Memory
  3) Semaphores
Enter your choice: 1
  1) all msqid_ds
  2) select one msqid_ds
  3) struct msg
Enter your choice: 1
Message Queue id 00000000 @ 019E6988

```

```

uid..... 00000000 gid..... 00000009
cuid..... 00000000 cgid..... 00000009
mode..... 000083B0 seq..... 0000
key..... 4107001C msg_first.... 00000000
msg_last.... 00000000 msg_cbytes... 00000000
msg_qnum.... 00000000 msg_qbytes... 0000FFFF
msg_lspid.... 00000000 msg_lrpid.... 00000000
msg_stime.... 00000000 msg_rtime.... 00000000
msg_ctime.... 3250C406 msg_rwait.... 0000561D
msg_wwait.... FFFFFFFF msg_reqevents. 0000
Message Queue id 00000001 @ 019E69D8
uid..... 00000000 gid..... 00000000
cuid..... 00000000 cgid..... 00000000
mode..... 000083B6 seq..... 0000
key..... 77020916 msg_first.... 00000000
msg_last.... 00000000 msg_cbytes... 00000000
msg_qnum.... 00000000 msg_qbytes... 0000FFFF
msg_lspid.... 00000000 msg_lrpid.... 00000000
msg_stime.... 00000000 msg_rtime.... 00000000
msg_ctime.... 3250C40B msg_rwait.... 00006935
msg_wwait.... FFFFFFFF msg_reqevents. 0000

```

lockanch Subcommand

The **lka** subcommand prints VMM lock anchor. The **tblk** subcommand prints the specified transaction block.

example:

```

KDB(4)> lka display VMM lock anchor
VMM LOCKANCH vmmdseg +69C8654
nexttid..... : 003AB65A
freetid..... : 0000009A
maxtid..... : 000000B8
lwptr..... : BEDCD000
freelock..... : 0000027B
morelocks..... : BEDD4000
syncwait..... : 00000000
tblkwait..... : 00000000
freewait..... : 00000000
 @tblk[1] vmmdseg +69C86BC
logtid... 003AB611 next..... 000002CF tid..... 00000001 flag..... 00000000
cpn..... 00000000 ceor..... 00000000 cxor..... 00000000 csn..... 00000000
waitsid... 00006A78 waitline.. 00000009 locker... 00000015 lsidx.... 0000096C
logage... 00B84FEC gcwait.... FFFFFFFF waitors... 00000000 cqnext.... 00000000
 @tblk[2] vmmdseg +69C86FC
logtid... 003AB61A next..... 00000000 tid..... 00000002 flag..... 00000000
cpn..... 00000000 ceor..... 00000000 cxor..... 00000000 csn..... 00000000
waitsid... 00000000 waitline.. 00000000 locker... 00000000 lsidx.... 0000096C
logage... 00B861B8 gcwait.... FFFFFFFF waitors... 00000000 cqnext.... 00000000
 @tblk[3] vmmdseg +69C873C tblk[3].cqnext vmmdseg +69C8D3C
logtid... 003AB625 next..... 0000010D tid..... 00000003 flag..... 00000007
cpn..... 00000B8B ceor..... 00000198 cxor..... 37A17C95 csn..... 00000342
waitsid... 00000000 waitline.. 00000000 locker... 00000000 lsidx.... 0000096C
logage... 00B2AFC8 gcwait.... 00031825 waitors... E6012300 cqnext.... B69C8D3C
flag..... QUEUE READY COMMIT
 @tblk[4] vmmdseg +69C877C
logtid... 003AB649 next..... 00000301 tid..... 00000004 flag..... 00000000
cpn..... 00000000 ceor..... 00000000 cxor..... 00000000 csn..... 00000000
waitsid... 00000000 waitline.. 00000000 locker... 00000000 lsidx.... 0000096C
logage... 00B35FB8 gcwait.... FFFFFFFF waitors... 00000000 cqnext.... 00000000
 @tblk[5] vmmdseg +69C87BC
logtid... 003AB418 next..... 00000000 tid..... 00000005 flag..... 00000000
cpn..... 00000000 ceor..... 00000000 cxor..... 00000000 csn..... 00000000
waitsid... 00007E7D waitline.. 00000014 locker... 0000002D lsidx.... 0000096C
logage... 00B46244 gcwait.... FFFFFFFF waitors... 00000000 cqnext.... 00000000
 @tblk[6] vmmdseg +69C87FC

```

```

logtid.... 003AB5AD next..... 0000003D tid..... 00000006 flag..... 00000000
cpn..... 00000000 ceor..... 00000000 cxor..... 00000000 csn..... 00000000
waitsid... 00007E7D waitline.. 0000001C locker.... 00000046 lsidx.... 0000096C
logage.... 00B2BF9C gcwait.... FFFFFFFF waitors... E603CE40 cqnext.... 00000000
@tblk[7] vmmdseg +69C883C
logtid.... 003AB1EC next..... 000001A3 tid..... 00000007 flag..... 00000000
cpn..... 00000000 ceor..... 00000000 cxor..... 00000000 csn..... 00000000
waitsid... 00000000 waitline.. 00000000 locker.... 00000000 lsidx.... 0000096C
logage.... 00B11F74 gcwait.... FFFFFFFF waitors... 00000000 cqnext.... 00000000
(4)> more (^C to quit) ?

```

lockhash Subcommand

The `lkh` subcommand prints VMM lock hash list.

example:

```

KDB(4)> lkh display VMM lock hash list
          BUCKET HEAD      COUNT
vmmdseg +69CC67C    1 00000144    3
vmmdseg +69CC680    2 0000019D    3
vmmdseg +69CC684    3 0000028E    2
vmmdseg +69CC688    4 00000179    2
vmmdseg +69CC68C    5 00000275    4
vmmdseg +69CC690    6 00000249    1
vmmdseg +69CC694    7 000000D4    2
vmmdseg +69CC698    8 00000100    2
vmmdseg +69CC69C    9 0000005E    2
vmmdseg +69CC6A0   10 00000171    2
vmmdseg +69CC6A4   11 00000245    2
vmmdseg +69CC6AC   13 00000136    2
vmmdseg +69CC6B4   15 000002F1    3
vmmdseg +69CC6B8   16 00000048    1
vmmdseg +69CC6BC   17 00000344    2
vmmdseg +69CC6C4   19 000001E9    2
vmmdseg +69CC6C8   20 0000021C    4
vmmdseg +69CC6D0   22 00000239    1
vmmdseg +69CC6D4   23 00000008    2
vmmdseg +69CC6D8   24 00000304    2
vmmdseg +69CC6DC   25 00000228    6
vmmdseg +69CC6E8   28 0000008A    2
vmmdseg +69CC6EC   29 000002F8    3
vmmdseg +69CC6F0   30 0000005F    1
vmmdseg +69CC6F4   31 000001FB    1
vmmdseg +69CC6FC   33 00000107    1
vmmdseg +69CC700   34 0000032A    2
vmmdseg +69CC704   35 00000326    1
vmmdseg +69CC708   36 0000006B    2
vmmdseg +69CC70C   37 000002CF    1
vmmdseg +69CC710   38 00000034    1
vmmdseg +69CC718   40 000000CC    2
vmmdseg +69CC71C   41 000001A4    1
vmmdseg +69CC728   44 000000C5    2
vmmdseg +69CC72C   45 000001C8    1
vmmdseg +69CC730   46 00000075    3
vmmdseg +69CC734   47 00000347    2
vmmdseg +69CC738   48 000001C0    2
vmmdseg +69CC73C   49 00000321    4
vmmdseg +69CC740   50 0000033C    3
vmmdseg +69CC744   51 00000201    3
vmmdseg +69CC750   54 000002CE    3
vmmdseg +69CC754   55 00000325    1
vmmdseg +69CC758   56 00000263    2
vmmdseg +69CC75C   57 0000014D    3
vmmdseg +69CC760   58 000001FE    6
...
KDB(4)> lkh 58 display VMM lock hash list 58

```

HASH ENTRY(58): B69CC760

		NEXT	TIDNXT	SID	PAGE	TID	FLAGS
510	vmm	695	445	0061BA	0103	0013	WRITE
695	vmm	478	817	007E7D	00C4	000C	WRITE FREE
478	vmm	669	778	006A78	00C1	009E	WRITE FREE
669	vmm	449	204	00326E	0057	004C	WRITE
449	vmm	593	782	00729E	0527	0007	WRITE BIGALLOC
593	vmm	0	815	00729E	0127	0007	WRITE BIGALLOC

lockword Subcommand

The `lkw` subcommand prints VMM lock words.

example:

KDB(4)> `lkw display VMM lock words`

		NEXT	TIDNXT	SID	PAGE	TID	FLAGS
0	vmm	0	0	000000	0000	0000	
1	vmm	620	679	00729E	0104	004C	WRITE FREE BIGALLOC
2	vmm	365	460	00729E	0169	00B7	WRITE FREE BIGALLOC
3	vmm	222	650	00729E	0163	00B7	WRITE FREE BIGALLOC
4	vmm	501	BEDCD140	0025A3	0000	0188	
5	vmm	748	115	00729E	0557	0025	WRITE FREE BIGALLOC
6	vmm	145	534	0061BA	0103	0046	WRITE FREE
7	vmm	79	586	006038	0080	0024	WRITE FREE
8	vmm	97	439	00224A	005C	0091	WRITE FREE
9	vmm	38	33	00729E	047F	00B7	WRITE FREE BIGALLOC
10	vmm	4	BEDD1820	0025A3	0000	0184	
11	vmm	BEDCD160	BEDCEA40	006B1B	0000	0070	
12	vmm	684	440	00729E	0062	004C	WRITE FREE BIGALLOC
13	vmm	736	402	00729E	0467	00B7	WRITE FREE BIGALLOC
14	vmm	0	BEDD3300	006B1B	0000	008C	
15	vmm	0	BEDCEAE0	006B1B	0000	0004	
16	vmm	BEDCDAE0	BEDD0840	007B3B	0000	0020	
17	vmm	109	78	001E85	0065	005D	WRITE FREE
18	vmm	0	005A74	007C	00A3	00A3	WRITE
19	vmm	563	797	00729E	0511	004C	WRITE FREE BIGALLOC
20	vmm	0	BEDCEB20	002D89	0000	001C	
21	vmm	0	0	000D86	0000	0047	WRITE
22	vmm	0	BEDD1460	007B3B	0000	0034	
23	vmm	505	234	00729E	009E	0007	WRITE BIGALLOC
24	vmm	30	614	00729E	0221	00B7	WRITE FREE BIGALLOC
25	vmm	660	244	007E7D	0101	0074	WRITE FREE
26	vmm	143	821	00729E	013C	00B7	WRITE FREE BIGALLOC
27	vmm	0	593	00729E	028D	0007	WRITE BIGALLOC
28	vmm	0	BEDD06A0	006B1B	0000	00B4	
29	vmm	701	407	00729E	016D	00B7	WRITE FREE BIGALLOC
30	vmm	75	24	00729E	0392	00B7	WRITE FREE BIGALLOC
31	vmm	0	BEDD0E00	006B1B	0000	0088	
32	vmm	477	BEDD1300	0025A3	0000	0144	
33	vmm	9	151	00729E	04D5	00B7	WRITE FREE BIGALLOC
34	vmm	178	589	001221	0075	0063	WRITE FREE
35	vmm	304	794	00729E	03D3	0025	WRITE FREE BIGALLOC
36	vmm	314	BEDCFBA0	0025A3	0000	0150	
37	vmm	682	149	006038	0082	00A1	WRITE FREE
38	vmm	555	9	00729E	021E	00B7	WRITE FREE BIGALLOC
39	vmm	218	322	00729E	0416	00B7	WRITE FREE BIGALLOC
40	vmm	207	66	006A78	005A	0030	WRITE FREE
41	vmm	244	307	005376	0000	0074	WRITE FREE
42	vmm	549	626	00729E	0420	004C	WRITE FREE BIGALLOC
43	vmm	155	830	00619C	0000	0081	WRITE FREE
44	vmm	118	BEDCFA80	00499A	0000	016C	
45	vmm	BEDD1280	BEDD3160	006B1B	0000	0068	

...

KDB(4)> `lkw 45 display VMM lock word 45`

		NEXT	TIDNXT	SID	PAGE	TID	FLAGS
45	vmm	BEDD1280	BEDD3160	006B1B	0000	0068	

```

bits..... 1000154A log..... 1000154B
home..... 10001540 extmem..... 100015C0
next..... BEDD1280 vmmmdseg +EDD1280
tidnxt..... BEDD3160 vmmmdseg +EDD3160
      NEXT  TIDNXT  SID PAGE  TID FLAGS
779 vmmmdseg +EDD3160 BEDCE660 BEDD0C20 006B1B 0000 0064
bits..... 10001480 log..... 10001483
home..... 10001500 extmem..... 10001501
next..... BEDCE660 vmmmdseg +EDCE660
tidnxt..... BEDD0C20 vmmmdseg +EDD0C20
      NEXT  TIDNXT  SID PAGE  TID FLAGS
481 vmmmdseg +EDD0C20 BEDCFAA0 BEDD1FA0 006B1B 0000 0060
bits..... 10001484 log..... 10001485
home..... 10001486 extmem..... 10001482
next..... BEDCFAA0 vmmmdseg +EDCFAA0
tidnxt..... BEDD1FA0 vmmmdseg +EDD1FA0
      NEXT  TIDNXT  SID PAGE  TID FLAGS
637 vmmmdseg +EDD1FA0 BEDD2200 BEDD1220 006B1B 0000 0040
bits..... 100012A3 log..... 100012A4
home..... 10001299 extmem..... 1000131C
next..... BEDD2200 vmmmdseg +EDD2200
tidnxt..... BEDD1220 vmmmdseg +EDD1220
      NEXT  TIDNXT  SID PAGE  TID FLAGS
529 vmmmdseg +EDD1220 BEDCF980 BEDD31A0 006B1B 0000 0028
bits..... 10001187 log..... 10001189
home..... 100011A3 extmem..... 1000118B
next..... BEDCF980 vmmmdseg +EDCF980
tidnxt..... BEDD31A0 vmmmdseg +EDD31A0
      NEXT  TIDNXT  SID PAGE  TID FLAGS
781 vmmmdseg +EDD31A0 BEDCD2C0 BEDCFB40 006B1B 0000 0014
bits..... 10001166 log..... 10001167
home..... 1000115A extmem..... 10001157
next..... BEDCD2C0 vmmmdseg +EDCD2C0
tidnxt..... BEDCFB40 vmmmdseg +EDCFB40
      NEXT  TIDNXT  SID PAGE  TID FLAGS
346 vmmmdseg +EDCFB40      0 BEDCFFC0 006B1B 0000 0058
bits..... 100013C1 log..... 100013C2
home..... 100013C3 extmem..... 10001400
tidnxt..... BEDCFFC0 vmmmdseg +EDCFFC0
      NEXT  TIDNXT  SID PAGE  TID FLAGS
382 vmmmdseg +EDCFFC0      0 BEDD15C0 006B1B 0000 005C
bits..... 10001403 log..... 10001488
home..... 10001489 extmem..... 1000148A
tidnxt..... BEDD15C0 vmmmdseg +EDD15C0
      NEXT  TIDNXT  SID PAGE  TID FLAGS
558 vmmmdseg +EDD15C0      0 BEDCFC40 006B1B 0000 0050
(4)> more ( C to quit) ?
bits..... 10001386 log..... 10001387
home..... 10001389 extmem..... 1000138C
tidnxt..... BEDCFC40 vmmmdseg +EDCFC40
      NEXT  TIDNXT  SID PAGE  TID FLAGS
354 vmmmdseg +EDCFC40      0 BEDD36E0 006B1B 0000 0054
bits..... 1000138A log..... 1000138B
home..... 10001382 extmem..... 10001385
tidnxt..... BEDD36E0 vmmmdseg +EDD36E0
      NEXT  TIDNXT  SID PAGE  TID FLAGS
823 vmmmdseg +EDD36E0      0 BEDD1D20 006B1B 0000 0010
bits..... 10001548 log..... 10001546
home..... 10001544 extmem..... 10001547
tidnxt..... BEDD1D20 vmmmdseg +EDD1D20
      NEXT  TIDNXT  SID PAGE  TID FLAGS
617 vmmmdseg +EDD1D20      0 BEDD2D40 006B1B 0000 0030
bits..... 100011A7 log..... 100011FC
home..... 100011FD extmem..... 100011E8
tidnxt..... BEDD2D40 vmmmdseg +EDD2D40
      NEXT  TIDNXT  SID PAGE  TID FLAGS
746 vmmmdseg +EDD2D40      0 BEDD16A0 006B1B 0000 000C

```

```

bits..... 10001553 log..... 10001554
home..... 10001545 extmem..... 10001541
tidnxt..... BEDD16A0 vmmidseg +EDD16A0
                        NEXT TIDNXT SID PAGE TID FLAGS
565 vmmidseg +EDD16A0 0 BEDD2C20 006B1B 0000 0020
bits..... 10001159 log..... 10001141
home..... 1000115D extmem..... 1000115C
tidnxt..... BEDD2C20 vmmidseg +EDD2C20
                        NEXT TIDNXT SID PAGE TID FLAGS
737 vmmidseg +EDD2C20 0 BEDCDAE0 006B1B 0000 0048
bits..... 1000130B log..... 1000131D
home..... 1000131A extmem..... 1000131B
tidnxt..... BEDCDAE0 vmmidseg +EDCDAE0
                        NEXT TIDNXT SID PAGE TID FLAGS
87 vmmidseg +EDCDAE0 0 BEDD2E80 006B1B 0000 0000
bits..... 1000108F log..... 10001110
home..... 1000114E extmem..... 1000114F
tidnxt..... BEDD2E80 vmmidseg +EDD2E80
                        NEXT TIDNXT SID PAGE TID FLAGS
756 vmmidseg +EDD2E80 0 BEDD0960 006B1B 0000 004C
bits..... 1000132B log..... 1000132C
home..... 10001342 extmem..... 10001388
tidnxt..... BEDD0960 vmmidseg +EDD0960
                        NEXT TIDNXT SID PAGE TID FLAGS
459 vmmidseg +EDD0960 0 BEDD1140 006B1B 0000 0034
bits..... 100011CF log..... 100011E2
home..... 100011D0 extmem..... 100011D1
tidnxt..... BEDD1140 vmmidseg +EDD1140
(4)> more (^C to quit) ?
                        NEXT TIDNXT SID PAGE TID FLAGS
522 vmmidseg +EDD1140 0 BEDCE580 006B1B 0000 0024
bits..... 10001188 log..... 10001184
home..... 10001186 extmem..... 1000118A
tidnxt..... BEDCE580 vmmidseg +EDCE580
                        NEXT TIDNXT SID PAGE TID FLAGS
172 vmmidseg +EDCE580 0 BEDCEC60 006B1B 0000 001C
bits..... 100011A0 log..... 1000119E
home..... 100011F1 extmem..... 100011F2
tidnxt..... BEDCEC60 vmmidseg +EDCEC60
                        NEXT TIDNXT SID PAGE TID FLAGS
227 vmmidseg +EDCEC60 0 BEDCD1E0 006B1B 0000 0008
bits..... 10001549 log..... 10001543
home..... 10001542 extmem..... 10001552
tidnxt..... BEDCD1E0 vmmidseg +EDCD1E0
                        NEXT TIDNXT SID PAGE TID FLAGS
15 vmmidseg +EDCD1E0 0 BEDCEAE0 006B1B 0000 0004
bits..... 10001155 log..... 10001173
home..... 10001140 extmem..... 10001156
tidnxt..... BEDCEAE0 vmmidseg +EDCEAE0
                        NEXT TIDNXT SID PAGE TID FLAGS
215 vmmidseg +EDCEAE0 0 BEDCE0E0 006B1B 0000 003C
bits..... 100011E4 log..... 100011E5
home..... 10001297 extmem..... 10001298
tidnxt..... BEDCE0E0 vmmidseg +EDCE0E0
                        NEXT TIDNXT SID PAGE TID FLAGS
135 vmmidseg +EDCE0E0 0 BEDCE440 006B1B 0000 0044
bits..... 10001318 log..... 1000133B
home..... 1000133C extmem..... 1000130F
tidnxt..... BEDCE440 vmmidseg +EDCE440
                        NEXT TIDNXT SID PAGE TID FLAGS
162 vmmidseg +EDCE440 0 BEDCF160 006B1B 0000 002C
bits..... 100011A4 log..... 100011A5
home..... 100011A6 extmem..... 10001185
tidnxt..... BEDCF160 vmmidseg +EDCF160
                        NEXT TIDNXT SID PAGE TID FLAGS
267 vmmidseg +EDCF160 0 BEDCF2E0 006B1B 0000 0038
bits..... 100011EA log..... 100011EB

```

```

home..... 100011C8 extmem..... 100011D5
tidnxt..... BEDCF2E0 vmmaseg +EDCF2E0
                NEXT  TIDNXT  SID PAGE  TID FLAGS
 279 vmmaseg +EDCF2E0      0      0 006B1B 0000 0018
bits..... 10001117 log..... 10001168
home..... 10001169 extmem..... 10001158
KDB(4)>

```

vmdmap Subcommand

The **vmdmap** subcommand prints VMM disk maps. Argument could be a PDT index. Default is all paging and file system disk map. To look at more, it is necessary to initialize the segment register 13 with the corresponding srrval.

example:

```

KDB(1)> vmdmap display VMM disk maps
PDT slot [0000] Vmdmap [D0000000] dmsrval [00000C03] <-- paging space 0
mapsize.....00007400 freecnt.....00004D22
agsize.....00000800 agcnt.....00000007
totalags.....0000000F lastalloc.....00003384
maptype.....00000003 clsize.....00000001
clmask.....00000080 version.....00000000
agfree@.....D0000030 tree@.....D00000A0
spare1@.....D00001F4 mapsorsummary@.....D0000200
PDT slot [0001] Vmdmap [D0800000] dmsrval [00000C03] <-- paging space 1
mapsize.....00005400 freecnt.....00003CF6
agsize.....00000800 agcnt.....00000007
totalags.....0000000B lastalloc.....000047F4
maptype.....00000003 clsize.....00000001
clmask.....00000080 version.....00000000
agfree@.....D0800030 tree@.....D08000A0
spare1@.....D08001F4 mapsorsummary@.....D0800200
PDT slot [0002] Vmdmap [D1000000] dmsrval [00000C03] <-- paging space 2
mapsize.....00005800 freecnt.....0000418C
agsize.....00000800 agcnt.....00000007
totalags.....0000000B lastalloc.....000047A8
maptype.....00000003 clsize.....00000001
clmask.....00000080 version.....00000000
agfree@.....D1000030 tree@.....D10000A0
spare1@.....D10001F4 mapsorsummary@.....D1000200
PDT slot [0011] Vmdmap [D0000000] dmsrval [00003C2F] <-- file system
mapsize.....00006400 freecnt.....000057CC
agsize.....00000800 agcnt.....00000007
totalags.....0000000D lastalloc.....00001412
maptype.....00000001 clsize.....00000008
clmask.....000000FF version.....00000000
agfree@.....D0000030 tree@.....D00000A0
spare1@.....D00001F4 mapsorsummary@.....D0000200
PDT slot [0013] Vmdmap [D0000000] dmsrval [00005455] <-- file system
mapsize.....00000800 freecnt.....0000030A
agsize.....00000400 agcnt.....00000002
totalags.....00000002 lastalloc.....0000011A
maptype.....00000001 clsize.....00000020
clmask.....00000000 version.....00000001
agfree@.....D0000030 tree@.....D00000A0
spare1@.....D00001F4 mapsorsummary@.....D0000200
...
KDB(1)> vmdmap 21 display VMM disk map slot 0x21
PDT slot [0021] Vmdmap [D0000000] dmsrval [000075BC]
mapsize.....00000800 freecnt.....000006B4
agsize.....00000800 agcnt.....00000001
totalags.....00000001 lastalloc.....00000060
maptype.....00000001 clsize.....00000008
clmask.....000000FF version.....00000000
agfree@.....D0000030 tree@.....D00000A0
spare1@.....D00001F4 mapsorsummary@.....D0000200

```


vmlocks Subcommand

The `vl` subcommand prints VMM spin locks.

For example:

```
KDB(1)> vl display VMM spin locks
GLOBAL LOCKS
pmap lock at @ 00000000 FREE
vmker lock at @ 0009A1AC LOCKED by thread: 0039AED
pdt lock at @ B69C84D4 FREE
vmap lock at @ B69C8514 FREE
ame lock at @ B69C8554 FREE
rpt lock at @ B69C8594 FREE
alloc lock at @ B69C85D4 FREE
apt lock at @ B69C8614 FREE
lw lock at @ B69C8678 FREE
SCOREBOARD
scoreboard cpu 0 :
hint.....00000000
00: empty
01: empty
02: empty
03: empty
04: empty
05: empty
06: empty
07: empty
scoreboard cpu 1 :
hint.....00000000
00: lock@ B6A31E60 lockword E804F380
01: empty
02: empty
03: empty
04: empty
05: empty
06: empty
07: empty
scoreboard cpu 2 :
hint.....00000002
00: lock@ B6A2851C lockword E8048B60
01: empty
02: empty
03: empty
04: empty
05: empty
06: empty
07: empty
scoreboard cpu 3 :
hint.....00000005
00: empty
(1)> more (^C to quit) ?
01: empty
02: empty
03: empty
04: lock@ B6AB04D8 lockword E8096E20
05: lock@ B69F2E54 lockword E8022760
06: empty
07: empty
scoreboard cpu 4 :
hint.....00000000
00: lock@ B6AAC380 lockword E8095740
01: empty
02: empty
03: empty
04: empty
05: empty
```

```

06: empty
07: empty
scoreboard cpu 5 :
hint.....00000001
00: lock@ B6A7BBE0  lockword E805CC40
01: lock@ B69CCD84  lockword E8000C80
02: empty
03: empty
04: empty
05: empty
06: empty
07: empty
scoreboard cpu 6 :
hint.....00000000
00: empty
01: empty
02: empty
03: empty
04: empty
05: empty
06: empty
07: empty
scoreboard cpu 7 :
hint.....00000001
00: empty
01: lock@ B6AA8FF8  lockword E807CA00
02: empty
03: empty
04: empty
05: empty
06: empty
07: empty
KDB(1)>

```

SMP Subcommands for the KDB Kernel Debugger and kdb Command

Note: The subcommands in this section are only valid for SMP machines.

KDB processor states are:

- running, outside **kdb**
- stopped, after a stop subcommand
- switched, after a cpu subcommand
- debug waiting, after a break point
- debug, inside **kdb**.

start and stop Subcommands

Note: This subcommand is only available within the **kdb** command; it is not included in the KDB Kernel Debugger.

The **stop** subcommand may be used to stop the specified processor. The **start** subcommand may be used to start the specified processor. MPC is used to test an incoming **kdb** subcommand. When the processor is stopped, it is looping inside KDB. Stopped means that the processor does not go back to AIX.

Example

```

KDB(1)> stop 0 stop processor 0
KDB(1)> cpu display processors status
cpu 0 status VALID STOPPED action STOP
cpu 1 status VALID DEBUG

```

```

KDB(1)> start 0 start processor 0
KDB(1)> cpu display processors status
cpu 0 status VALID action START
cpu 1 status VALID DEBUG
KDB(1)> b sy_decint set break point
KDB(1)> e exit the debugger
Breakpoint
.sy_decint+000000    mflr    r0                <.dec_flih+000014>
KDB(0)> cpu display processors status
cpu 0 status VALID DEBUG action RESUME
cpu 1 status VALID DEBUGWAITING
KDB(0)> cpu 1 switch to processor 1
Breakpoint
.sy_decint+000000    mflr    r0                <.dec_flih+000014>
KDB(1)> cpu display processors status
cpu 0 status VALID SWITCHED action SWITCH
cpu 1 status VALID DEBUG
KDB(1)> cpu 0 switch to processor 0
KDB(0)> cpu display processors status
cpu 0 status VALID DEBUG
cpu 1 status VALID SWITCHED action SWITCH
KDB(0)> q exit the debugger

```

cpu Subcommand

The **cpu** subcommand may be used to switch from the current processor to the specified processor. Without argument, the **cpu** subcommand prints processor status. The switched processor is blocked until next **start** or **cpu** subcommand. Switching between processors do not change processor state.

Note: If the selected processor can not be reached, it is possible to go back to the previous one by typing `^\\` twice.

Example

```

KDB(4)> cpu display processors status
cpu 0 status VALID SWITCHED action SWITCH
cpu 1 status VALID SWITCHED action SWITCH
cpu 2 status VALID SWITCHED action SWITCH
cpu 3 status VALID SWITCHED action SWITCH
cpu 4 status VALID DEBUG action RESUME
cpu 5 status VALID SWITCHED action SWITCH
cpu 6 status VALID SWITCHED action SWITCH
cpu 7 status VALID SWITCHED action SWITCH
KDB(4)> cpu 7 switch to processor 7
Debugger entered via keyboard.
.waitproc+0000B0    lbz    r0,0(r30)                r0=0,0(r30)=ppda+0014D0
KDB(7)> cpu display processors status
cpu 0 status VALID SWITCHED action SWITCH
cpu 1 status VALID SWITCHED action SWITCH
cpu 2 status VALID SWITCHED action SWITCH
cpu 3 status VALID SWITCHED action SWITCH
cpu 4 status VALID SWITCHED action SWITCH
cpu 5 status VALID SWITCHED action SWITCH
cpu 6 status VALID SWITCHED action SWITCH
cpu 7 status VALID DEBUG
KDB(7)>

```

bat/Block Address Translation Subcommands for the KDB Kernel Debugger and kdb Command

dbat Subcommand

On POWER PC machine, the **dbat** subcommand may be used to display **dbat** registers. (See *POWER PC Operating Environment Architecture* (book III) and *POWER PC Implementation Definition for the Processor* (book IV) to have more information about **bat** register fields).

Example

```
KDB(3)> dbat display POWER 601 BAT registers
BAT0 00000000 00000000
  bepi 0000 brpn 0000 bl 0000 v 0 ks 0 kp 0 wimg 0 pp 0
BAT1 00000000 00000000
  bepi 0000 brpn 0000 bl 0000 v 0 ks 0 kp 0 wimg 0 pp 0
BAT2 00000000 00000000
  bepi 0000 brpn 0000 bl 0000 v 0 ks 0 kp 0 wimg 0 pp 0
BAT3 00000000 00000000
  bepi 0000 brpn 0000 bl 0000 v 0 ks 0 kp 0 wimg 0 pp 0
KDB(1)> dbat display POWER 604 data BAT registers
DBAT0 00000000 00000000
  bepi 0000 brpn 0000 bl 0000 vs 0 vp 0 wimg 0 pp 0
DBAT1 00000000 00000000
  bepi 0000 brpn 0000 bl 0000 vs 0 vp 0 wimg 0 pp 0
DBAT2 00000000 00000000
  bepi 0000 brpn 0000 bl 0000 vs 0 vp 0 wimg 0 pp 0
DBAT3 00000000 00000000
  bepi 0000 brpn 0000 bl 0000 vs 0 vp 0 wimg 0 pp 0
KDB(0)> dbat display POWER 620 data BAT registers
DBAT0 0000000000000000 000000000000001A
  bepi 000000000000 brpn 000000000000 bl 0000 vs 0 vp 0 wimg 3 pp 2
DBAT1 0000000000000000 00000000C000002A
  bepi 000000000000 brpn 000000006000 bl 0000 vs 0 vp 0 wimg 5 pp 2
DBAT2 0000000000000000 000000008000002A
  bepi 000000000000 brpn 000000004000 bl 0000 vs 0 vp 0 wimg 5 pp 2
DBAT3 0000000000000000 00000000A000002A
  bepi 000000000000 brpn 000000005000 bl 0000 vs 0 vp 0 wimg 5 pp 2
```

ibat Subcommand

On POWER PC machine, the **ibat** subcommand may be used to display **ibat** registers. (See *POWER PC Operating Environment Architecture* (book III) and *POWER PC Implementation Definition for the Processor* (book IV) for more information about **bat** register fields).

Example

```
KDB(0)> ibat display POWER 601 BAT registers
BAT0 00000000 00000000
  bepi 0000 brpn 0000 bl 0000 v 0 ks 0 kp 0 wimg 0 pp 0
BAT1 00000000 00000000
  bepi 0000 brpn 0000 bl 0000 v 0 ks 0 kp 0 wimg 0 pp 0
BAT2 00000000 00000000
  bepi 0000 brpn 0000 bl 0000 v 0 ks 0 kp 0 wimg 0 pp 0
BAT3 00000000 00000000
  bepi 0000 brpn 0000 bl 0000 v 0 ks 0 kp 0 wimg 0 pp 0
KDB(2)> ibat display POWER 604 instruction BAT registers
IBAT0 00000000 00000000
  bepi 0000 brpn 0000 bl 0000 vs 0 vp 0 wimg 0 pp 0
IBAT1 00000000 00000000
  bepi 0000 brpn 0000 bl 0000 vs 0 vp 0 wimg 0 pp 0
IBAT2 00000000 00000000
  bepi 0000 brpn 0000 bl 0000 vs 0 vp 0 wimg 0 pp 0
```

```

IBAT3 00000000 00000000
  bepi 0000 brpn 0000 b1 0000 vs 0 vp 0 wimg 0 pp 0
KDB(0)> ibat display POWER 620 instruction BAT registers
IBAT0 0000000000000000 0000000000000000
  bepi 000000000000 brpn 000000000000 b1 0000 vs 0 vp 0 wimg 0 pp 0
IBAT1 0000000000000000 0000000000000000
  bepi 000000000000 brpn 000000000000 b1 0000 vs 0 vp 0 wimg 0 pp 0
IBAT2 0000000000000000 0000000000000000
  bepi 000000000000 brpn 000000000000 b1 0000 vs 0 vp 0 wimg 0 pp 0
IBAT3 0000000000000000 0000000000000000
  bepi 000000000000 brpn 000000000000 b1 0000 vs 0 vp 0 wimg 0 pp 0

```

mdbat Subcommand

Each **dbat** register may be altered by the **mdbat** subcommand. The processor data **bat** register is altered immediately. KDB takes care of the valid bit, the word with valid bit is set at last.

Example

```

On POWER 601 processor
KDB(0)> dbat 2 display bat register 2
BAT2: 00000000 00000000
  bepi 0000 brpn 0000 b1 0000 v 0 wimg 0 ks 0 kp 0 pp 0
KDB(0)> mdbat 2 alter bat register 2
BAT register, enter <RC> twice to select BAT field, enter <.> to quit
BAT2 upper 00000000 = <CR/LF>
BAT2 lower 00000000 = <CR/LF>
BAT field, enter <RC> to select field, enter <.> to quit
BAT2.bepi: 00000000 = 00007FE0
BAT2.brpn: 00000000 = 00007FE0
BAT2.b1 : 00000000 = 0000001F
BAT2.v : 00000000 = 00000001
BAT2.ks : 00000000 = 00000001
BAT2.kp : 00000000 = <CR/LF>
BAT2.wimg: 00000000 = 00000003
BAT2.pp : 00000000 = 00000002
BAT2: FFC0003A FFC0005F
  bepi 7FE0 brpn 7FE0 b1 001F v 1 wimg 3 ks 1 kp 0 pp 2
  eaddr = FFC00000, paddr = FFC00000 size = 4096 KBytes
KDB(0)> mdbat 2 clear bat register 2
BAT register, enter <RC> twice to select BAT field, enter <.> to quit
BAT2 upper FFC0003A = 0
BAT2 lower FFC0005F = 0
BAT2 00000000 00000000
  bepi 0000 brpn 0000 b1 0000 v 0 wimg 0 ks 0 kp 0 pp 0
On POWER 604 processor
KDB(0)> mdbat 2 alter bat register 2
BAT register, enter <RC> twice to select BAT field, enter <.> to quit
DBAT2 upper 00000000 =
DBAT2 lower 00000000 =
BAT field, enter <RC> to select field, enter <.> to quit
DBAT2.bepi: 00000000 = 00007FE0
DBAT2.brpn: 00000000 = 00007FE0
DBAT2.b1 : 00000000 = 0000001F
DBAT2.vs : 00000000 = 00000001
DBAT2.vp : 00000000 = <CR/LF>
DBAT2.wimg: 00000000 = 00000003
DBAT2.pp : 00000000 = 00000002
DBAT2 FFC0007E FFC0001A
  bepi 7FE0 brpn 7FE0 b1 001F vs 1 vp 0 wimg 3 pp 2
  eaddr = FFC00000, paddr = FFC00000 size = 4096 KBytes [Supervisor state]
KDB(0)> mdbat 2 clear bat register 2
BAT register, enter <RC> twice to select BAT field, enter <.> to quit
DBAT2 upper FFC0007E = 0

```

```

DBAT2 lower FFC0001A = 0
DBAT2 00000000 00000000
bepi 0000 brpn 0000 bl 0000 vs 0 vp 0 wimg 0 pp 0

```

mibat Subcommand

Each **ibat** register may be altered by the **mibat** subcommand. The processor instruction **bat** register is altered immediately.

Example

```

On POWER 601 processor
KDB(0)> ibat 2 display bat register 2
BAT2: 00000000 00000000
bepi 0000 brpn 0000 bl 0000 v 0 wimg 0 ks 0 kp 0 pp 0
KDB(0)> mibat 2 alter bat register 2
BAT register, enter <RC> twice to select BAT field, enter <.> to quit
BAT2 upper 00000000 = <CR/LF>
BAT2 lower 00000000 = <CR/LF>
BAT field, enter <RC> to select field, enter <.> to quit
BAT2.bepi: 00000000 = 00007FE0
BAT2.brpn: 00000000 = 00007FE0
BAT2.bl : 00000000 = 0000001F
BAT2.v : 00000000 = 00000001
BAT2.ks : 00000000 = 00000001
BAT2.kp : 00000000 = <CR/LF>
BAT2.wimg: 00000000 = 00000003
BAT2.pp : 00000000 = 00000002
BAT2: FFC0003A FFC0005F
bepi 7FE0 brpn 7FE0 bl 001F v 1 wimg 3 ks 1 kp 0 pp 2
eaddr = FFC00000, paddr = FFC00000 size = 4096 KBytes
KDB(0)> mibat 2 clear bat register 2
BAT register, enter <RC> twice to select BAT field, enter <.> to quit
BAT2 upper FFC0003A = 0
BAT2 lower FFC0005F = 0
BAT2 00000000 00000000
bepi 0000 brpn 0000 bl 0000 v 0 wimg 0 ks 0 kp 0 pp 0
On POWER 604 processor
KDB(0)> mibat 2
BAT register, enter <RC> twice to select BAT field, enter <.> to quit
IBAT2 upper 00000000 = <CR/LF>
IBAT2 lower 00000000 = <CR/LF>
BAT field, enter <RC> to select field, enter <.> to quit
IBAT2.bepi: 00000000 = <CR/LF>
IBAT2.brpn: 00000000 = <CR/LF>
IBAT2.bl : 00000000 = 3ff
IBAT2.vs : 00000000 = 1
IBAT2.vp : 00000000 = <CR/LF>
IBAT2.wimg: 00000000 = 2
IBAT2.pp : 00000000 = 2
IBAT2 00000FFE 00000012
bepi 0000 brpn 0000 bl 03FF vs 1 vp 0 wimg 2 pp 2
eaddr = 00000000, paddr = 00000000 size = 131072 KBytes [Supervisor state]

```

btac/BRAT Subcommands for the KDB Kernel Debugger and kdb Command

btac, cbtac, lbtac, lcbtac Subcommands

Note: This subcommand is only available within the **kdb** command; it is not included in the KDB Kernel Debugger.

On POWER PC architecture, a hardware register may be used (called **HID2** on POWER 601) to enter KDB when a specified effective address is decoded. The **HID2** register holds the effective address, and the **HID1** register specifies full branch target address compare and trap to address vector 0x1300 (0x2000 on 601). See *PowerPC Implementation Definition for the 601-604-620 Processor* (book IV) to have more information. The **btac** subcommand may be used to stop when Branch Target Address Compare is true. The **cbtac** subcommand may be used to clear the last **btac** subcommand. This subcommand is global to all processors, each processor may have different address to compare with the local subcommands **lbtac** **lcbtac**.

It is possible to specify if the address is physical or virtual with **-p** or **-v** option. 604 and 620 processors take care of the translation mode, and it is necessary to specify which mode is used. By default KDB chooses the current state of the machine: if the subcommand is entered before VMM initialisation, the address is physical (real address), else virtual (effective address).

Example

```

KDB(7)> btac open set BRAT on open function
KDB(7)> btac display current BRAT status
CPU 0: .open+000000 eaddr=001B5354 vsid=00000000 hit=0
CPU 1: .open+000000 eaddr=001B5354 vsid=00000000 hit=0
CPU 2: .open+000000 eaddr=001B5354 vsid=00000000 hit=0
CPU 3: .open+000000 eaddr=001B5354 vsid=00000000 hit=0
CPU 4: .open+000000 eaddr=001B5354 vsid=00000000 hit=0
CPU 5: .open+000000 eaddr=001B5354 vsid=00000000 hit=0
CPU 6: .open+000000 eaddr=001B5354 vsid=00000000 hit=0
CPU 7: .open+000000 eaddr=001B5354 vsid=00000000 hit=0
KDB(7)> e exit the debugger
...
Branch trap: 001B5354 <.open+000000>
.sys_call+000000 bcctrl <.open>
KDB(5)> btac display current BRAT status
CPU 0: .open+000000 eaddr=001B5354 vsid=00000000 hit=0
CPU 1: .open+000000 eaddr=001B5354 vsid=00000000 hit=0
CPU 2: .open+000000 eaddr=001B5354 vsid=00000000 hit=0
CPU 3: .open+000000 eaddr=001B5354 vsid=00000000 hit=0
CPU 4: .open+000000 eaddr=001B5354 vsid=00000000 hit=0
CPU 5: .open+000000 eaddr=001B5354 vsid=00000000 hit=1
CPU 6: .open+000000 eaddr=001B5354 vsid=00000000 hit=0
CPU 7: .open+000000 eaddr=001B5354 vsid=00000000 hit=0
KDB(5)> lbtac close set local BRAT on close function
KDB(5)> e exit the debugger
...
Branch trap: 001B5354 <.open+000000>
.sys_call+000000 bcctrl <.open>
KDB(7)> e exit the debugger
...
Branch trap: 00197D40 <.close+000000>
.sys_call+000000 bcctrl <.close>
KDB(5)> e exit the debugger
...
Branch trap: 001B5354 <.open+000000>
.sys_call+000000 bcctrl <.open>
KDB(6)> btac display current BRAT status
CPU 0: .open+000000 eaddr=001B5354 vsid=00000000 hit=0
CPU 1: .open+000000 eaddr=001B5354 vsid=00000000 hit=0
CPU 2: .open+000000 eaddr=001B5354 vsid=00000000 hit=0
CPU 3: .open+000000 eaddr=001B5354 vsid=00000000 hit=0
CPU 4: .open+000000 eaddr=001B5354 vsid=00000000 hit=0
CPU 5: .close+000000 eaddr=00197D40 vsid=00000000 hit=1
CPU 6: .open+000000 eaddr=001B5354 vsid=00000000 hit=1
CPU 7: .open+000000 eaddr=001B5354 vsid=00000000 hit=1
KDB(6)> cbtac reset all BRAT registers

```

machdep Subcommands for the KDB Kernel Debugger and kdb Command

reboot Subcommand

Note: This subcommand is only available within the **kdb** command; it is not included in the KDB Kernel Debugger.

The **reboot** subcommand may be used to reboot the machine. The soft reboot interface is called (**sr_slrh(1)**).

Example

```
KDB(0)> reboot reboot the machine
Rebooting ...
```

Using the KDB Kernel Debug Program

The example files provide a demonstration kernel extension and a program to load, execute, and unload the extension. These programs may be compiled, linked, and executed as indicated in the following material. Note, to use these programs to follow the examples you need a machine with a C compiler, a console, and running with a KDB kernel enabled for debugging. To use the KDB Kernel Debugger you will need exclusive use of the machine.

Examples using the KDB Kernel Debugger with the demonstration programs are included in each of the following sections. The examples are shown in tables which contain two columns. The first column of the table contains an indication of the system prompt and the user input to perform each step. The second column of each table explains the function of the command and includes example output, where applicable. In the examples, since only the console is used, the demo program is switched between the background and the foreground as needed.

Example Files

The files listed below are used in examples throughout this section.

- **demo.c** - Source program to load, execute, and unload a demonstration kernel extension.
- **demokext.c** - Source for a demonstration kernel extension
- **demo.h** - Include file used by **demo.c** and **demokext.c**
- **demokext.exp** - Export file for linking **demokext**
- **comp_link** - Example script to build demonstration program and kernel extension

To build the demonstration programs:

- Save each of the above files in a directory
- As the root user, execute the **comp_link** script

This script produces:

- An executable file **demo**
- An executable file **demokext**
- A list file **demokext.lst**
- A map file **demokext.map**

The following sections describe compilation and link options used in the **comp_link** script in more detail and also cover using the map and list files.

Generating Maps and Listings

Assembler listing and map files are useful tools for debugging using the KDB Kernel Debugger. In order to create the assembler list file during compilation, use the `-qlist` option. Also use the `-qsource` option to get the C source listing in the same file:

```
cc -c -DEBUG -D_KERNEL -DIBMR2 demokext.c -qsource -qlist
```

In order to obtain a map file, use the `-bmap:FileName` option for the link editor. The following example creates a map file of `demokext.map`:

```
ld -o demokext demokext.o -edemokext -bimport:/lib/syscalls.exp \  
-bimport:/lib/kernex.exp -lcsys -bexport:demokext.exp -bmap:demokext.map
```

Compiler Listing

The assembler and source listing is used to correlate any C source line with the corresponding assembler lines. The following is a portion of the list file, created by the `cc` command used earlier, for the demonstration kernel extension. This information is included in the compilation listing because of the `-qsource` option for the `cc` command. The left column is the line number in the source code:

```
.  
:  
63 | case 1: /* Increment */  
64 |     sprintf(buf, "Before increment: j=%d demokext_j=%d\n",  
65 |           j, demokext_j);  
66 |     write_log(fpp, buf, &bytes_written);  
67 |     demokext_j++;  
68 |     j++;  
69 |     sprintf(buf, "After increment: j=%d demokext_j=%d\n",  
70 |           j, demokext_j);  
71 |     write_log(fpp, buf, &bytes_written);  
72 |     break;  
:  
:
```

The following is the assembler listing for the corresponding C code shown above. This information was included in the compilation listing because of the `-qlist` option used on the `cc` command earlier.

```
.  
:  
64 | 0000B0 l      80BF0030 2    L4A    gr5=j(gr31,48)  
64 | 0000B4 l      83C20008 1    L4A    gr30=.demokext_j(gr2,0)  
64 | 0000B8 l      80DE0000 2    L4A    gr6=demokext_j(gr30,0)  
64 | 0000BC ai     30610048 1    AI     gr3=gr1,72  
64 | 0000C0 ai     309F005C 1    AI     gr4=gr31,92  
64 | 0000C4 bl     4BFFFF3D 0    CALL   gr3=sprintf,4,buf",gr3,""5",gr4-gr6,sprintf",  
        gr1,cr[01567]","gr0",gr4"-gr12",fp0"-fp13"  
  
64 | 0000C8 cror   4DEF7B82 1  
66 | 0000CC l      80610040 1    L4A    gr3=fpp(gr1,64)  
66 | 0000D0 ai     30810048 1    AI     gr4=gr1,72  
66 | 0000D4 ai     30A100AC 1    AI     gr5=gr1,172  
66 | 0000D8 bl     4800018D 0    CALL   gr3=write_log,3,gr3,buf",gr4,bytes_written",  
        gr5,write_log",gr1,cr[01567]","gr0",  
        gr4"-gr12",fp0"-fp13"  
  
66 | 0000DC cal    387E0000 2    LR     gr3=gr30  
67 | 0000E0 l      80830000 1    L4A    gr4=demokext_j(gr3,0)  
67 | 0000E4 ai     30840001 2    AI     gr4=gr4,1  
67 | 0000E8 st     90830000 1    ST4A   demokext_j(gr3,0)=gr4  
68 | 0000EC l      809F0030 1    L4A    gr4=j(gr31,48)  
68 | 0000F0 ai     30A40001 2    AI     gr5=gr4,1  
68 | 0000F4 st     90BF0030 1    ST4A   j(gr31,48)=gr5  
69 | 0000F8 l      80C30000 1    L4A    gr6=demokext_j(gr3,0)  
69 | 0000FC ai     30610048 1    AI     gr3=gr1,72
```

```

69 | 000100 ai      309F0084  1    AI      gr4=gr31,132
69 | 000104 bl      4BFFFEFD  0    CALL    gr3=sprintf,4,buf",gr3,""6",gr4-gr6,sprintf",
        gr1,cr[01567]"",gr0",gr4"-gr12",fp0"-fp13"

69 | 000108 cror    4DEF7B82  1
71 | 00010C l       80610040  1    L4A     gr3=fpp(gr1,64)
71 | 000110 ai      30810048  1    AI      gr4=gr1,72
71 | 000114 ai      30A100AC  1    AI      gr5=gr1,172
71 | 000118 bl      4800014D  0    CALL    gr3=write_log,3,gr3,buf",gr4,bytes_written",
        gr5,write_log",gr1,cr[01567]"",gr0",
        gr4"-gr12",fp0"-fp13"

72 | 00011C b       48000098  1    B       CL.8,-1
.
.

```

With both the assembler listing and the C source listing, the assembly instructions associated with each C statement may be found. As an example, consider the C source line at line 67 of the demonstration kernel extension:

```
67 | demokext_j++;
```

The corresponding assembler instructions are:

```

67 | 0000E0 l       80830000  1    L4A     gr4=demokext_j(gr3,0)
67 | 0000E4 ai      30840001  2    AI      gr4=gr4,1
67 | 0000E8 st      90830000  1    ST4A    demokext_j(gr3,0)=gr4

```

The offsets of these instructions within the demonstration kernel extension (demokext) are 0000E0, 0000E4, and 0000E8.

Map File

The binder map file is a symbol map in address order format. Each symbol listed in the map file has a storage class (CL) and a type (TY) associated with it.

Storage classes correspond to the **XMC_XX** variables defined in the **syms.h** file. Each storage class belongs to one of the following section types:

- .text** Contains read-only data (instructions). Addresses listed in this section use the beginning of the **.text** section as origin. The **.text** section can contain one of the following storage class (CL) values:
 - DB** Debug Table. Identifies a class of sections that has the same characteristics as read only data.
 - GL** Glue Code. Identifies a section that has the same characteristics as a program code. This type of section has code to interface with a routine in another module. Part of the interface code requirement is to maintain TOC addressability across the call.
 - PR** Program Code. Identifies the sections that provide executable instructions for the module.
 - R0** Read Only Data. Identifies the sections that contain constants that are not modified during execution.
 - TB** Reserved.
 - TI** Reserved.
 - XO** Extended Op. Identifies a section of code that is to be treated as a pseudo-machine instruction.

- .data** Contains read-write initialized data. Addresses listed in this section use the beginning of the **.data** section as origin. The **.data** section can contain one of the following storage class (CL) values:
- DS** Descriptor. Identifies a function descriptor. This information is used to describe function pointers in languages such as C and Fortran.
 - RW** Read Write Data. Identifies a section that contains data that is known to require change during execution.
 - SV** SVC. Identifies a section of code that is to be treated as a supervisory call.
 - T0** TOC Anchor. Used only by the predefined TOC symbol. Identifies the special symbol TOC. Used only by the TOC header.
 - TC** TOC Entry. Identifies address data that will reside in the TOC.
 - TD** TOC Data Entry. Identifies data that will reside in the TOC.
 - UA** Unclassified. Identifies data that contains data of an unknown storage class.
- .bss** Contains read-write uninitialized data. Addresses listed in this section use the beginning of the **.data** section as origin. The **.bss** section contain one of the following storage class (CL) values:
- BS** BSS class. Identifies a section that contains uninitialized data.
 - UC** Unnamed Fortran Common. Identifies a section that contains read write data.

Types correspond to the **XTY_XX** variables defined in the **syms.h** file. The type (TY) can be one of the following values:

- ER** External Reference
- LD** Label Definition
- SD** Section Definition
- CM** BSS Common Definition

The following is the map file for the demonstration kernel extension. This file was created because of the **-bmap:demokext.map** option of the **ld** command shown earlier.

```

1 ADDRESS MAP FOR demokext
2 *IE ADDRESS LENGTH AL CL TY Sym# NAME SOURCE-FIL E(OBJECT) or
IMPORT-FILE{SHARED-OBJECT}
3
4 I ER S1 _system_configuration /lib/syscalls.exp{/unix}
5 I ER S2 fp_open /lib/kernex.exp{/unix}
6 I ER S3 fp_close /lib/kernex.exp{/unix}
7 I ER S4 fp_write /lib/kernex.exp{/unix}
8 I ER S5 sprintf /lib/kernex.exp{/unix}
9 00000000 000360 2 PR SD S6 <> demokext.c(demokext.o)
10 00000000 PR LD S7 .demokext
11 00000210 PR LD S8 .close_log
12 00000264 PR LD S9 .write_log
13 000002F4 PR LD S10 .open_log
14 00000360 000108 5 PR SD S11 .strcpy strcpy.s(/usr/lib/
libcsys.a[strcpy.o])
15 00000468 000028 2 GL SD S12 <.sprintf> glink.s(/usr/lib/glink.o)
16 00000468 GL LD S13 .sprintf
17 00000490 000028 2 GL SD S14 <.fp_close> glink.s(/usr/lib/glink.o)
18 00000490 GL LD S15 .fp_close
19 000004C0 0000F8 5 PR SD S16 .strlen strlen.s(/usr/lib/

```

```

20      000005B8 000028 2 GL SD S17  <.fp_write>          libcsys.a[strlen.o])
21      000005B8          GL LD S18  .fp_write          glink.s(/usr/lib/glink.o)
22      000005E0 000028 2 GL SD S19  <.fp_open>         glink.s(/usr/lib/glink.o)
23      000005E0          GL LD S20  .fp_open
24      00000000 0000F9 3 RW SD S21  <_STATIC>         demokext.c(demokext.o)
25      E 000000FC 000004 2 RW SD S22  demokext_j        demokext.c(demokext.o)
26      * 00000100 00000C 2 DS SD S23  demokext          demokext.c(demokext.o)
27      0000010C 000000 2 T0 SD S24  <TOC>
28      0000010C 000004 2 TC SD S25  <_STATIC>
29      00000110 000004 2 TC SD S26  <_system_configuration>
30      00000114 000004 2 TC SD S27  <demokext_j>
31      00000118 000004 2 TC SD S28  <sprintf>
32      0000011C 000004 2 TC SD S29  <fp_close>
33      00000120 000004 2 TC SD S30  <fp_write>
34      00000124 000004 2 TC SD S31  <fp_open>

```

In the above map file, the **.data** section starts at the statement for line 24:

```

24      00000000 0000F9 3 RW SD S21  <_STATIC>         demokext.c(demokext.o)

```

The TOC (Table Of Contents) starts at the statement for line 27:

```

27      0000010C 000000 2 T0 SD S24  <TOC>

```

Setting Breakpoints

Setting a breakpoint is essential for debugging kernel extensions. To set a breakpoint, use the following sequence of steps:

1. Locate the assembler instruction corresponding to the C statement.
2. Get the offset of the assembler instruction from the listing.
3. Locate the address where the kernel extension is loaded.
4. Add the address of the assembler instruction to the address where kernel extension is loaded.
5. Set the breakpoint with the KDB **b** (break) subcommand.

The process of locating the assembler instruction and getting its offset is explained in the previous section. To continue with the demokext example, we will set a break at the C source line 67, which increments the variable *demokext_j*. The list file indicates that this line starts at an offset of 0xE0. So the next step is to determine the address where the kernel extension is loaded.

Determine the Location of your Kernel Extension

To determine the address at which a kernel extension has been loaded, use the following procedure. First, find the load point (the entry point) of the executable kernel extension. This is a label supplied with the **-e** option for the **ld** command. In the example this is the **demokext** routine.

Use one of the following methods to locate the address of this load point and set a breakpoint at the appropriate offset from this point.

- **Method 1**

Normally, with the KDB Kernel Debugger a breakpoint may be set directly by using the **b** subcommand in conjunction with the routine name and the offset. For example, **b demokext+4** will set a break at the instruction 4 bytes from the beginning of the demokext routine.

- **Method 2**

The KDB **lke** subcommand displays a list of loaded kernel extensions. To find the address of the modules for a particular extension use the KDB subcommand **lke entry_number**, where entry_number is the extension number of interest. In

the displayed data is a list of Process Trace Backs which shows the beginning addresses of routines contained in the extension.

- **Method 3**

If the kernel extension is not stripped the KDB Kernel Debugger may be used to locate the address of the load point by name. For example the subcommand **nm demokext** returns the address of the demokext routine after it is loaded. This address may then be used to set a breakpoint.

- **Method 4**

Another method to locate the address of the entry point for a kernel extension is to use the value of the **kmid** pointer returned by the **sysconfig(SYS_KLOAD)** subroutine when the kernel extension is loaded. The **kmid** pointer points to the address of the load point routine. Hence to get the address of the load point, print the **kmid** value during the **sysconfig** call from the configuration method; in the current example, this is the demo.c module. Then go into the KDB Kernel Debugger and display the value pointed to by **kmid**.

- **Method 5**

If the kernel extension is a device driver, use the KDB **devsw** subcommand to locate the desired address. The **devsw** subcommand lists all the function addresses for the device driver (that are in the dev switch table). Usually the **config** routine will be the load point routine.

```
MAJ#010  OPEN          CLOSE          READ          WRITE
         0123DE04     0123DC04     0123DB20     0123DA3C
         IOCTL        STRATEGY      TTY          SELECT
         0123D090     01244DF0     00000000     00059774
         CONFIG      PRINT        DUMP        MPX
         0123E8C8     00059774     00059774     00059774
         REVOKE      DSDPTR      SELPTR      OPTS
         00059774     00000000     00000000     00000002
```

The following provides examples of each of the above methods using the demo and demokext routines compiled earlier. Note, the following must be run as the root user. For this example, assume that a break is to be set at line 67, which has an offset from the beginning of demokext of 0xE0.

Prompt and Console Input	Function and Example Output
Load the demokext kernel extension	
\$./demo	Run the demo program, this loads the demokext extension. Note, the value printed for kmid, this is used later in this example.
\$ <CTRL-Z>	Stop the demo program.
\$ bg	Put the demo program in the background.
\$ <CTRL-\>	Activate KDB, use the appropriate key sequence for your configuration. You should have a KDB prompt on completion of this step.
Set a breakpoint using Method 1	
KDB(0)> b demokext+E0	Set a breakpoint using the symbol demokext. This is the easiest and most common way of setting a breakpoint within KDB. KDB responds with an indication of the address where the break is set.
KDB(0)> b	List all breakpoints. KDB displays a list of all breakpoints currently active.
KDB(0)> ca	This clears all breakpoints
KDB(0)> b	List all breakpoints. KDB indicates there are no active breakpoints.
Set a breakpoint using Method 2	

Prompt and Console Input	Function and Example Output
KDB(0)> lke	<p>List loaded extensions. The output from this subcommand will be similar to:</p> <pre> ADDRESS FILE FILESIZE FLAGS MODULE NAME 1 04E17F80 01303F00 000007F0 00000272 ./demokext 2 04E17E80 0503A000 00000E88 00000248 /unix 3 04E17C00 04FA3000 00071B34 00000272 /usr/lib/ drivers/nfs.ext 4 04E17A80 05021000 00000E88 00000248 /unix 5 04E17800 01303B98 00000348 00000272 /usr/lib/ drivers/nfs_kdes.ext 6 04E17B80 04F96000 00000E34 00000248 /unix 7 04E17500 01301A10 0000217C 00000272 /etc/drivers/ blockset64 . .</pre> <p>Enter <CTRL-C> to exit the KDB Kernel Debugger paging function, if more than one page of data is present. You may exit the paging function at any time by using <CTRL-C>. Pressing <ENTER> displays the next page of data; <SPACE> displays the next line of data. Additionally, the number of lines per page may be changed using the <i>set screen_size xx</i> subcommand; where xx is the number of lines to be considered a page.</p>
KDB(0)> lke 1	<p>List detailed information about the extension of interest. The parameter to the <i>lke</i> subcommand is the slot number for the <i>./demokext</i> entry from the previous step. The output from this command will be similar to:</p> <pre> ADDRESS FILE FILESIZE FLAGS MODULE NAME 1 04E17F80 01303F00 000007F0 00000272 ./demokext le_flags..... TEXT KERNELEX DATAINTEXT DATA DATAEXISTS le_next..... 04E17E80 le_fp..... 00000000 le_filename... 04E17FD8 le_file..... 01303F00 le_filesize... 000007F0 le_data..... 013045C8 le_tid..... 00000000 le_datasize... 00000128 le_usecount... 00000003 le_loadcount... 00000001 le_ndepend... 00000001 le_maxdepend... 00000001 le_ule..... 0502E000 le_deferred... 00000000 le_exports... 0502E000 le_de..... 6C696263 le_searchlist.. B0000420 le_dlusecount.. 00000000 le_dlindex.... 00002F6C le_lex..... 00000000 le_fh..... 00000000 le_depend.... @ 04E17FD4 TOC@..... 013046D4</pre> <p style="text-align: center;"><PROCESS TRACE BACKS></p> <pre> .demokext 01304040 .close_log 013041FC .write_log 01304240 .open_log 013042B4 .strcpy 01304320 .sprintf.glink 01304428 .fp_close.glink 01304450 .strlen 01304480 .fp_write.glink 01304578 .fp_open.glink 013045A0</pre> <p>From the PROCESS TRACE BACKS we see that the first instruction of demokext is at 01304040. So the break for line 67 would be at this address plus E0.</p>
KDB(0)> b 01304040+e0	Set the break at the desired location. KDB responds with an indication of the address that the breakpoint is at.
KDB(0)> ca	Clear all breakpoints.
Set a breakpoint using Method 3	

Prompt and Console Input	Function and Example Output
KDB(0)> nm demokext	This translates a symbol to an effective address. The output from this subcommand will be similar to: Symbol Address : 01304040 TOC Address : 013046D4 The value of the symbol demokext is the address of the first instruction of the demokext routine. So this value can be used to set a breakpoint, just as in the previous example.
KDB(0)> b 01304040+e0	Set the break at the desired location. KDB responds with an indication of the address that the breakpoint is at.
KDB(0)> dw 01304040+e0	Display the word at the breakpoint. KDB will respond with something similar to: 01304120: 80830000 30840001 90830000 809F00300.....0 This can then be checked against the assembly code in the listing to verify that the break is set at the correct location.
KDB(0)> ca	Clear all breakpoints.
Set a breakpoint using Method 4	
KDB(0)> dw 1304748	Display the memory at the address returned as the kmid from the sysconfig routine at the beginning of this example. KDB responds with something similar to: demokext+000000: 01304040 01304754 00000000 01304648 .000.0GT.....0FH The first word of data displayed is the address of the first instruction of the <i>demokext</i> routine. Note, the data displayed is at the location demokext+000000. This corresponds to line 26 of the map presented earlier. However, the most important thing to note is that demokext+000000 and .demokext+000000 are not the same address. The location .demokext+000000 corresponds to line 10 of the map and is the address of the first instruction for the demokext routine.
KDB(0)> b 01304040+e0	Set the break at the location indicated from the previous command plus the offset to get to line 67. KDB responds with an indication of the address that the breakpoint is at.
ca	Clear all breakpoints.
Set a breakpoint using Method 5	
KDB(0)> devsw 1	Display the device switch table for the first entry. Note, the demonstration program that is being used is not a device driver; so this example just uses the addresses of the first device driver in the device switch table and is not related in any way to the demonstration program. The KDB devsw command displays data similar to: Slot address 50006040 MAJ#001 OPEN CLOSE READ WRITE .syopen .nulldev .syread .sywrite IOCTL STRATEGY TTY SELECT .syioc1 .nodev 00000000 .syselect CONFIG PRINT DUMP MPX .nodev .nodev .nodev .nodev REVOKE DSDPTR SELPTR OPTS .nodev 00000000 00000000 00000012
KDB(0)> b .syopen+20	Set a breakpoint at an offset of 0x20 from the beginning of the open routine for the first device driver in the device switch table. Note, KDB responds with an indication of where the break was set.
KDB(0)> ca	Clear all breakpoints.
KDB(0)> ns	Turn off symbolic name translation.

Prompt and Console Input	Function and Example Output
KDB(0)> devsw 1	Display the device switch table for the first device driver again. This time, with symbolic name translation turned off addresses instead of names will be displayed. The output from this subcommand is similar to: Slot address 50006040 MAJ#001 OPEN CLOSE READ WRITE 00208858 00059750 002086D4 0020854C IOCTL STRATEGY TTY SELECT 00208290 00059774 00000000 00208224 CONFIG PRINT DUMP MPX
KDB(0)> b 00208858+20	Set a break at an offset of 0x20 from the beginning of the open routine for the first device driver in the device switch table. This will set the same break that was set at the beginning of <i>Set a breakpoint using Method 5</i> . KDB responds with an indication of where the break was set.
KDB(0)> ns	Toggle symbolic name translation on.
KDB(0)> ca	Clear all breaks.
KDB(0)> g	Exit the KDB Kernel Debugger and let the system resume normal execution.
Unload the demokext kernel extension	
\$ fg	Bring the demo program to the foreground. Note, it will be waiting for user input of 0 to unload and exit, 1 to increment counters, or 2 decrement counters (the prompt will not be redisplayed, since it was shown prior to stopping the program and placing it in the background).
./demo 0	Enter a value of 0 to indicate that the kernel extension is to be unloaded and that the demo program is to terminate.

Viewing and Modifying Global Data

Global data may be accessed by several methods. In this section three methods are presented to access global data. The demo and demokext programs continue to be used in the illustrations in this section. In particular, the variable *demokext_j* (which is exported) is used in the examples.

The first method presented demonstrates the simplest method of access for global data. The second method presented demonstrates accessing global data using the TOC and the map file. This method requires that the system is stopped in the KDB Kernel Debugger within a procedure of the kernel extension to be debugged. Finally, the third method demonstrates a way to access global data using the map file, but without using the TOC.

- **Method 1**

Access of global variables within KDB is very simple. The variables may be accessed directly by name. For example to display the value of *demokext_j* the subcommand "*dw demokext_j*" may be used. If *demokext_j* was an array, a specific value could be viewed by adding the appropriate offset. For example, "*dw demokext_j+20*". Access to individual elements of a structure is accomplished by adding the proper offset to the base address for the variable.

- **Method 2**

To locate the address of global data using the address of the TOC and the map requires that the system be stopped in the KDB Kernel Debugger within a routine of the kernel extension to be debugged. This may be accomplished by setting a breakpoint within the kernel extension, as discussed in the previous section. When the KDB Kernel Debugger is invoked, general purpose register number 2 points to the address of the TOC. From the map file the offset from

the start of the TOC to the desired TOC entry may be calculated. Knowing this offset and the address at which the TOC starts allows the address of the TOC entry for the desired global variable to be calculated. Then the address of the TOC entry for the desired variable may be examined to determine the address of the data.

Example

As an example, assume that the KDB Kernel Debugger has been invoked because of a breakpoint at line 67 of the `demokext` routine and that the value for general purpose register number 2 is `0x01304754`. Then to find the address of the `demokext_j` data requires the following:

1. Calculate the offset from the beginning of the TOC to the TOC entry for `demokext_j`. From the map file, the TOC starts at `0x0000010C` and the TOC entry for `demokext_j` is at `0x00000114`. Therefore, the offset from the beginning of the TOC to the entry of interest is:
 $0x00000114 - 0x0000010C = 0x00000008$
2. Calculate the address of the TOC entry for `demokext_j`. This is the current value of general purpose register 2 plus the offset calculated in the preceding step:
 $0x01304754 + 0x00000008 = 0x0130475C$
3. Display the data at `0x0130475C`. The data displayed is the address of the data for `demokext_j`.

- **Method 3**

Unlike the procedure outlined in method 2, this method may be used at any time. This method requires the map file and the address at which the kernel extension has been loaded. Note, this method works because of the manner in which a kernel extension is loaded. Therefore, it may not work if the procedure for loading a kernel extension changes.

This method relies on the assumption that the address of a global variable may be found by using the formula:

$$\text{Addr of variable} = \text{Addr of the last function before the variable in the map} + \text{Length of the function} + \text{Offset of the variable}$$

To illustrate this calculation, refer to the following section of the map file for the `demokext` kernel extension.

```

20      000005B8 000028 2 GL SD S17 <.fp_write>          glink.s(/usr/lib/
                                           glink.o)
21      000005B8          GL LD S18 .fp_write
22      000005E0 000028 2 GL SD S19 <.fp_open>          glink.s(/usr/lib/
                                           glink.o)
23      000005E0          GL LD S20 .fp_open
24      00000000 0000F9 3 RW SD S21 <_STATIC>          demokext.c(demokext.o)
25      E 000000FC 000004 2 RW SD S22 demokext_j        demokext.c(demokext.o)
26      * 00000100 00000C 2 DS SD S23 demokext          demokext.c(demokext.o)
27      0000010C 000000 2 T0 SD S24 <TOC>
28      0000010C 000004 2 TC SD S25 <_STATIC>
29      00000110 000004 2 TC SD S26 <_system_configuration>

```

The last function in the `.text` section is at lines 22-23. The offset of this function from the map is `0x000005E0` (line 22, column 2). The length of the function is `0x000028` (Line 22, column 3). The offset of the `demokext_j` variable is `0x000000FC` (line 25, column 2). So the offset from the load point value to `demokext_j` is:

$$0x000005E0 + 0x000028 + 0x000000FC = 0x00000704$$

Adding this offset to the load point value of the demokext kernel extension yields the address of the data for *demokext_j*. Assuming a load point value of 0x01304040 (as used in previous examples), this would indicate that the data for *demokext_j* was located at:

$$0x01304040 + 0x00000704 = 0x01304744$$

Note that in Method 2, using the TOC, the address of the address of the data for *demokext_j* was calculated; while in Method 3 simply the address of the data for *demokext_j* was found. Also note that Method 1 is the primary method of access of global data when using the KDB Kernel Debugger. The other methods are mainly described to show alternatives and to allow the use of additional KDB subcommands in the following examples.

Prompt and Console Input	Function and Example Output
Load the demokext kernel extension	
\$./demo	Run the demo program, this loads the demokext extension.
\$ <CTRL-Z>	Stop the demo program.
\$ bg	Put the demo program in the background.
\$ <CTRL-\>	Activate KDB, use the appropriate key sequence for your configuration. You should have a KDB prompt on completion of this step.
Viewing/Modifying Global Data using Method 1	
KDB(0)> dw demokext_j	Display a word at the address of the <i>demokext_j</i> variable. Since the kernel extension was just loaded this variable should have a value of 99 and the KDB Kernel Debugger should display that value. The data displayed should be similar to the following: demokext_j+000000: 00000063 01304040 01304754 00000000 ...c.0@@.0GT....
KDB(0)> ns	Turn off symbolic name translation.
KDB(0)> dw demokext_j	This will again display the word at the address of the <i>demokext_j</i> variable, except with symbolic name translation turned off. So the data displayed should be similar to: 01304744: 00000063 01304040 01304754 00000000 ...c.0@@.0GT....
KDB(0)> ns	Turn symbolic name translation on.
KDB(0)> mw demokext_j	Modify the word at the address of the <i>demokext_j</i> variable. The KDB Kernel Debugger displays the current value of the word and waits for user input to change the value. The data displayed should be: 01304744: 00000063 = A new value may then be entered. After a new value is entered, the next word of memory is displayed for possible modification. To end memory modification a period (.) is entered. So to complete this step, enter a value of 64 (100 decimal) for the first address and then enter a period to end modification.
Viewing/Modifying Global Data using Method 2	

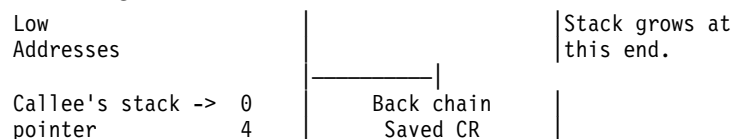
Prompt and Console Input	Function and Example Output
KDB(0)> b demokext+e0	Set a break at line 67 of the demokext routine (see the examples in the previous section). Breaking at this location will insure that the KDB Kernel Debugger is invoked while within the demokext routines. Then we can get the value of General Purpose Register 2, to determine the address of the TOC.
KDB(0)> g	Exit the KDB Kernel Debugger. This exits the debugger and we can then bring the demo program to the foreground and choose a selection to cause the demokext routine to be called for configuration. Since a break has been set this will cause the KDB Kernel Debugger to be invoked.
\$ fg	Bring the demo program to the foreground.
./demo 1	Enter a value of 1 to select the option to increment the counters within the demokext kernel extension. This causes a break at line 67 of demokext.
KDB(0)> dr	<p>Display the general purpose registers. The data displayed should be similar to the following:</p> <pre> r0 : 0130411C r1 : 2FF3B210 r2 : 01304754 r3 : 01304744 r4 : 0047B180 r5 : 0047B230 r6 : 000005FB r7 : 000DD300 r8 : 000005FB r9 : 000DD300 r10 : 00000000 r11 : 00000000 r12 : 013042F4 r13 : DEADBEEF r14 : 00000001 r15 : 2FF22D80 r16 : 2FF22D88 r17 : 00000000 r18 : DEADBEEF r19 : DEADBEEF r20 : DEADBEEF r21 : DEADBEEF r22 : DEADBEEF r23 : DEADBEEF r24 : 2FF3B6E0 r25 : 2FF3B400 r26 : 10000574 r27 : 22222484 r28 : E3001E30 r29 : E6001800 r30 : 01304744 r31 : 01304648 </pre> <p>Using the map, the offset to the TOC entry for <i>demokext_j</i> from the start of the TOC was 0x00000008 (see the above text concerning Method 2). Adding this offset to the value displayed for r2 indicates that the TOC entry of interest is at: 0x0130475C. Note, the KDB Kernel Debugger may be used to perform the addition. In this case the subcommand to use would be <i>hcal @r2+8</i>.</p>
KDB(0)> dw 0130475C	<p>Display the TOC entry for <i>demokext_j</i>. This entry will contain the address of the data for <i>demokext_j</i>. The data displayed should be similar to:</p> <pre> TOC+00000008: 01304744 000BCB34 00242E94 001E0518 .0GD...4.\$..... </pre> <p>The value for the first word displayed is the address of the data for the <i>demokext_j</i> variable.</p>

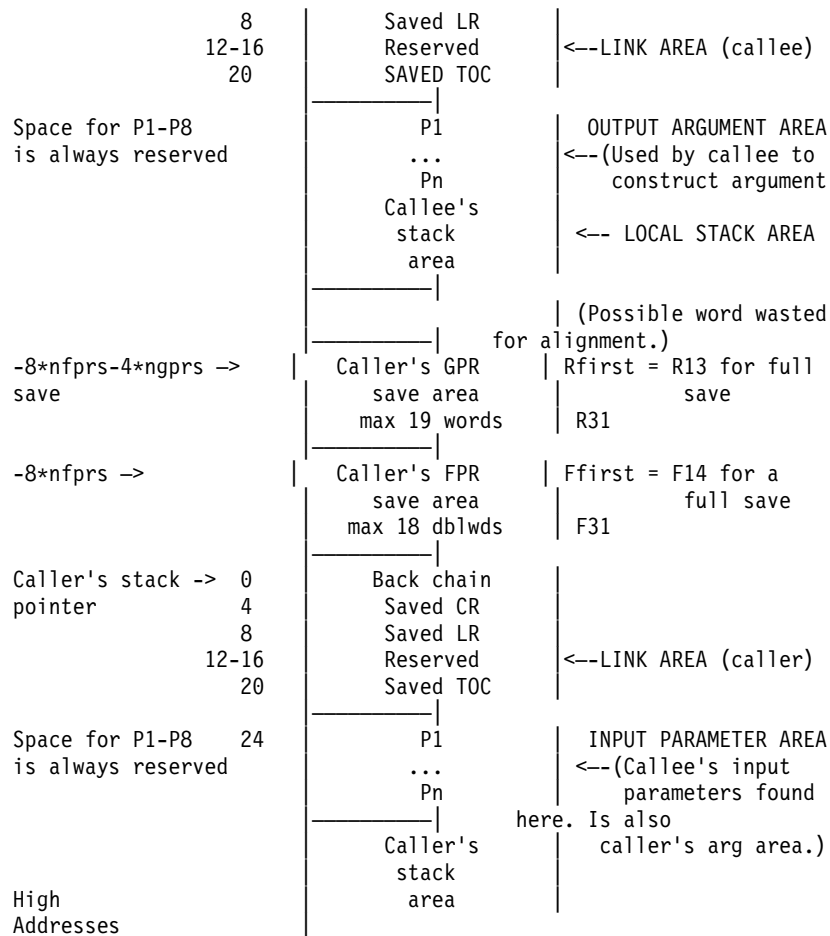
Prompt and Console Input	Function and Example Output
KDB(0)> dw 01304744	Display the data for <i>demokext_j</i> . The data displayed should indicate that the value for <i>demokext_j</i> is still 0x0000064, which we set it to earlier. This is because the breakpoint set was in the demokext routine prior to <i>demokext_j</i> being incremented. The data displayed should be similar to: demokext_j+000000: 00000064 01304040 01304754 00000000 ...d.0@@.0GT....
KDB(0)> ca	Clear all breakpoints.
KDB(0)> g	Exit the kernel debugger. Be careful here, when we exit, the demo program will still be in the foreground and there will be a prompt for the next option. Also note that the kernel extension is going to run and increment <i>demokext_j</i> ; so next time it should have a value of 0x0000065.
Enter choice: <CTRL-Z>	Enter <CTRL-Z> to stop the demo program.
\$ bg	Place the demo program in the background.
Viewing/Modifying Global Data using Method 3	
\$ <CTRL-\>	Activate KDB, use the appropriate key sequence for your configuration. You should have a KDB prompt on completion of this step.
KDB(0)> dw demokext+704	Display the data for the <i>demokext_j</i> variable. The 704 value is calculated from the map (refer to the above text for Method 3). This offset is then added to the load point of the demokext routine. Though there are numerous ways to find this address, in this case the simplest is to just use the symbolic name; to review other options refer back to the Setting Breakpoints section. As mentioned, earlier the value for <i>demokext_j</i> should now be 0x0000065. The data displayed should be similar to: demokext_j+000000: 00000065 01304040 01304754 00000000 ...e.0@@.0GT....
KDB(0)> g	Exit the KDB Kernel Debugger.
\$ fg	Bring the demo program to the foreground.
./demo 0	Enter an option of 0 to unload the demokext kernel extension and exit.

Stack Trace

The stack trace gives the stack history. This provides the sequence of procedure calls leading to the current IAR. The **Saved LR** is the address of the instruction calling this procedure. You can use the map file to locate the name of the procedure. Note that the first stack frame shown is almost always useless, since data either has not been saved yet, or is from a previous call. The last function preceding the **Saved LR** is the function that called the procedure.

The following is a concise view of the stack:





To illustrate some of the capabilities of the KDB Kernel Debugger for viewing the stack use the demo program and demokext kernel extension again. This time a break will be set in the `write_log` routine.

Prompt and Console Input	Function and Example Output
<i>Load the demokext kernel extension</i>	
\$./demo	Run the demo program, this loads the demokext extension.
\$ <CTRL-Z>	Stop the demo program.
\$ bg	Put the demo program in the background.
\$ <CTRL-\>	Activate KDB, use the appropriate key sequence for your configuration. You should have a KDB prompt on completion of this step.
Set and execute to a breakpoint in write_log	
KDB(0)> b demokext+280	Set a break at line 117 of demokext.c; this is the first line of write_log. The offset of 0x00000280 was determined from the list file as described in earlier sections.
KDB(0)> g	Exit the KDB Kernel Debugger.
\$ fg	Bring the demo program to the foreground.
./demo 1	Select option 1 to increment the counters in the kernel extension demokext. This causes the KDB Kernel Debugger to be invoked; stopped at the breakpoint set in write_log.

Prompt and Console Input	Function and Example Output
View the stack	
KDB(0)> stack	<p>This displays the stack for the current process, which was the the demo program calling the demokext kernel extension (since there was a break set). The stack trace back displays the routines called and even traces back through system calls. The displayed data should be similar to:</p> <pre> thread+001800 STACK: [013042C0]write_log+00001C (10002040, 2FF3B258, 2FF3B2BC) [013040B0]demokext+000070 (00000001, 2FF3B338) [001E3BF4]config_kmod+0000F0 (??, ??, ??) [001E3FA8]sysconfig+000140 (??, ??, ??) [000039D8].sys_call+000000 () [10000570]main+000280 (??, ??) [10000188]__start+000088 () </pre>
KDB(0)> s 4	<p>This subcommand steps 4 instructions. This should get into a strlen call. If it doesn't, continue stepping until strlen is entered.</p>
KDB(0)> stack	<p>Reexamine the stack. It should now include the strlen call and should be similar to:</p> <pre> thread+001800 STACK: [01304500]strlen+000000 () [013042CC]write_log+000028 (10002040, 2FF3B258, 2FF3B2BC) [013040B0]demokext+000070 (00000001, 2FF3B338) [001E3BF4]config_kmod+0000F0 (??, ??, ??) [001E3FA8]sysconfig+000140 (??, ??, ??) [000039D8].sys_call+000000 () [10000570]main+000280 (??, ??) [10000188]__start+000088 () </pre>
KDB(0)> set display_stack_frames	<p>This subcommand toggles a KDB Kernel Debugger option to display the top (lower addresses) 64 bytes for each stack frame.</p>

Prompt and Console Input	Function and Example Output
KDB(0)> stack	<p>Redisplay the stack with the display_stack_frames option turned on. The output should be similar to:</p> <pre> thread+001800 STACK: [01304510]strlen+000000 () ===== 2FF3B1C0: 2FF3 B210 2FF3 B380 0130 4364 0000 0000 /.../....0Cd.... 2FF3B1D0: 2FF3 B230 0130 4754 0023 AD5C 2222 2082 /..0.0GT.#.\"" . 2FF3B1E0: 0012 0000 2FF3 B400 0000 0480 0000 510C/.....Q. 2FF3B1F0: 2FF3 B260 4A22 2860 001D CEC8 0000 153C /..'J"('.....).... 2FF3B3E0: 2000 A1D8 0000 0000 0000 0000 0000 0000 2FF3B3F0: 0000 0000 0024 FA90 0000 0000 0000 0000\$. ===== [000039D8].sys_call+000000 () ===== 2FF21AA0: 2FF2 2D30 0000 0000 1000 0574 0000 0000 /..-0.....t.... 2FF21AB0: 0000 0000 2000 0B14 2000 08AC 2FF2 1AE0/... 2FF21AC0: 0000 000E F014 992D 6F69 6365 3A20 0000-oice: .. 2FF21AD0: FFFF FFFF D012 D1C0 0000 0000 0000 0000 ===== [10000570]main+000280 (??, ??) ===== 2FF22D30: 0000 0000 0000 0000 1000 018C 0000 0000 2FF22D40: 0000 0000 0000 0000 0000 0000 0000 0000 2FF22D50: 0000 0000 0000 0000 0000 0000 0000 0000 2FF22D60: 0000 0000 0000 0000 0000 0000 0000 0000 ===== [10000188]__start+000088 () </pre> <p>The displayed data can be interpreted using the diagram presented at the first of this section.</p>
KDB(0)> set display_stack_frames	Toggle the display_stack_frames option off.
KDB(0)> set display_stacked_regs	This subcommand toggles a KDB Kernel Debugger option to display the registers saved in each stack frame.

Prompt and Console Input	Function and Example Output
KDB(0)> stack	<p>Redisplay the stack with the display_stacked_regs option activated. The display should be similar to:</p> <pre> thread+001800 STACK: [01304510]strlen+000010 () [013042CC]write_log+000028 (10002040, 2FF3B258, 2FF3B2BC) r30 : 00000000 r31 : 01304648 [013040B0]demokext+000070 (00000001, 2FF3B338) r30 : 00000000 r31 : 00000000 [001E3BF4]config_kmod+0000F0 (??, ??, ??) r30 : 00000005 r31 : 2FF21AF8 [001E3FA8]sysconfig+000140 (??, ??, ??) r30 : 04DAE000 r31 : 00000000 [000039D8].sys_call+000000 () [10000570]main+000280 (??, ??) r25 : DEADBEEF r26 : DEADBEEF r27 : DEADBEEF r28 : DEADBEEF r29 : DEADBEEF r30 : DEADBEEF r31 : DEADBEEF [10000188]__start+000088 () </pre>
KDB(0)> set display_stacked_regs	Toggle the display_stacked_regs option off.
KDB(0)> dw @r1 90	<p>Display the stack in raw format. Note, the address for the stack is in general purpose register 1, so that may be used. The address could also have been obtained from the output when the display_stack_frames option was set. This subcommand displays 0x90 words of the stack in hex and ascii. The output should be similar to the following:</p> <pre> 2FF3B1C0: 2FF3B210 2FF3B380 01304364 00000000 ./.../....0Cd.... 2FF3B1D0: 2FF3B230 01304754 0023AD5C 22222082 /..0.0GT.#.\"" . 2FF3B1E0: 00120000 2FF3B400 00000480 0000510C .../.....Q. 2FF3B1F0: 2FF3B260 4A222860 001DCEC8 0000153C /..'J"('..... 2FF3B3E0: 2000A1D8 00000000 00000000 00000000 2FF3B3F0: 00000000 0024FA90 00000000 00000000\$. </pre> <p>This portion of the stack may be interpreted using the diagram at the beginning of this section.</p>
KDB(0)> ca	Clear all breakpoints.
KDB(0)> g	Exit the kernel debugger. Upon exiting the debugger the prompt from the demo program should be displayed.
Enter choice: 0	Enter an choice of 0 to unload the kernel extension and quit.

demo.c Example File

```

#include <sys/types.h>
#include <sys/sysconfig.h>
#include <memory.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>
#include "demo.h"
/* Extension loading data */

```



```

struct cfg_load cfg_load;
extern int sysconfig();
extern int errno;
#define NAME_SIZE 256
#define LIBPATH_SIZE 256
main(argc,argv)
int argc;
char *argv[];
{
char path[NAME_SIZE];
char libpath[LIBPATH_SIZE];
char buf[BUFLLEN];
struct cfg_kmod cfg_kmod;
struct extparms extparms = {argc,argv,buf,BUFLLEN};
int option = 1;
int status = 0;
/*
 * Load the demo kernel extension.
 */
memset(path, 0, sizeof(path));
memset(libpath, 0, sizeof(libpath));
strcpy(path, "./demokext");
cfg_load.path = path;
cfg_load.libpath = libpath;
if (sysconfig(SYS_KLOAD, &cfg_load, sizeof(cfg_load)) == CONF_SUCC)
{
printf("Kernel extension ./demokext was succesfully loaded, kmid=%x\n",
      cfg_load.kmid);
}
else
{
printf("Encountered errno=%d loading kernel extension %s\n",
      errno, cfg_load.path);
exit(1);
}
/*
 * Loop alterantely allocating and freeing 16K from memory.
 */
option = 1;
while (option != 0)
{
printf("\n\n");
printf("0. Quit and unload kernel extension\n");
printf("1. Configure kernel extension - increment counter\n");
printf("2. Configure kernel extension - decrement counter\n");
printf("\n");
printf("Enter choice: ");
scanf("%d", &option);
switch (option)
{
case 0:
break;
case 1:
bzero(buf,BUFLLEN);
strcpy(buf,"sample string");
cfg_kmod.kmid = cfg_load.kmid;
cfg_kmod.cmd = 1;
cfg_kmod.mdiptr = (char *)&extparms;
cfg_kmod.mdilen = sizeof(extparms);
if (sysconfig(SYS_CFGKMOD,&cfg_kmod, sizeof(cfg_kmod))==CONF_SUCC)
{
printf("Kernel extension %s was successfully configured\n",
      cfg_load.path);
}
}
else
{
printf("errno=%d configuring kernel extension %s\n",

```

```

        errno, cfg_load.path);
    }
    break;
    case 2:
        bzero(buf, BUFLLEN);
        strcpy(buf, "sample string");
        cfg_kmod.kmid = cfg_load.kmid;
        cfg_kmod.cmd = 2;
        cfg_kmod.mdiptr = (char *)&extparms;
        cfg_kmod.mdilen = sizeof(extparms);
        if (sysconfig(SYS_CFGKMOD, &cfg_kmod, sizeof(cfg_kmod)) == CONF_SUCC)
        {
            printf("Kernel extension %s was successfully configured\n",
                cfg_load.path);
        }
        else
        {
            printf("errno=%d configuring kernel extension %s\n",
                errno, cfg_load.path);
        }
        break;
    default:
        printf("\nUnknown option\n");
        break;
    }
}
/*
 * Unload the demo kernel extension.
 */
if (sysconfig(SYS_KULOAD, &cfg_load, sizeof(cfg_load)) == CONF_SUCC)
{
    printf("Kernel extension %s was successfully unloaded\n", cfg_load.path);
}
else
{
    printf("errno=%d unloading kernel extension %s\n", errno, cfg_load.path);
}
}

```

demokext.c Example File

```

#include <sys/types.h>
#include <sys/malloc.h>
#include <sys/uio.h>
#include <sys/dump.h>
#include <sys/errno.h>
#include <sys/unistd.h>
#include <fcntl.h>
#include "demo.h"
/* Log routine prototypes */
int open_log(char *path, struct file **fpp);
int write_log(struct file *fpp, char *buf, int *bytes_written);
int close_log(struct file *fpp);
/* Unexported symbol */
int demokext_i = 9;
/* Exported symbol */
int demokext_j = 99;
/*
 * Kernel extension entry point, called at config. time.
 *
 * input:
 * cmd - unused (typically 1=config, 2=unconfig)
 * uiop - points to the uio structure.
 */
int
demokext(int cmd, struct uio *uiop)
{

```

```

int rc;
char *bufp;
struct file *fpp;
int fstat;
char buf[100];
int bytes_written;
static int j = 0;
/*
 * Open the log file.
 */
strcpy(buf, "./demokext.log");
fstat = open_log(buf, &fpp);
if (fstat != 0) return(fstat);
/*
 * Put a message out to the log file.
 */
strcpy(buf, "demokext was called for configuration\n");
fstat = write_log(fpp, buf, &bytes_written);
if (fstat != 0) return(fstat);
/*
 * Increment or decrement j and demokext_j based on
 * the input value for cmd.
 */
{
switch (cmd)
{
case 1: /* Increment */
sprintf(buf, "Before increment: j=%d demokext_j=%d\n",
        j, demokext_j);
write_log(fpp, buf, &bytes_written);
demokext_j++;
j++;
sprintf(buf, "After increment: j=%d demokext_j=%d\n",
        j, demokext_j);
write_log(fpp, buf, &bytes_written);
break;
case 2: /* Decrement */
sprintf(buf, "Before decrement: j=%d demokext_j=%d\n",
        j, demokext_j);
write_log(fpp, buf, &bytes_written);
demokext_j--;
j--;
sprintf(buf, "After decrement: j=%d demokext_j=%d\n",
        j, demokext_j);
write_log(fpp, buf, &bytes_written);
break;
default: /* Unknown command value */
sprintf(buf, "Received unknown command of %d\n", cmd);
write_log(fpp, buf, &bytes_written);
break;
}
}
/*
 * Close the log file.
 */
fstat = close_log(fpp);
if (fstat != 0) return(fstat);
return(0);
}
/*****
 * Routines for logging debug information:      *
 * open_log - Opens a log file                  *
 * write_log - Output a string to a log file   *
 * close_log - Close a log file                *
 *****/
int open_log (char *path, struct file **fpp)
{

```

```

int rc;
rc = fp_open(path, O_CREAT | O_APPEND | O_WRONLY,
             S_IRUSR | S_IWUSR, 0, SYS_ADSPACE, fpp);
return(rc);
}
int write_log(struct file *fpp, char *buf, int *bytes_written)
{
int rc;
rc = fp_write(fpp, buf, strlen(buf), 0, SYS_ADSPACE, bytes_written);
return(rc);
}
int close_log(struct file *fpp)
{
int rc;
rc = fp_close(fpp);
return(rc);
}

```

demo.h Example File

```

#ifndef _demo
#define _demo
/*
 * Parameter structure
 */
struct extparms {
int argc;
char **argv;
char *buf; /* Message buffer */
size_t len; /* length */
};
#define BUFLen 4096 /* Test msg buffer length */
#endif /* _demo */

```

demokext.exp Example File

```

#!/unix
* export value from demokext
demokext_j

```

comp_link Example File

```

#!/bin/ksh
# Script to build the demo executable and the demokext kernel extension.
cc -o demo demo.c
cc -c -DEBUG -D_KERNEL -DIBMR2 demokext.c -qsource -qlist
ld -o demokext demokext.o -edemokext -bimport:/lib/syscalls.exp
    -bimport:/lib/kernex.exp -lcsys -bexport:demokext.exp -bmap:demokext.map

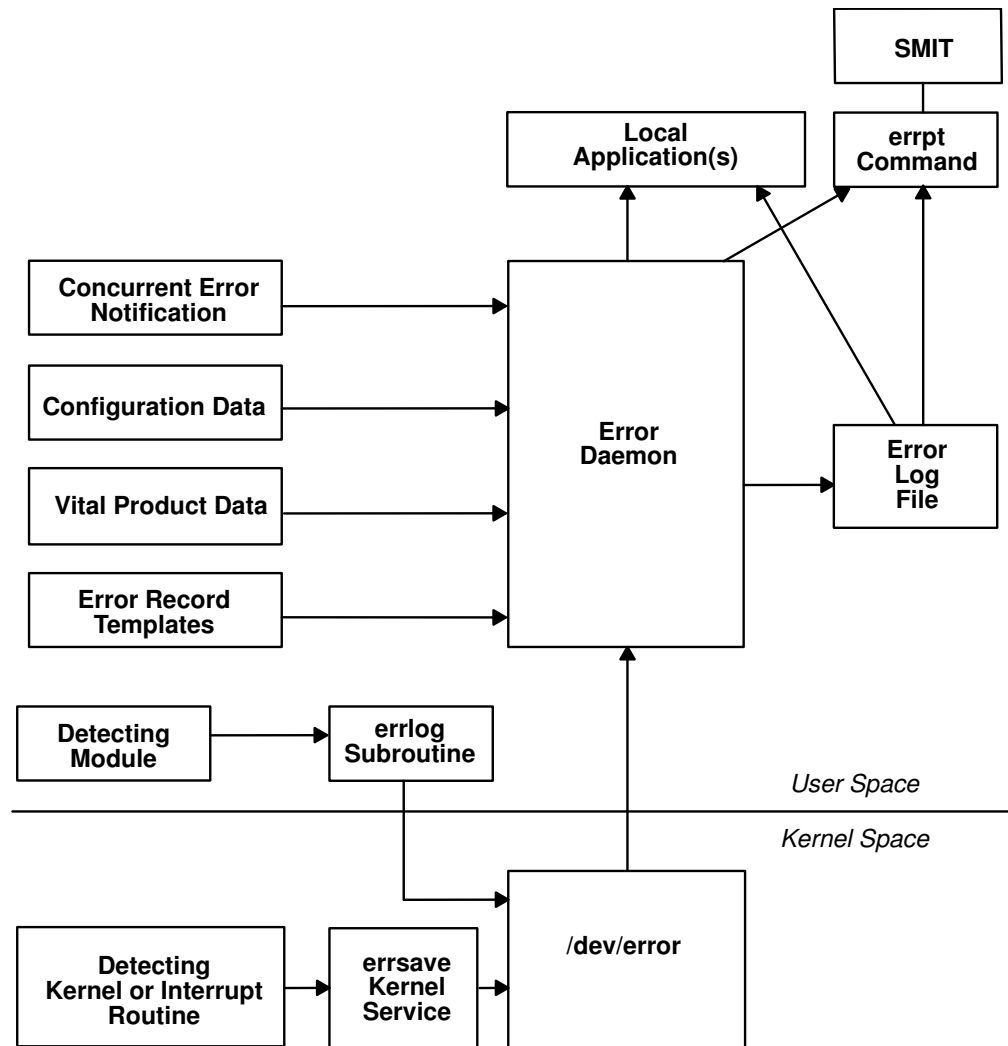
```

Error Logging

The error facility allows a device driver to have entries recorded in the system error log. These error log entries record any software or hardware failures that need to be available either for informational purposes or for fault detection and corrective action. The device driver, using the **errsave** kernel service, adds error records to the special file **/dev/error**.

The **errdemon** daemon then picks up the error record and creates an error log entry. When you access the error log either through SMIT (System Management Interface Tool) or with the **errpt** command, the error record is formatted according to the error template in the error template repository and presented in either a

summary or detailed report. See the "Flow of the Error Logging Facility" figure for an illustration of this.



Flow of the Error Logging Facility

Preceding Steps to Consider

Follow three preceding steps before initiating the error logging process. It is beneficial to understand what services are available to developers, and what the customer, service personnel, and defect personnel see.

Determine the Importance of the Error

The first preceding step is to review the error-logging documentation and determine whether a particular error should be logged. Do not use system resources for logging information that is unimportant or confusing to the intended audience.

It is, however, a worse mistake *not* to log an error that merits logging. You should work in concert with the hardware developer, if possible, to identify detectable errors and the information that should be relayed concerning those errors.

Determine the Text of the Message

The next step is to determine the text of the message. Use the **errmsg** command with the **-w** flag to browse the system error messages file for a list of available messages. If you are developing a product for wide-spread general distribution and do not find a suitable system error message, you can submit a request to your supplier for a new message or follow the procedures that your organization uses to request new error messages. If your product is an in-house application, you can use the **errmsg** command to define a new message that meets your requirements.

Determine the Correct Level of Thresholding

Finally, determine the correct level of thresholding. Each error to be logged, regardless of whether it is a software or hardware error, can be limited by thresholding to avoid filling the error log with duplicate information.

Side effects of runaway error logging include overwriting existing error log entries and unduly alarming the end user. The error log is not unlimited in size. When its size limit is reached, the log wraps. If a particular error is repeated needlessly, existing information is overwritten, possibly causing inaccurate diagnostic analyses. The end user or service person can perceive a situation as more serious or pervasive than it is if they see hundreds of identical or nearly identical error entries.

You are responsible for implementing the proper level of thresholding in the device driver code.

The error log currently equals 1MB. As shipped, it cleans up any entries older than 30 days. In order to ensure that your error log entries are actually informative, noticed, and remain intact, *test your driver thoroughly*.

Coding Steps

To begin error logging,

1. Select the error text.
2. Construct error record templates.
3. Add error logging calls into the device driver code.

Selecting the Error Text

The first task is to select the error text. After browsing the contents of the system message file, three possible paths exist for selecting the error text. Either all of the desired messages for the new errors exist in the message file, none of the messages exist, or a combination of errors exists.

- If the messages required already exist in the system message file, make a note of the four-digit hexadecimal identification number, as well as the message-set identification letter. For instance, a desired error description can be:

```
SET E
E859 "The wagon wheel is broken."
```

- If none of the system error messages meet your requirements, and if you are responsible for developing a product for wide-spread general distribution, you can either contact your supplier to allocate new messages or follow the procedures that your organization uses to request new messages. If you are creating an in-house product, use the **errmsg** command to write suitable error messages and use the **errinstall** command to install them. Refer to "Software Product Packaging" in *AIX Version 4.3 General Programming Concepts: Writing and Debugging Programs* for more information. Take care not to overwrite other error messages.

- It is also possible to use a combination of existing messages and new messages within the same error record template definition.

Constructing Error Record Templates

The second step is to construct your *error record templates*. An error record template defines the text that appears in the error report. Each error record template has the following general form:

```
Error Record Template
+LABEL:
    Comment =
    Class =
    Log =
    Report =
    Alert =
    Err_Type =
    Err_Desc =
    Probable_Causes =
    User_Causes =
    User_Actions =
    Inst_Causes =
    Inst_Actions =
    Fail_Causes =
    Fail_Actions =
    Detail_Data = <data_len>, <data_id>, <data_encoding>
```

Each field in this stanza has well-defined criteria for input values. See the **errupdate** command for more information. The fields are:

Label	Requires a unique label for each entry to be added. The label must follow C language rules for identifiers and must not exceed 16 characters in length.
Comment	Indicates this is a comment field. You must enclose the comment in double quotation marks; and it cannot exceed 40 characters.
Class	Requires class values of H (hardware), S (software), or U (Undetermined).
Log	Requires values True or False. If failure occurs, the errors are logged only if this field value is set to True. When this value is False the Report and Alert fields are ignored.
Report	The values for this field are True or False. If the logged error is to be displayed using error report, the value of this field must be True.
Alert	Set this field to True for errors that are alertable. For errors that are not alertable, set this field to False.

Err_Type	<p>Describes the severity of the failure that occurred. Possible values are INFO, PEND, PERF, PERM, TEMP, and UNKN where:</p> <p>INFO The error log entry is informational and was not the result of an error.</p> <p>PEND A condition in which it is determined that the loss of availability of a device or component is imminent.</p> <p>PERF A condition in which the performance of a device or component was degraded below an acceptable level.</p> <p>PERM A permanent failure is defined as a condition that was not recoverable. For example, an operation was retried a prescribed number of times without success.</p> <p>TEMP Recovery from this temporary failure was successful, yet the number of unsuccessful recovery attempts exceeded a predetermined threshold.</p> <p>UNKN A condition in which it is not possible to assess the severity of a failure.</p>
Err_Desc	Describes the failure that occurred. Proper input for this field is the four-digit hexadecimal identifier of the error description message to be displayed from SET E in the message file.
Prob_Causes	Describes one or more probable causes for the failure that occurred. You can specify a list of up to four Prob_Causes identifiers separated by commas. A Prob_Causes identifier displays a probable cause text message from SET P in the message file. List probable causes in the order of decreasing probability. At least one probable cause identifier is required.
User_Causes	Specifies a condition that an operator can resolve without contacting any service organization. You can specify a list of up to four User_Causes identifiers separated by commas. A User_Causes identifier displays a text message from SET U in the message file. List user causes in the order of decreasing probability. Leave this field blank if it does not apply to the failure that occurred. If this field is blank, either the Inst_Causes or the Fail_Causes field must not be blank.
User_Actions	<p>Describes recommended actions for correcting a failure that resulted from a user cause. You can specify a list of up to four recommended User_Actions identifiers separated by commas. A recommended User_Actions identifier displays a recommended action text message, SET R in the message file. You must leave this field blank if the User_Causes field is blank.</p> <p>The order in which the recommended actions are listed is determined by the expense of the action and the probability that the action corrects the failure. Actions that have little or no cost and little or no impact on system operation should always be listed first. When actions for which the probability of correcting the failure is equal or nearly equal, list the least expensive action first. List remaining actions in order of decreasing probability.</p>
Inst_Causes	Describes a condition that resulted from the initial installation or setup of a resource. You can specify a list of up to four Inst_Causes identifiers separated by commas. An Inst_Causes identifier displays a text message, SET I in the message file. List the install causes in the order of decreasing probability. Leave this field blank if it is not applicable to the failure that occurred. If this field is blank, either the User_Causes or the Failure_Causes field must not be blank.

Inst_Actions	Describes recommended actions for correcting a failure that resulted from an install cause. You can specify a list of up to four recommended Inst_actions identifiers separated by commas. A recommended Inst_actions identifier identifies a recommended action text message, SET R in the message file. Leave this field blank if the Inst_Causes field is blank. The order in which the recommended actions are listed is determined by the expense of the action and the probability that the action corrects the failure. See the User_Actions field for the list criteria.
Fail_Causes	Describes a condition that resulted from the failure of a resource. You can specify a list of up to four Fail_Causes identifiers separated by commas. A Fail_Causes identifier displays a failure cause text message, SET F in the message file. List the failure causes in the order of decreasing probability. Leave this field blank if it is not applicable to the failure that occurred. If you leave this field blank, either the User_Causes or the Inst_Causes field must not be blank.
Fail_Actions	Describes recommended actions for correcting a failure that resulted from a failure cause. You can specify a list of up to four recommended action identifiers separated by commas. The Fail_Actions identifiers must correspond to recommended action messages found in SET R of the message file. Leave this field blank if the Fail_Causes field is blank. Refer to the description of the User_Actions field for criteria in listing these recommended actions.
Detail_Data	Describes the detailed data that is logged with the error when the failure occurs. The Detail_data field includes the name of the detecting module, sense data, or return codes. Leave this field blank if no detailed data is logged with the error.

You can repeat the Detail_Data field. The amount of data logged with an error must not exceed the maximum error record length defined in the `sys/err_rec.h` header file. Save failure data that cannot be contained in an error log entry elsewhere, for example in a file. The detailed data in the error log entry contains information that can be used to correlate the failure data to the error log entry. Three values are required for each detail data entry:

data_len

Indicates the number of bytes of data to be associated with the **data_id** value. The **data_len** value is interpreted as a decimal value.

data_id Identifies a text message to be printed in the error report in front of the detailed data. These identifiers refer to messages in SET D of the message file.

data_encoding

Describes how the detailed data is to be printed in the error report. Valid values for this field are:

ALPHA

The detailed data is a printable ASCII character string.

DEC

The detailed data is the binary representation of an integer value, the decimal equivalent is to be printed.

HEX

The detailed data is to be printed in hexadecimal.

Sample Error Record Template

An example of an error record template is:

```
+ MISC_ERR:
  Comment = "Interrupt: I/O bus timeout or channel check"
  Class = H
```

```

Log = TRUE
Report = TRUE
Alert = FALSE
Err_Type = UNKN
Err_Desc = E856
Prob_Causes = 3300, 6300
User_Causes =
User_Actions =
Inst_Causes =
Inst_Actions =
Fail_Causes = 3300, 6300
Fail_Actions = 0000
Detail_Data = 4, 8119, HEX      *IOCC bus number
Detail_Data = 4, 811A, HEX     *Bus Status Register
Detail_Data = 4, 811B, HEX     *Misc. Interrupt Register

```

Construct the error templates for all new errors to be added in a file suitable for entry with the **errupdate** command. Run the **errupdate** command with the **-h** flag and the input file. The new errors are now part of the error record template repository. A new header file is also created (**file.h**) in the same directory in which the **errupdate** command was run. This header file must be included in the device driver code at compile time. Note that the **errupdate** command has a built-in syntax checker for the new stanza that can be called with the **-c** flag.

Adding Error Logging Calls into the Code

The third step in coding error logging is to put the error logging calls into the device driver code. The **errsave** kernel service allows the kernel and kernel extensions to write to the error log. Typically, you define a routine in the device driver that can be called by other device driver routines when a loggable error is encountered. This function takes the data passed to it, puts it into the proper structure and calls the **errsave** kernel service. The syntax for the **errsave** kernel service is:

```

#include <sys/errids.h>
void errsave(buf, cnt)
char *buf;
unsigned int cnt;

```

where,

- buf** Specifies a pointer to a buffer that contains an error record as described in the **sys/errids.h** header file.
- cnt** Specifies a number of bytes in the error record contained in the buffer pointed to by the *buf* parameter.

The following sample code is an example of a device driver error logging routine. This routine takes data passed to it from some part of the main body of the device driver. This code simply fills in the structure with the pertinent information, then passes it on using the **errsave** kernel service.

```

void
errsv_ex (int err_id, unsigned int port_num,
          int line, char *file, uint data1, uint data2)
{
    dderr    log;
    char     errbuf[255];
    ddex_dds *p_dds;
    p_dds = dds_dir[port_num];
    log.err.error_id = err_id;
    if (port_num == BAD_STATE) {
        sprintf(log.err.resource_name, "%s :%d",

```

```

        p_dds->dds_vpd.adpt_name, data1);
    data1 = 0;
}
else
    sprintf(log.err.resource_name,"%s",p_dds->dds_vpd.devname);
    sprintf(errbuf, "line: %d file: %s", line, file);
    strncpy(log.file, errbuf, (size_t)sizeof(log.file));
    log.data1 = data1;
    log.data2 = data2;
    errsava(&log, (uint)sizeof(dderr)); /* run actual logging */
} /* end errlog_ex */

```

The data to be passed to the **errsava** kernel service is defined in the **dderr** structure which is defined in a local header file, **dderr.h**. The definition for **dderr** is:

```

typedef struct dderr {
    struct err_rec0 err;
    int data1; /* use data1 and data2 to show detail */
    int data2; /* data in the errlog report. Define */
              /* these fields in the errlog template */
              /* These fields may not be used in all */
              /* cases. */
} dderr;

```

The first field of the **dderr.h** header file is comprised of the **err_rec0** structure, which is defined in the **sys/err_rec.h** header file. This structure contains the ID (or label) and a field for the resource name. The two data fields hold the detail data for the error log report. As an alternative, you could simply list the fields within the function.

You can also log a message into the error log from the command line. To do this, use the **errlogger** command.

After you add the templates using the **errupdate** command, compile the device driver code along with the new header file. Simulate the error and verify that it was written to the error log correctly. Some details to check for include:

- Is the error demon running? This can be verified by running the **ps -ef** command and checking for **/usr/lib/errdemon** as part of the output.
- Is the error part of the error template repository? Verify this by running the **errpt -at** command.
- Was the new header file, which was created by the **errupdate** command and which contains the error label and unique error identification number, included in the device driver code when it was compiled?

Writing to the **/dev/error** Special File

The error logging process begins when a loggable error is encountered and the device driver error logging subroutine sends the error information to the **errsava** kernel service. The error entry is written to the **/dev/error** special file. Once the information arrives at this file, it is time-stamped by the **errdemon** daemon and put in a buffer. The **errdemon** daemon constantly checks the **/dev/error** special file for new entries, and when new data is written, the daemon collects other information pertaining to the resource reporting the error. The **errdemon** daemon then creates an entry in the **/var/adm/ras/errlog** error logging file.

Debug and Performance Tracing

The AIX **trace** facility is useful for observing a running device driver and system. The **trace** facility captures a sequential flow of time-stamped system events, providing a fine level of detail on system activity. Events are shown in time sequence and in the context of other events. The **trace** facility is useful in expanding the trace event information to understand who, when, how, and even why the event happened.

Introduction

The operating system is shipped with permanent trace event points. These events provide general visibility to system execution. You can extend the visibility into applications by inserting additional events and providing formatting rules.

Care was taken in the design and implementation of this facility to make the collection of **trace** data efficient, so that system performance and flow would be minimally altered by activating **trace**. Because of this, the facility is extremely useful as a performance analysis tool and as a problem determination tool.

The **trace** facility is more flexible than traditional system monitor services that access and present statistics maintained by the system. With traditional monitor services, data reduction (conversion of system events to statistics) is largely coupled to the system instrumentation. For example, the system can maintain the minimum, maximum, and average elapsed time observed for runs of a task and permit this information to be extracted.

The **trace** facility does not strongly couple data reduction to instrumentation, but provides a stream of system events. It is not required to presuppose what statistics are needed. The statistics or data reduction are to a large degree separated from the instrumentation.

You can choose to develop the minimum, maximum, and average time for task A from the flow of events. But it is also possible to extract the average time for task A when called by process B, extract the average time for task A when conditions XYZ are met, develop a standard deviation for task A, or even decide that some other task, recognized by a stream of events, is more meaningful to summarize. This flexibility is invaluable for diagnosing performance or functional problems.

The **trace** facility generates large volumes of data. This data cannot be captured for extended periods of time without overflowing the storage device. This allows two practical ways that the **trace** facility can be used natively.

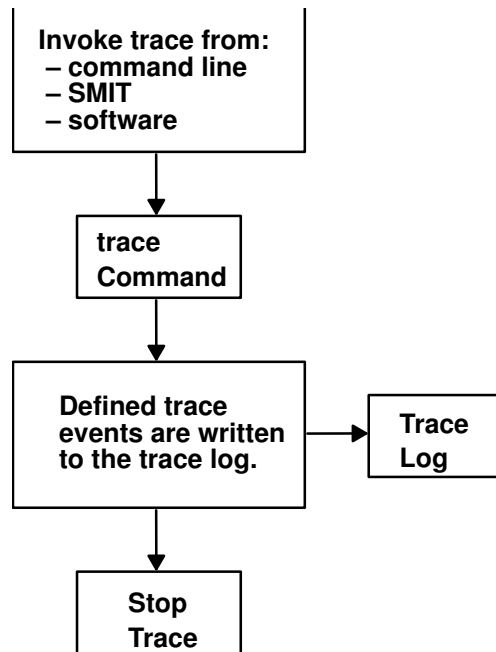
First, the **trace** facility can be triggered in multiple ways to capture small increments of system activity. It is practical to capture seconds to minutes of system activity in this way for post-processing. This is sufficient time to characterize major application transactions or interesting sections of a long task.

Second, the **trace** facility can be configured to direct the event stream to standard output. This allows a realtime process to connect to the event stream and provide data reduction in real-time, thereby creating a long term monitoring capability. A logical extension for specialized instrumentation is to direct the data stream to an auxiliary device that can either store massive amounts of data or provide dynamic data reduction.

You can start the system trace from:

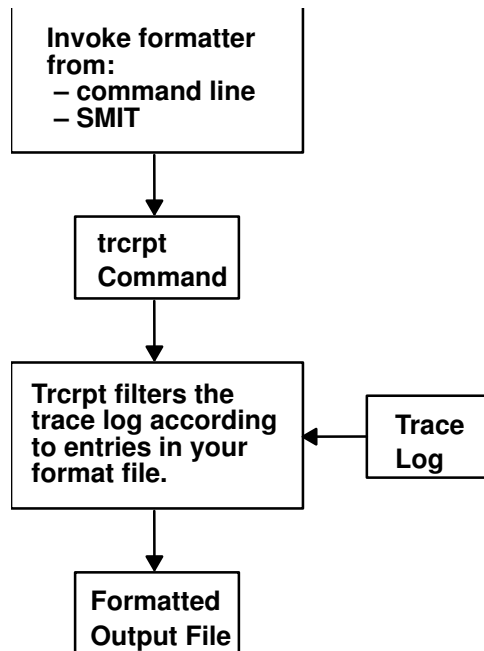
- The command line
- SMIT
- Software

As shown in the "Starting and Stopping Trace" figure, the trace facility causes predefined events to be written to a trace log. The tracing action is then stopped. Tracing from a command line is discussed in "Controlling trace" on page 559. Tracing from a software application is discussed and an example is presented in "Examples of Coding Events and Formatting Events" on page 576.



Starting and Stopping Trace

After a trace is started and stopped, you must format it before viewing it. This is illustrated in the "Trace Formatting" figure. To format the trace events that you have defined, you must provide a stanza that describes how the trace formatter is to interpret the data that has been collected. This is described in "Syntax for Stanzas in the trace Format File" on page 568.



Trace Formatting

The **trcrpt** command provides a general purpose report facility. The report facility provides little data reduction, but converts the raw binary event stream to a readable ASCII listing of the event stream. Data can be visually extracted by a reader, or tools can be developed to further reduce the data.

For an event to be traced, you must write an *event hook* (sometimes called a *trace hook*) into the code that you want to trace. Tracing can be done on either the system channel (channel 0) or on a generic channel (channels 1-7). All preshipped trace points are output to the system channel.

Usually, when you want to show interaction with other system routines, use the system channel. The generic channels are provided so that you can control how much data is written to the trace log. Only your data is written to one of the generic channels.

For more information on trace hooks, see “Macros for Recording trace Events” on page 565.

Using the trace Facility

The following sections describe the use of the **trace** facility.

Configuring and Starting trace Data Collection

The **trace** command configures the trace facility and starts data collection. The syntax of this command is:

```

trace [ -a | -f | -l ] [ -c ] [ -d ] [ -h ] [ -j Event [ ,Event ] ]
[ -k Event [ ,Event ] ] [ -m Message ] [ -n ] [ -o OutName ] [ -o- ] [ -s ]
[ -L Size ] [ -T Size ] [-1234567]
  
```

The various options of the **trace** command are:

- f or -l** Control the capture of trace data in system memory. If you specify neither the **-f** nor **-l** option, the trace facility creates two buffer areas in system memory to capture the trace data. These buffers are alternately written to the log file (or standard output if specified) as they become full. The **-f** or **-l** flag provides you with the ability to prevent data from being written to the file during data collection. The options are to collect data only until the memory buffer becomes full (**-f** for first), or to use the memory buffer as a circular buffer that captures only the last set of events that occurred before **trace** was terminated (**-l**). The **-f** and **-l** options are mutually exclusive. With either the **-f** or **-l** option, data is not transferred from the memory collection buffers to file until **trace** is terminated.
- a** Run the **trace** collection asynchronously (as a background task), returning a normal command line prompt. Without this option, the **trace** command runs in a subcommand mode (similar to the **crash** command) and returns a **>** prompt. You can issue subcommands and regular shell commands from the **trace** subcommand mode by preceding the shell commands with an **!** (exclamation point).
- c** Saves the previous trace log file adding **.old** to its name. Generates an error if a previous trace log file does not exist. When using the **-o Name** flag, the user-defined trace log file is renamed.
- d** Delay data collection. The trace facility is only configured. Data collection is delayed until one of the collection trigger events occurs. Various methods for triggering data collection on and off are provided. These include the following:
- **trace** subcommands
 - **trace** commands
 - **ioctl**s to **/dev/systrctl**
- j Event or -k Event** Specifies a set of events to include (**-j**) or exclude (**-k**) from the collection process. The *Event* list items can be separated by commas, or enclosed in double quotation marks and separated by commas or blanks.
- s** Terminate **trace** data collection if the **trace** log file reaches its maximum specified size. The default without this option is to wrap and overwrite the data in the log file on a FIFO basis.
- h** Do not write a **date/sysname/message** header to the **trace** log file.
- m Message** Specify a text string (message) to be included in the **trace** log header record. The message is included in reports generated by the **trcrpt** command.
- n** Adds some information to the trace log header: lock information, hardware information, and, for each loader entry, the symbol name, address, and type.
- o Outfile** Specify a file to use as the log file. If you do not use the **-o** option, the default log file is **/usr/adm/ras/trcfile**. To direct the trace data to standard output, code the **-o** option as **-o -**. (When **-o-** is specified the **-c** flag is ignored.) Use this technique only to pipe the data stream to another process since the trace data contains raw binary events that are not displayable.

-1234567

Duplicate the **trace** design for multiple channels. Channel 0 is the default channel and is always used for recording system events. The other channels are generic channels, and their use is not predefined. There are various uses of generic channels in the system. The generic channels are also available to user applications. Each created channel is a separate events data stream. Events recorded to channel 0 are mixed with the predefined system events data stream. The other channels have no predefined use and are assigned generically.

A program can request that a generic channel be opened by using the **trcstart** subroutine. A channel number is returned, similar to the way a file descriptor is returned when it opens a file. The program can record events to this channel and, thus, have a private data stream. The **trace** command allows a generic channel to be specifically configured by defining the channel number with this option. However, this is not generally the way a generic channel is started. It is more likely to be started from a program using the **trcstart** subroutine, which uses the returned channel ID to record events.

-T Size and **-L Size**

Specify the size of the collection memory buffers and the maximum size of the log file in bytes. The trace facility pins the data collection buffers, making this amount of memory unavailable to the rest of the system. It is important to be aware of this, because it means that the trace facility can impact performance in a memory constrained environment. If the application being monitored is not memory constrained, or if the percentage of memory consumed by the trace routine is small compared to what is available in the system, the impact of **trace** "stolen" memory should be small.

If you do not specify a value, trace uses a default size. The trace facility pins a little more than the specified buffer size. This additional memory is required for the trace facility itself. Trace pins a little more than the amount specified for first buffer mode (**-f** option). Trace pins a little more than twice the amount specified for trace configured in alternate buffer or last (circular) buffer mode.

You can also start **trace** from a command line or with a **trcstart** subroutine call. The **trcstart** subroutine is in the **librts.a** library. The syntax of the **trcstart** subroutine is:

```
int trcstart(char *args)
```

where *args* is simply the options list desired that you would enter using the trace command if starting a system trace (channel 0). If starting a generic trace, include a **-g** option in the *args* string. On successful completion, **trcstart** returns the channel ID. For generic tracing this channel ID can be used to record to the private generic channel.

For an example of the **trcstart** routine, see the sample code on page 560.

When compiling a program using this subroutine, you must request the link to the **librts.a** library. Use **-l rts** as a compile option.

Controlling trace

Once **trace** is configured by the **trace** command or the **trcstart** subroutine, controls to **trace** trigger the collection of data on, trigger the collection of data off, and stop the trace facility (stop deconfigures **trace** and unpins buffers). These basic controls exist as subcommands, commands, subroutines, and ioctl controls to the **trace** control device, **/dev/systrctl**. These controls are described in the following sections.

Controlling trace in Subcommand Mode

If the **trace** routine is configured without the **-a** option, it runs in subcommand mode. Instead of the normal shell prompt, **->** is the prompt. In this mode the following subcommands are recognized:

trcon	Triggers collection of trace data on.
trcoff	Triggers collection of trace data off.
q or quit	Stops collection of trace data (like trcoff) and terminates trace (deconfigures).
!command	Runs the specified shell command.

The following is an example of a trace session in which the trace subcommands are used. First, the system trace points have been displayed. Second, a trace on the system calls have been selected. Of course, you can trace on more than one trace point. Be aware that trace takes a lot of data. Only the first few lines are shown in the following example:

```
# trcrpt -j |pg
004 TRACEID IS ZERO
100 FLIH
200 RESUME
102 SLIH
103 RETURN FROM SLIH
101 SYSTEM CALL
104 RETURN FROM SYSTEM CALL
106 DISPATCH
10C DISPATCH IDLE PROCESS
11F SET ON READY QUEUE
134 EXEC SYSTEM CALL
139 FORK SYSTEM CALL
107 FILENAME TO VNODE (lookupn)
15B OPEN SYSTEM CALL
130 CREAT SYSTEM CALL
19C WRITE SYSTEM CALL
163 READ SYSTEM CALL
10A KERN_PFS
10B LVM BUF STRUCT FLOW
116 XMALLOC size,align,heap
117 XMFREE address,heap
118 FORKCOPY
11E ISSIG
169 SBREAK SYSTEM CALL
# trace -d -j 101 -m "system calls trace example"
-> trcon
-> !cp /tmp/xbugs .
-> trcoff
-> quit
# trcrpt -O "exec=on,pid=on" > cp.trace
# pg cp.trace
pr 3 11:02:02 1991
System: AIX smiller Node: 3
Machine: 000247903100
Internet Address: 00000000 0.0.0.0
system calls trace example
trace -d -j 101 -m -m system calls trace example
```

ID	PROCESS NAME	PID	I	ELAPSED_SEC	DELTA MSEC	APPL SYSCALL
001	trace	13939		0.000000000	0.000000	TRACE ON chan 0
101	trace	13939		0.000251392	0.251392	kwritev
101	trace	13939		0.000940800	0.689408	sigprocmask
101	trace	13939		0.001061888	0.121088	kreadv
101	trace	13939		0.001501952	0.440064	kreadv
101	trace	13939		0.001919488	0.417536	kiocntl
101	trace	13939		0.002395648	0.476160	kreadv
101	trace	13939		0.002705664	0.310016	kiocntl

Controlling the trace Facility by Commands

If you configure the **trace** routine to run asynchronously (the **-a** option), you can control the trace facility with the following commands:

trcon Triggers collection of trace data on.
trcoff Triggers collection of trace data off.
trcstop Stops collection of trace data (like **trcoff**) and terminates the **trace** routine.

Controlling the trace Facility by Subroutines

The controls for the **trace** routine are available as subroutines from the **librts.a** library. The subroutines return zero on successful completion. The subroutines are:

trcon Triggers collection of **trace** data on.
trcoff Triggers collection of **trace** data off.
trcstop Stops collection of **trace** data (like **trcoff**) and terminates the **trace** routine.

Controlling the trace Facility with ioctl Calls

The subroutines for controlling **trace** open the trace control device (*/dev/systrctl*), issue the appropriate **ioctl** command, close the control device and return. To control tracing around sections of code, it can be more efficient for a program to issue the **ioctl** controls directly. This avoids the unnecessary, repetitive opening and closing of the trace control device, at the expense of exposing some of the implementation details of **trace** control. To use the **ioctl** call in a program, include **sys/trcctl.h** to define the **ioctl** commands. The syntax of the **ioctl** is as follows:

```
ioctl (fd, CMD, Channel)
```

where:

fd File descriptor returned from opening */dev/systrctl*
CMD TRCON, TRCOFF, or TRCSTOP
Channel Trace channel (0 for system trace).

The following code sample shows how to start a **trace** from a program and only trace around a specified section of code:

```
#include <sys/trcctl.h>
extern int trcstart(char *arg);
char *ctl_dev = "/dev/systrctl";
int ctl_fd
main()
{
    printf("configuring trace collection \n");
    if (trcstart("-ad")){
        perror("trcstart");
        exit(1);
    }
    if((ctl_fd =open (ctl_dev))<0){
        perror("open ctl_dev");
        exit(1);
    }
}
```

```

    }
    printf("turning trace collection on \n");
    if(ioctl(ctl_fd,TRCON,0)){
        perror("TRCON");
        exit(1);
    }
    /* code between here and trcoff ioctl will be traced */
    printf("turning trace off\n");
    if (ioctl(ctl_fd,TRCOFF,0)){
        perror("TRCOFF");
        exit(1);
    }
    exit(0);
}

```

Producing a trace Report

A trace report facility formats and displays the collected event stream in readable form. This report facility displays text and data for each event according to rules provided in the trace format file. The default trace format file is `/etc/trcfmt` and contains a stanza for each event ID. The stanza for the event provides the report facility with formatting rules for that event. This technique allows you to add your own events to programs and insert corresponding event stanzas in the format file to have their new events formatted.

This report facility does not attempt to extract summary statistics (such as CPU utilization and disk utilization) from the event stream. This can be done in several other ways. To create simple summaries, consider using `awk` scripts to process the output obtained from the `trcrpt` command.

The trcrpt Command

The syntax of the `trcrpt` command is as follows:

```

trcrpt [ -c ] [ -d List ] [ -e Date ] [ -h ] [ -j ] [ -k List ] [ -n Name ] [ -o File ]
[ -p List ] [ -q ] [ -r ] [ -s Date ] [ -t File ] [ -v ] [ -O Options ] [ -T List ]
[ LogFile ]

```

Normally the `trcrpt` output goes to standard output. However, it is generally more useful to redirect the report output to a file. The options are:

- c** Causes the `trcrpt` command to check the syntax of the trace format file. The trace format file checked is either the default (`/etc/trcfmt`) or the file specified by the `-t` flag with this command. You can check the syntax of the new or modified format files with this option before attempting to use them.
- d List** Allows you to specify a list of events to be included in the `trcrpt` output. This is useful for eliminating information that is superfluous to a given analysis and making the volume of data in the report more manageable. You may have commonly used event profiles, which are lists of events that are useful for a certain type of analysis.
- e Date** Ends the report time with entries on, or before the specified date. The *Date* parameter has the form *mmddhhmmssyy* (month, day, hour, minute, second, and year). Date and time are recorded in the trace data only when trace data collection is started and stopped. If you stop and restart trace data collection multiple times during a trace session, date and time are recorded

each time you start or stop a trace data collection. Use this flag in combination with the **-s** flag to limit the trace data to data collected during a certain time interval.

- h** Omit the column headings of the report.
- j** Causes the **trcrpt** command to produce a list of all the defined events from the specified trace format file. This option is useful in creating an initial file that you can edit to use as an include or exclude list for the **trcrpt** or **trace** command.
- k List** Similar to the **-d** flag, but allows you to specify a list of events to exclude from the **trcrpt** output.
- n Name**
Specifies the kernel name list file to be used by **trcrpt** to convert kernel addresses to routine names. If not specified, the report facility uses the symbol table in **/unix**. A kernel name list file that matches the system the data was collected on is necessary to produce an accurate trace report. You can create such a file for a given level of system with the **trcnm** command:

```
trcnm /unix > Name
```
- o File** Writes the report to a file instead of to standard output.
- p List** Limits the **trcrpt** output to events that occurred during the running of specific processes. List the processes by process name or process ID.
- q** Suppresses detailed output of syntax error messages. This is not an option you typically use.
- r** Produces a raw binary format of the trace data. Each event is output as a record in the order of occurrence. This is not necessarily the order in which the events are in the trace log file since the logfile can wrap. If you use this option, direct the output to a file (or process), since the binary form of the data is not displayable.
- t File** Allows you to specify a trace format file other than the default (**/etc/trcfmt**).
- T List** Limits the report to the kernel thread IDs specified by the *List* parameter. The list items are kernel thread IDs separated by commas. Starting the list with a kernel thread ID limits the report to all kernel thread IDs *in* the list. Starting the list with a **!** (exclamation point) followed by a kernel thread ID limits the report to all kernel thread IDs *not in* the list.
- O options**
Allows you to specify formatting options to the **trcrpt** command in a comma separated list. Do not put spaces after the commas. These options take the form of **option=selection**. If you do not specify a selection, the command uses the default selection. The possible options are discussed in the following sections. Each option is introduced by showing its default selection.
 - 2line=off**
This option lets the user specify whether the lines in the event report are split and displayed across two lines. This is useful when more columns of information have been requested than can be displayed on the width of the output device.
 - cpuid=off**
Lets you specify whether to display a column that contains the physical processor number.

endtime=nnn.nnnnnnnnn

The **starttime** and **endtime** option permit you to specify an elapsed time interval in which the **trcrpt** produces output. The elapsed time interval is specified in seconds with nanosecond resolution.

exec=off

Lets you specify whether a column showing the path name of the current process is displayed. This is useful in showing what process (by name) was active at the time of the event. You typically want to specify this option. We recommend that you specify **exec=on** and **pid=on**.

ids=on

Lets you specify whether to display a column that contains the event IDs. If the selection is on, a three-digit hex ID is shown for each event. The alternative is off.

pagesize=0

Lets you specify how often the column headings is reprinted. The default selection of 0 displays the column headings initially only. A selection of 10 displays the column heading every 10 lines.

pid=off

Lets you specify whether a column showing the process ID of the current process is displayed. It is useful to have the process ID displayed to distinguish between several processes with the same executable name. We recommend that you specify **exec=on** and **pid=on**.

starttime=nnn.nnnnnnnnn

The **starttime** and **endtime** option permit you to specify an elapsed time interval in which the **trcrpt** command produces output. The elapsed time interval is specified in seconds with nanosecond resolution.

svc=off

Lets you specify whether the report should contain a column that indicates the active system call for those events that occur while a system call is active.

tid=off

Lets you specify whether a column showing the thread ID of the current thread is displayed. It is useful to have the thread ID displayed to distinguish between several threads within the same process. Alternatively, you can specify **tid=on**.

timestamp=0

The report can contain two time columns. One column is elapsed time since the **trace** command was initiated. The other possible time column is the delta time between adjacent events. The option controls if and how these times are displayed. The selections are:

- 0 Provides both an elapsed time from the start of **trace** and a delta time between events. The elapsed time is shown in seconds and the delta time is shown in milliseconds. Both fields show resolution to a nanosecond. This is the default value.
- 1 Provides only an elapsed time column displayed as seconds with resolution shown to microseconds.

- 2 Provides both an elapsed time and a delta time column. The elapsed time is shown in seconds with nanosecond resolution, and delta time is shown in microseconds with microsecond resolution.
- 3 Omits all time stamps from the report.

logfile The **logfile** is the name of the file that contains the event data to be processed by the **trcrpt** command. The default is the **/usr/adm/ras/trcfile** file.

Defining trace Events

The operating system is shipped with predefined trace hooks (events). You need only activate **trace** to capture the flow of events from the operating system. You may want to define trace events in your code during development for tuning purposes. This provides insight into how the program is interacting with the system. The following sections provide the information that is required to do this.

Possible Forms of a trace Event Record

A trace event can take several forms. An event consists of a:

- Hookword
- Data words (optional)
- A TID, or thread identifier
- Timestamp (optional)

The "Format of a Trace Event Record" figure illustrates a trace event. A four-bit type is defined for each form the event record can take. The type field is imposed by the recording routine so that the report facility can always skip from event to event when processing the data, even if the formatting rules in the trace format file are incorrect or missing for that event.

12 bit Hook ID	4 bit Type	16 bit Data Field
D1 Optional Data Word 1		
D2 Optional Data Word 2		
D3 Optional Data Word 3		
D4 Optional Data Word 4		
D5 Optional Data Word 5		
TID (Thread ID)		
Optional Time Stamp		

Format of a Trace Event Record

An event record should be as short as possible. Many system events use only the hookword and timestamp. There is another event type you should seldom use because it is less efficient. It is a long format that allows you to record a variable length of data. In this long form, the 16-bit data field of the hookword is converted to a *length* field that describes the length of the event record.

Macros for Recording trace Events

There is a macro to record each possible type of event record. The macros are defined in the `sys/trcmacros.h` header file. The event IDs are defined in the `sys/trckid.h` header file. Include these two header files in any program that is recording **trace** events. The macros to record system (channel 0) events with a time stamp are:

- `TRCHKL0T` (hw)
- `TRCHKL1T` (hw,D1)
- `TRCHKL2T` (hw,D1,D2)
- `TRCHKL3T` (hw,D1,D2,D3)
- `TRCHKL4T` (hw,D1,D2,D3)
- `TRCHKL5T` (hw,D1,D2,D3,D4,D5)

Similarly, to record non-time stamped system events (channel 0), use the following macros:

- `TRCHKL0` (hw)
- `TRCHKL1` (hw,D1)
- `TRCHKL2` (hw,D1,D2)
- `TRCHKL3` (hw,D1,D2,D3)
- `TRCHKL4` (hw,D1,D2,D3,D4)
- `TRCHKL5` (hw,D1,D2,D3,D4,D5)

There are only two macros to record events to one of the generic channels (channels 1-7). These are:

- **TRCGEN** (ch,hw,d1,len,buf)
- **TRCGENT** (ch,hw,d1,len,buf)

These macros record a hookword (hw), a data word (d1) and a length of data (len) specified in bytes from the user's data segment at the location specified (buf) to the event stream specified by the channel (ch).

Use of Event IDs (hookids)

Event IDs are 12 bits (or 3-digit hexadecimal), for a possibility of 4096 IDs. Event IDs that are permanently left in and shipped with code need to be permanently assigned. Permanently assigned event IDs are defined in the `sys/trchkid.h` header file.

To allow you to define events in your environments or during development, a range of event IDs exist for temporary use. The range of event IDs for temporary use is hex 010 through hex 0FF. No permanent (shipped) events are assigned in this range. You can freely use this range of IDs in your own environment. If you do use IDs in this range, do not let the code leave your environment.

Permanent events must have event IDs assigned by the current owner of the trace component. You should conserve event IDs because they are limited. Event IDs can be extended by the data field. The only reason to have a unique ID is that an ID is the level at which collection and report filtering is available in the trace facility. An ID can be collected or not collected by the trace collection process and reported or not reported by the trace report facility. Whole applications can be instrumented using only one event ID. The only restriction is that the granularity on choosing visibility is to choose whether events for that application are visible.

A new event can be formatted by the trace report facility (`trcrpt` command) if you create a stanza for the event in the trace format file. The trace format file is an editable ASCII file. The syntax for a format stanzas is shown in "Syntax for Stanzas in the trace Format File" on page 568. All permanently assigned event IDs should have an appropriate stanza in the default trace format file shipped with the base operating system.

Suggested Locations and Data for Permanent Events

The intent of permanent events is to give an adequate level of visibility to determine execution, and data flow, and have an adequate accounting for how CPU time is being consumed. During code development, it can be desirable to make very detailed use of trace for a component. For example, you can choose to trace the entry and exit of every subroutine in order to understand and tune pathlength. However, this would generally be an excessive level of instrumentation to ship for a component.

We suggest that you consult a performance analyst for decisions regarding what events and data to capture as permanent events for a new component. The following paragraphs provide some guidelines for these decisions.

Events should capture execution flow and data flow between major components or major sections of a component. For example, there are existing events that capture the interface between the virtual memory manager and the logical volume manager. If work is being queued, data that identifies the queued item (a handle)

should be recorded with the event. When a queue element is being processed, the "dequeue" event should provide this identifier as data also, so that the queue element being serviced is identified.

Data or requests that are identified by different handles at different levels of the system should have events and data that allow them to be uniquely identified at any level. For example, a read request to the physical file system is identified by a file descriptor and a current offset in the file. To a virtual memory manager, the same request is identified by a segment ID and a virtual page address. At the disk device driver level, this request is identified as a pointer to a structure which contains pertinent data for the request.

The file descriptor or segment information is not available at the device driver level. Events must provide the necessary data to link these identifiers so that, for example, when a disk interrupt occurs for incoming data, the identifier at that level (which can simply be the buffer address for where the data is to be copied) can be linked to the original user request for data at some offset into a file.

Events should provide visibility to major protocol events such as requests, responses, acknowledgements, errors, and retries. If a request at some level is fragmented into multiple requests, a trace event should indicate this and supply linkage data to allow the multiple requests to be tracked from that point. If multiple requests at some level are coalesced into a single request, a trace event should also indicate this and provide appropriate data to track the new request.

Use events to give visibility to resource consumption. Whenever resources are claimed, returned, created or deleted an event should record the fact. For example, claiming or returning buffers to a buffer pool or growing or shrinking the number of buffers in the pool.

The following guidelines can help you determine where and when you should have trace hooks in your code:

- Tracing entry and exit points of every function is not necessary. Provide only key actions and data.
- Show linkage between major code blocks or processes.
- If work is queued, associate a name (handle) with it and output it as data.
- If a queue is being serviced, the trace event should indicate the unique element being serviced.
- If a work request or response is being referenced by different handles as it passes through different software components, trace the transactions so the action or receipt can be identified.
- Place trace hooks so that requests, responses, errors, and retries can be observed.
- Identify when resources are claimed, returned, created, or destroyed.

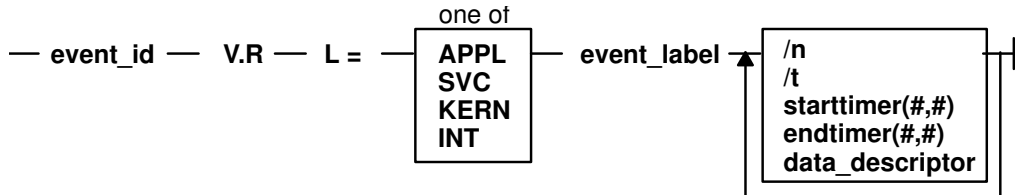
Also note that:

- A trace ID can be used for a group of events by "switching" on one of the data fields. This means that a particular data field can be used to identify from where the trace point was called. The trace format routine can be made to format the trace data for that unique trace point.
- The trace hook is the level at which a group of events can be enabled or disabled. Note that trace hooks can be grouped in SMIT. For more information, see "SMIT Trace Hook Groups" on page 579.

Syntax for Stanzas in the trace Format File

The intent of the trace format file is to provide rules for presentation and display of the expected data for each event ID. This allows new events to be formatted without changing the report facility. Rules for new events are simply added to the format file. The syntax of the rules provide flexibility in the presentation of the data.

Refer to the `/etc/tcrfmt` file to see examples of the syntax for stanzas that appear in the trace format file.



Syntax of a Stanza in the Format File

A trace format stanza can be as long as required to describe the rules for any particular event. The stanza can be continued to the next line by terminating the present line with a `\` (backslash) character. The fields are:

event_id	Each stanza begins with the three-digit hexadecimal event ID that the stanza describes, followed by a space.
V.R	This field describes the version (V) and release (R) that the event was first assigned. Any integers work for V and R, and you may want to keep your own tracking mechanism.
L=	The text description of an event can begin at various indentation levels. This improves the readability of the report output. The indentation levels correspond to the level at which the system is running. The recognized levels are: <ul style="list-style-type: none"> APPL Application level SVC Transitioning system call KERN Kernel level INT Interrupt
event_label	The <i>event_label</i> is an ASCII text string that describes the overall use of the event ID. This is used by the <code>-j</code> option of the <code>tcrpt</code> command to provide a listing of events and their first level description. The event label also appears in the formatted output for the event unless the <code>event_label</code> field starts with an <code>@</code> character.
\n	The event stanza describes how to parse, label and present the data contained in an event record. You can insert a <code>\n</code> (newline) in the event stanza to continue data presentation of the data on a new line. This allows the presentation of the data for an event to be several lines long.
\t	The <code>\t</code> (tab) function inserts a tab at the point it is encountered in parsing the description. This is similar to the way the <code>\n</code> function inserts new lines. Spacing can also be inserted by spaces in the <code>data_label</code> or <code>match_label</code> fields.

starttimer(##)

The starttimer and endtimer fields work together. The (##) field is a unique identifier that associates a particular starttimer value with an endtimer that has the same identifier. By convention, if possible, the identifiers should be the ID of starting event and the ID of the ending event.

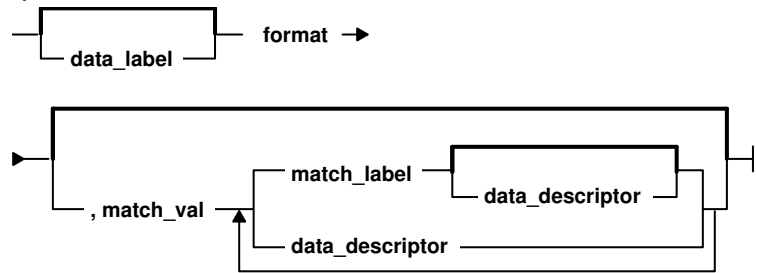
When the report facility encounters a start timer directive while parsing an event, it associates the starting events time with the unique identifier. When an end timer with the same identifier is encountered, the report facility outputs the delta time (this appears in brackets) that elapsed between the starting event and ending event. The begin and end system call events make use of this capability. On the return from system call event, a delta time is shown that indicates how long the system call took.

endtimer(##)

See the starttimer field in the preceding paragraph.

data_descriptor

The data_descriptor field is the fundamental field that describes how the report facility consumes, labels, and presents the data. The Syntax of the data_descriptor Field figure illustrates this field's syntax.



Syntax of the data_descriptor Field

The various subfields of the data_descriptor field are:

data_label

The data label is an ASCII string that can optionally precede the output of data consumed by the following format field.

format

Review the format of an event record depicted in the Format of a trace Event Record figure. You can think of the report facility as having a pointer into the data portion of an event. This data pointer is initialized to point to the beginning of the event data (the 16-bit data field in the hookword). The format field describes how much data the report facility consumes from this point and how the data is considered. For example, a value of **Bm.n** tells the report facility to consume m bytes and n bits of data and to consider it as binary data.

The possible format fields are described in the following section. If this field is not followed by a comma, the report facility outputs the consumed data in the format specified. If this field is followed by a comma, it signifies that the data is not to be displayed but instead compared against the following match_val's field. The data descriptor associated with the matching match_val field is then applied to the remainder of the data.

match_val

The match value is data of the same format described by the preceding format fields. Several match values typically follow a format field that is being matched. The successive match fields are separated by commas. The last match value is not followed by a comma. Use the character string * as a pattern-matching character to match anything. A pattern-matching character is frequently used as the last element of the match_val field to specify default rules if the preceding match_val field did not occur.

match_label The match label is an ASCII string that is output for the corresponding match.

Each of the possible format fields is described in the comments of the `/etc/trcfmt` file. The following shows several possibilities:

Format field	descriptions
Am.n	This value specifies that m bytes of data are consumed as ASCII text, and that it is displayed in an output field that is n characters wide. The data pointer is moved m bytes.
S1, S2, S4	Left justified string. The length of the field is defined as 1 byte (S1), 2 bytes (S2), or 4 bytes (S4). The data pointer is moved accordingly.
Bm.n	Binary data of m bytes and n bits. The data pointer is moved accordingly.
Xm	Hexadecimal data of m bytes. The data pointer is moved accordingly.
D2, D4	Signed decimal format. Data length of 2 (D2) bytes or 4 (D4) bytes is consumed.
U2, U4	Unsigned decimal format. Data length of 2 or 4 bytes is consumed.
F4, F8	Floating point of 4 or 8 bytes.
Gm.n	Positions the data pointer. It specifies that the data pointer is positioned m bytes and n bits into the data.
Om.n	Skip or omit data. It omits m bytes and n bits.
Rm	Reverse the data pointer m bytes.

Some macros are provided that can be used as format fields to quickly access data. For example:

\$D1, \$D2, \$D3, \$D4, \$D5	These macros access data words 1 through 5 of the event record without moving the data pointer. The data accessed by a macro is hexadecimal by default. A macro can be cast to a different data type (X, D, U, B) by using a % character followed by the new format code. For example, the following macro causes data word one to be accessed, but to be considered as 2 bytes and 3 bits of binary data: \$D1%B2.3
\$HD	This macro accesses the first 16 bits of data contained in the hookword, in a similar manner as the \$D1 through \$D5 macros access the various data words. It is also considered as hexadecimal data, and also can be cast.

You can define other macros and use other formatting techniques in the trace format file. This is shown in the following trace format file example.

Example Trace Format File

```
# .
# .
# .
#
# I. General Information
#
# A. Binary format for the tracehook calls. (1 column = 4 bits)
# trchk      MmTDDDD
# trchkt     MmTDDDDttttttt
# trchk1     MmTDDDD11111111
# trchk1t    MmTDDDD11111111ttttttt
# trchk1g    MmTDDDD1111111122222222333333334444444455555555
```



```

#          xxxxxhello world\0xxxxxx
#
# ii. A16.16 results in:          |hello world |
#   DATA_POINTER-----|
#                               V
#          xxxxxhello world\0xxxxxx
#
# iii. A16 results in:          |hello world|
#   DATA_POINTER-----|
#                               V
#          xxxxxhello world\0xxxxxx
#
# iv. A0.16 results in:          |           |
#   DATA_POINTER|
#               V
#          xxxxxhello world\0xxxxxx
#
# S1, S2, S4
# Left justified ascii string.
# The length of the string is in the first byte(half-word, word)
# of the data. This length of the string does not include this byte.
# The data pointer is advanced by the length value.
#   Example
#   DATA_POINTER|
#               V
#          xxxxBhello worldxxxxxx   (B = hex 0x0b)
#
# i. S1 results in:          |hello world|
#   DATA_POINTER-----|
#                               V
#          xxxxBhello worldxxxxxx
#
# $reg%S1
#   A register with the format code of 'Sx' works "backwards"
#   from a register with a different type. The format is Sx,
#   but the length of the string comes from $reg instead of the
#   next n bytes.
#
# Bm.n
#   Binary format.
#   m = length in bytes
#   n = length in bits
#   The length in bits of the data is m * 8 + n. B2.3 and B0.19
#   are the same. Unlike the other printing FORMAT codes, the
#   DATA_POINTER can be bit aligned and is not rounded up to
#   the next byte boundary.
#
# Xm
#   Hex format.
#   m = length in bytes. m=0 thru 16
#   The DATA_POINTER is advanced by m.
#
# D2, D4
#   Signed decimal format.
#   The length of the data is 2 (4) bytes.
#   The DATA_POINTER is advanced by 2 (4).
#
# U2, U4
#   Unsigned decimal format.
#   The length of the data is 2 (4) bytes.
#   The DATA_POINTER is advanced by 2 (4).
#
# F4
#   Floating point format. (like %0.4E)
#   The length of the data is 4 bytes.
#   The format of the data is that of C type 'float'.
#   The DATA_POINTER is advanced by 4.

```

```

#
# F8
#   Floating point format. (like %0.4E)
#   The length of the data is 8 bytes.
#   The format of the data is that of C type 'double'.
#   The DATA_POINTER is advanced by 8.
#
# HB
#   Number of bytes in trcgen() variable length buffer.
#   This is also equal to the 16 bit hookdata.
#   The DATA_POINTER is not changed.
#
# HT
#   The hooktype. (0 - E)
#   trcgen = 0, trchk = 1, trchl = 2, trchkg = 6
#   trcgent = 8, trchkt = 9, trchlt = A, trchkg = E
#   HT & 0x07 masks off the timestamp bit
#   This is used for allowing multiple, different trchkx() calls with
#   the same template.
#   The DATA_POINTER is not changed.
#
# C. Codes that interpret the data in some way before output.
# T4
#   Output the next 4 bytes as a data and time string,
#   in GMT timezone format. (as in ctime(&seconds))
#   The DATA_POINTER is advanced by 4.
#
# E1,E2,E4
#   Output the next byte (half_word, word) as an 'errno' value,
#   replacing the numeric code with the corresponding #define name in
#   /usr/include/sys/errno.h
#   The DATA_POINTER is advanced by 1, 2, or 4.
#
# P4
#   Use the next word as a process id (pid), and output the
#   pathname of the executable with that process id.Process
#   ids and their pathnames are acquired by the trace command at
#   the start of a trace and by trcrpt via a special EXEC tracehook.
#   The DATA_POINTER is advanced by 4.
#
# \t
#   Output a tab. \t\t\t outputs 3 tabs. Tabs are expanded to spaces,
#   using a fixed tabstop separation of 8.If L=0 indentation is used,
#   the first tabstop is at 3.
#   The DATA_POINTER advances over the \t.
#
# \n
#   Output a newline. \n\n\n outputs 3 newlines.
#   The newline is left-justified according to the INDENTATION LEVEL.
#   The DATA_POINTER advances over the \n.
#
# $macro
#   The value of 'macro' is output as a %04X value. Undefined
#   macros have the value of 0000.
#   The DATA_POINTER is not changed.
#   An optional format can be used with macros:
#   $v1%X4 will output the value $v1 in X4 format.
#   $zz%B0.8 will output the value $v1 in 8 bits of binary.
#   Understood formats are: X, D, U, B. Others default to X2.
#
# "string" 'string' data type
#   Output the characters inside the double quotes exactly. A string
#   is treated as a descriptor. Use "" as a NULL string.
#
# 'string format $macro' If a string is backquoted, it is expanded
#   as a quoted string, except that FORMAT codes and $registers are
#   expanded as registers.

```

```

#
# III. SWITCH statement
#   A format code followed by a comma is a SWITCH statement.
#   Each CASE entry of the SWITCH statement consists of
#     1. a 'matchvalue' with a type (usually numeric) corresponding
#        to the format code.
#     2. a simple 'string' or a new 'descriptor' bounded by braces.
#        A descriptor is a sequence of format codes, strings,
#        switches and loops.
#     3. and a comma delimiter.
#   The switch is terminated by a CASE entry without a comma
#   delimiter. The CASE entry is selected to as the first
#   entry whose matchvalue is equal to the expansion of the format
#   code. The special matchvalue '\*' is a wildcard and matches
#   anything.
#   The DATA_POINTER is advanced by the format code.
#
#
# IV. LOOP statement
#   The syntax of a 'loop' is
#   LOOP format_code { descriptor }
#   The descriptor is executed N times, where N is the numeric value
#   of the format code. The DATA_POINTER is advanced by the
#   format code plus whatever the descriptor does. Loops are used to
#   output binary buffers of data, so descriptor is
#   usually simply X1 or X0. Note that X0 is like X1 but does not
#   supply a space separator ' ' between each byte.
#
# V. macro assignment and expressions
#   'macros' are temporary (for the duration of that event) variables
#   that work like shell variables.
#   They are assigned a value with the syntax:
#   {{ $xxx = EXPR }}
#   where EXPR is a combination of format codes, macros, and constants.
#   Allowed operators are + - / *
#   For example:
#   {{{ $dog = 7 + 6 }} {{ $cat = $dog * 2 }} $dog $cat
#
#   will output:
#   #000D 001A
#
#   Macros are useful in loops where the loop count is not always
#   just before the data:
#   #G1.5 {{ $count = B0.5 }} G11 LOOP $count {X0}
#
#   Up to 25 macros can be defined per template.
#
#
# VI. Special macros:
# $RELLINENO   line number for this event. The first line starts at 1.
# $D1 - $D5    dataword 1 through dataword 5. No change to datapointer.
# $HD          hookdata (lower 16 bits)
# $SVC         Output the name of the current SVC
# $EXECPTH     Output the pathname of the executable for current process.
# $PID         Output the current process id.
# $ERROR       Output an error message to the report and exit from the
#              template after the current descriptor is processed.
#              The error message supplies the logfile, logfile offset of
#              the start of that event, and the traceid.
# $LOGIDX      Current logfile offset into this event.
# $LOGIDX0     Like $LOGIDX, but is the start of the event.
# $LOGFILE     Name of the logfile being processed.
# $TRACEID     Traceid of this event.
# $DEFAULT     Use the DEFAULT template 008
# $STOP        End the trace report right away
# $BREAK       End the current trace event
# $SKIP        Like break, but don't print anything out.

```



```

# $DATAPOINTER The DATA_POINTER. It can be set and manipulated
#           like other user-macros.
#           {{ $DATAPOINTER = 5 }} is equivalent to G5
# $BASEPOINTER Usually 0. It is the starting offset into an event.The
#           actual offset is the DATA_POINTER + BASE_POINTER. It is used
#           with template subroutines, where the parts on an event have
#           the same structure, and can be printed by the same template,
#           but may have different starting points into an event.
#
# VII. Template subroutines
# If a macro name consists of 3 hex digits, it is a "template
# subroutine". The template whose traceid equals the macro name
# is inserted in place of the macro.
#
# The data pointer is where it was when the template
# substitution was encountered.Any change made to the data pointer
# by the template subroutine remains in affect when the template
# ends.
#
# Macros used within the template subroutine correspond to those
# in the calling template. The first definition of a macro in the
# called template is the same variable as the first in the called.
# The names are not related.
#
# Example:
# Output the trace label ESDI STRATEGY.
# The macro '$stat' is set to bytes 2 and 3 of the trace event.
# Then call template 90F to interpret a buf header. The macro
# '$return' corresponds to the macro '$rv', since they were
# declared in the same order. A macro definition with
# no '=' assignment just declares the name
# like a place holder. When the template returns,the saved special
# status word is output and the returned minor device number.
#
#900 1.0 "ESDI STRATEGY" {{ $rv = 0 }} {{ $stat = X2 }} \
#   $90F \n\
#special_esdi_status=$stat for minor device $rv
#
#90F 1.0 "" G4 {{ $return }} \
#   block number X4 \n\
#   byte count  X4 \n\
#   B0.1, 1 B_FLAG0 \
#   B0.1, 1 B_FLAG1 \
#   B0.1, 1 B_FLAG2 \
#   G16 {{ $return = X2 }}
#
# Note: The $DEFAULT reserved macro is the same as $008
#
# VII. BITFLAGS statement
# The syntax of a 'bitflags' is
# BITFLAGS [format_code|register],
#         flag_value string {optional string if false}, or
#         '&' mask field_value string,
#         ...
#
# This statement simplifies expanding state flags, since it look
# a lot like a series of #defines.
# The '&' mask is used for interpreting bit fields.
# The mask is anded to the register and the result is compared to
# the field_value. If a match, the string is printed.
# The base is 16 for flag_values and masks.
# The DATA_POINTER is advanced if a format code is used.
# Note:the default base for BITFLAGS is 16. If the mask or field
# value has a leading 0, the number is octal. 0x or 0X makes the
# number hex.
# A 000 traceid will use this template

```

```

# This id is also used to define most of the template functions.
# filemode(omode) expand omode the way ls -l does. The
# call to setdelim() inhibits spaces between the chars.
#

```

Examples of Coding Events and Formatting Events

There are five basic steps involved in generating a trace from your software program.

Step 1: Enable the trace: Enable and disable the trace from your software that has the trace hooks defined. The following code shows the use of trace events to time the running of a program loop.

```

#include <sys/trcctl.h>
#include <sys/trcmacros.h>
#include <sys/trckid.h>
char *ctl_file = "/dev/systrctl";
int ctld;
int i;
main()
{
    printf("configuring trace collection \n");
    if (trcstart("-ad")){
        perror("trcstart");
        exit(1);
    }
    if((ctld = open(ctl_file,0)<0){
        perror(ctl_file);
        exit(1);
    }
    printf("turning trace on \n");
    if(ioctl(ctld,TRCON,0)){
        perror("TRCON");
        exit(1);
    }
    /* here is the code that is being traced */
    for(i=1;i<11;i++){
        TRCHKLIT(HKWD_USER1,i);
        /* sleep(1) */
        /* you can uncomment sleep to make the loop
        /* take longer. If you do, you will want to
        /* filter the output or you will be */
        /* overwhelmed with 11 seconds of data */
    }
    /* stop tracing code */
    printf("turning trace off\n");
    if(ioctl(ctld,TRCSTOP,0)){
        perror("TRCOFF");
        exit(1);
    }
    exit(0);
}

```

Step 2: Compile your program: When you compile the sample program, you need to link to the `librts.a` library:

```
cc -o sample sample.c -l rts
```

Step 3: Run the program: Run the program. In this case, it can be done with the following command:

```
./sample
```

You must have root privilege if you use the default file to collect the trace information (`/usr/adm/ras/trcfile`).

Step 4: Add a stanza to the format file: This provides the report generator with the information to correctly format your file. The report facility does not know how to format the `HKWD_USER1` event, unless you provide rules in the trace format file.

The following is an example of a stanza for the `HKWD_USER1` event. The `HKWD_USER1` event is event ID 010 hexadecimal. You can verify this by looking at the `sys/trchkid.h` header file.

```
# User event HKWD_USER1 Formatting Rules Stanza
# An example that will format the event usage of the sample program
010 1.0 L=APPL "USER EVENT - HKWD_USER1" 02.0 \n\
    "The # of loop iterations =" U4\n\
    "The elapsed time of the last loop = "\
    endtimer(0x010,0x010) starttimer(0x010,0x010)
```

Note: When entering the example stanza, do not modify the master format file `/etc/trcfmt`. Instead, make a copy and keep it in your own directory. This allows you to always have the original trace format file available.

Step 5: Run the format/filter program: Filter the output report to get only your events. To do this, run the `trcrpt` command:

```
trcrpt -d 010 -t mytrcfmt -0 exec=on -o sample.rpt
```

The formatted trace results are:

ID	PROC NAME	I	ELAPSED_SEC	DELTA_MSEC	APPL	SYSCALL	KERNEL	INTERRUPT
010	sample		0.000105984	0.105984	USER HOOK 1			
					The data field for the user hook = 1			
010	sample		0.000113920	0.007936	USER HOOK 1			
					The data field for the user hook = 2 [7 usec]			
010	sample		0.000119296	0.005376	USER HOOK 1			
					The data field for the user hook = 3 [5 usec]			
010	sample		0.000124672	0.005376	USER HOOK 1			
					The data field for the user hook = 4 [5 usec]			
010	sample		0.000129792	0.005120	USER HOOK 1			
					The data field for the user hook = 5 [5 usec]			
010	sample		0.000135168	0.005376	USER HOOK 1			
					The data field for the user hook = 6 [5 usec]			
010	sample		0.000140288	0.005120	USER HOOK 1			
					The data field for the user hook = 7 [5 usec]			
010	sample		0.000145408	0.005120	USER HOOK 1			
					The data field for the user hook = 8 [5 usec]			
010	sample		0.000151040	0.005632	USER HOOK 1			
					The data field for the user hook = 9 [5 usec]			
010	sample		0.000156160	0.005120	USER HOOK 1			
					The data field for the user hook = 10 [5 usec]			

Usage Hints

The following sections provide some examples and suggestions for use of the trace facility.

Viewing trace Data

Include several optional columns of data in the trace output. This causes the output to exceed 80 columns. It is best to view the reports on an output device that supports 132 columns.

Bracketing Data Collection

Trace data accumulates rapidly. Bracket the data collection as closely around the area of interest as possible. One technique for doing this is to issue several commands on the same command line. For example, the command:

```
trace -a; cp /etc/trcfmt /tmp/junk; trcstop
```

captures the total execution of the copy command.

Note: This example is more educational if the source file is not already cached in system memory. The `trcfmt` file can be in memory if you have been modifying it or producing trace reports. In that case, choose as the source file some other file that is 50 to 100KB and has not been touched.

Reading a trace Report

The trace facility displays system activity. It is a useful learning tool to observe how the system actually performs. The previous output is an interesting example to browse. To produce a report of the copy, use the following:

```
trcrpt -0 "exec=on,pid=on" > cp.rpt
```

In the `cp.rpt` file you can see the following activities:

- The fork, exec, and page fault activities of the `cp` process.
- The opening of the `/etc/trcfmt` file for reading and the creation of the `/tmp/junk` file.
- The successive **read** and **write** subroutines to accomplish the copy.
- The `cp` process becoming blocked while waiting for I/O completion, and the wait process being dispatched.
- How logical volume requests are translated to physical volume requests.
- The files are mapped rather than buffered in traditional kernel buffers. The read accesses cause page faults that must be resolved by the virtual memory manager.
- The virtual memory manager senses sequential access and begins to prefetch the file pages.
- The size of the prefetch becomes larger as sequential access continues.
- The writes are delayed until the file is closed (unless you captured execution of the `sync` daemon that periodically forces out modified pages).
- The disk device driver coalesces multiple file requests into one I/O request to the drive when possible.

The trace output looks a little overwhelming at first. This is a good example to use as a learning aid. If you can discern the activities described, you are well on your way to being able to use the trace facility to diagnose system performance problems.

Effective Filtering of the trace Report

The full detail of the trace data may not be required. You can choose specific events of interest to be shown. For example, it is sometimes useful to find the number of times a certain event occurred. Answer the question, "how many opens occurred in the copy example?" First, find the event ID for the **open** subroutine:

```
trcrpt -j | pg
```

You can see that event ID 15b is the open event. Now, process the data from the copy example (the data is probably still in the log file) as follows:

```
trcrpt -d 15b -0 "exec=on"
```

The report is written to standard output and you can determine the number of opens that occurred. If you want to see only the opens that were performed by the `cp` process, run the report command again using:

```
trcrpt -d 15b -p cp -0 "exec=on"
```

This command shows only the opens performed by the **cp** process.

SMIT Trace Hook Groups

Combining multiple trace hooks into a trace hook group allows all hooks to be turned on at once when starting a trace.

The first item on the Start Trace menu is EVENT GROUPS to trace. The groups that can be traced are defined in the SWservAt ODM object class. Each group uses the *Name_trcgrpdesc* and *Name_trcgrp* attributes. *Name* is the name of the trace group that must be no longer than 8 characters. The *_trcgrpdesc* attribute specifies the group description, and the *_trcgrp* attribute specifies the trace hooks in the group. A group can have only one *_trcgrpdesc* description, but it can have multiple *_trcgrp* hook lists.

The following example shows the definition of the general kernel activity (gka) trace group.

```
SWservAt:
  attribute = "gka_trcgrpdesc"
  value = "GENERAL KERNEL ACTIVITY (files,execs,dispatches)"
SWservAt:
  attribute = "gka_trcgrp"
  value = "106,134,139,107,135,15b,12e"
```

Note that the value of **gka_trcgrpdesc** is shown by SMIT as the description of the group. The value list for **gka_trcgrp** contains the hooks in the group. You can add another **gka_trcgrp** attribute to add additional trace hooks to that group. The hook ids must be enclosed in double quotes (") and separated by commas.

Memory Overlay Detection System (MODS)

AIX Kernel Memory Overlay Detection System (MODS)

Some of the most difficult types of problems to debug are what are generally called "memory overlays." Memory overlays include the following:

- Writing to memory that is owned by another program or routine
- Writing past the end (or before the beginning) of declared variables or arrays
- Writing past the end (or before the beginning) of dynamically-allocated memory
- Writing to or reading from freed memory
- Freeing memory twice
- Calling memory allocation routines with incorrect parameters or under incorrect conditions.

In the kernel environment (including the AIX kernel, kernel extensions, and device drivers), memory overlay problems have been especially difficult to debug because tools for finding them have not been available. Starting with AIX Version 4.2.1, however, the Memory Overlay Detection System (MODS) helps detect memory overlay problems in the AIX kernel, kernel extensions, and device drivers.

Note: This feature does not detect problems in application code; it only watches kernel and kernel extension code.

bosdebug command

The **bosdebug** command turns the MODS facility on and off. Only the root user can run the **bosdebug** command.

To turn on the base MODS support, type:

```
bosdebug -M
```

For a description of all the available options, type:

```
bosdebug -?
```

Once you have run **bosdebug** with the options you want, run the **bosboot -a** command, then shutdown and reboot your system (using the **shutdown -r** command). If you need to make any changes to your **bosdebug** settings, you must run **bosboot -a** and **shutdown -r** again.

When to use the MODS feature

This feature is useful in the following circumstances:

- When developing your own kernel extensions or device drivers and want to test them thoroughly.
- When asked to turn this feature on by IBM AIX service to help in further diagnosing a problem that you are experiencing.

How MODS works

The primary goal of the MODS feature is to produce a dump file that accurately identifies the problem.

MODS works by turning on additional checking to help detect the conditions listed above. When any of these conditions is detected, your system crashes immediately and produces a dump file that points directly at the offending code. (Previously, a system dump might point to unrelated code that happened to be running later when the invalid situation was finally detected.)

If your system crashes while the MODS is turned on, then MODS has most likely done its job.

To make it easier to detect that this situation has occurred, the **crash** command has been extensively modified. The **stat** subcommand of **crash** now displays both:

- Whether the MODS (also called "xmalloc debug") has been turned on
- Whether this crash was the result of the MODS detecting an incorrect situation.

The **xmalloc** subcommand provides details on exactly what memory address (if any) was involved in the situation, and displays mini-tracebacks for the allocation and/or free of this memory.

Similarly, the **netm** command displays allocation and free records for memory allocated using the **net_malloc** kernel service (for example, **mbufs**, **mclusters**, etc.).

You can use these commands, as well as standard crash techniques, to determine exactly what went wrong.

MODS limitations

There are limitations to the Memory Overlay Detection System. Although it significantly improves your chances, MODS cannot detect all memory overlays. Also, turning MODS on has a small negative impact on overall system performance and causes somewhat more memory to be used in the kernel and the network memory heaps. If your system is running at full CPU utilization, or if you

are already near the maximums for kernel memory usage, turning on the MODS may cause performance degradation and/or system hangs.

Our practical experience with the MODS, however, is that the great majority of customers will be able to use it with minimal impact to their systems.

MODS benefits

You'll see these benefits from using the MODS:

- You can more easily test and debug your own kernel extensions and devicedrivers.
- Difficult problems that once required multiple attempts to recreate and debug them will generally require many fewer such attempts.

Appendix A. Alphabetical List of Kernel Services

This list is divided into parts based on the execution environment from which each kernel service can be called (see “Understanding Execution Environments” on page 7):

- “Kernel Services Available in Process and Interrupt Environments”
- “Kernel Services Available in the Process Environment Only” on page 588

“System Calls Available to Kernel Extensions” on page 31 lists the systems calls that can be called by kernel extensions.

Kernel Services Available in Process and Interrupt Environments

<code>add_domain_af</code>	Adds an address family to the Address Family domain switch table.
<code>add_input_type</code>	Adds a new input type to the Network Input table.
<code>add_netisr</code>	Adds a network software interrupt service to the Network Interrupt table.
<code>add_netopt</code>	Adds a network option structure to the list of network options.
<code>as_getsrval</code>	Obtains a handle to the virtual memory object for the specified address given in the specified address space.
<code>bdwrite</code>	Releases the specified buffer after marking it for delayed write.
<code>brelease</code>	Frees the specified buffer.
<code>_check_lock</code>	Conditionally updates a single word variable atomically, issuing an <i>import fence</i> for multiprocessor systems.
<code>_clear_lock</code>	Atomically writes a single word variable, issuing an <i>export fence</i> for multiprocessor systems.
<code>clrbuf</code>	Sets the memory for the specified buffer structure’s buffer to all zeros.
<code>clrjmpx</code>	Removes a saved context by popping the most recently saved jump buffer from the list of saved contexts.
<code>compare_and_swap</code>	Conditionally updates or returns a single word variable atomically.
<code>curtime</code>	Reads the current time into a time structure.
<code>d_align</code>	Assists in allocation of DMA buffers.
<code>d_cflush</code>	Flushes the processor and I/O controller (IOCC) data caches when using the long term DMA_WRITE_ONLY mapping of Direct Memory Access (DMA) buffers approach to bus device DMA.
<code>d_clear</code>	Frees a DMA channel.
<code>d_complete</code>	Cleans up after a DMA transfer.
<code>d_init</code>	Initializes a DMA channel.
<code>d_mask</code>	Disables a DMA channel.
<code>d_master</code>	Initializes a block-mode DMA transfer for a DMA master.

<code>d_move</code>	Provides consistent access to system memory that is accessed asynchronously by a device and by the processor on the system.
<code>d_roundup</code>	Assists in allocation of DMA buffers.
<code>d_slave</code>	Initializes a block-mode DMA transfer for a DMA slave.
<code>d_unmask</code>	Enables a DMA channel.
<code>del_domain_af</code>	Deletes an address family from the Address Family domain switch table.
<code>del_input_type</code>	Deletes an input type from the Network Input table.
<code>del_netisr</code>	Deletes a network software interrupt service routine from the Network Interrupt table.
<code>del_netopt</code>	Deletes a network option structure from the list of network options.
<code>devdump</code>	Calls a device driver dump-to-device routine.
<code>devstrat</code>	Calls a block device driver's strategy routine.
<code>devswqry</code>	Checks the status of a device switch entry in the device switch table.
<code>disable_lock</code>	Raises the interrupt priority, and locks a simple lock if necessary.
DTOM macro	Converts an address anywhere within an mbuf structure to the head of that mbuf structure.
<code>e_clear_wait</code>	Clears the wait condition for a kernel thread.
<code>e_wakeup</code> , <code>e_wakeup_one</code> , or <code>e_wakeup_w_result</code>	Notifies kernel threads waiting on a shared event of the event's occurrence.
<code>e_wakeup_w_sig</code>	Posts a signal to sleeping kernel threads.
<code>errsave</code> and <code>errlast</code>	Allows the kernel and kernel extensions to write to the error log.
<code>et_post</code>	Notifies a kernel thread of the occurrence of one or more events.
<code>fetch_and_add</code>	Increments a single word variable atomically.
<code>fetch_and_and</code> , <code>fetch_and_or</code>	Manipulates bits in a single word variable atomically.
<code>find_input_type</code>	Finds the given packet type in the Network Input Interface switch table and distributes the input packet according to the table entry for that type.
<code>getc</code>	Retrieves a character from a character list.
<code>getcb</code>	Removes the first buffer from a character list and returns the address of the removed buffer.
<code>getcbp</code>	Retrieves multiple characters from a character buffer and places them at a designated address.
<code>getcf</code>	Retrieves a free character buffer.
<code>getcx</code>	Returns the character at the end of a designated list.
<code>geterror</code>	Determines the completion status of the buffer.
<code>getexcept</code>	Allows kernel exception handlers to retrieve additional exception information.
<code>getpid</code>	Gets the process ID of the current process.
<code>i_disable</code>	Disables all of the interrupt levels at a particular interrupt priority and all interrupt levels at a less-favored interrupt priority.
<code>i_enable</code>	Enables all of the interrupt levels at a particular interrupt priority and all interrupt levels at a more-favored interrupt priority.
<code>i_mask</code>	Disables an interrupt level.

<code>i_reset</code>	Resets the system's hardware interrupt latches.
<code>i_sched</code>	Schedules off-level processing.
<code>i_unmask</code>	Enables an interrupt level.
<code>if_attach</code>	Adds a network interface to the network interface list.
<code>if_detach</code>	Deletes a network interface from the network interface list.
<code>if_down</code>	Marks an interface as down.
<code>if_nostat</code>	Zeros statistical elements of the interface array in preparation for an attach operation.
<code>ifa_ifwithaddr</code>	Locates an interface based on a complete address.
<code>ifa_ifwithdstaddr</code>	Locates the point-to-point interface with a given destination address.
<code>ifa_ifwithnet</code>	Locates an interface on a specific network.
<code>ifunit</code>	Returns a pointer to the <code>ifnet</code> structure of the requested interface.
<code>io_att</code>	Selects, allocates, and maps a region in the current address space for I/O access.
<code>io_det</code>	Unmaps and deallocates the region in the current address space at the given address.
<code>iodone</code>	Performs block I/O completion processing.
<code>kgethostname</code>	Retrieves the name of the current host.
<code>kgettickd</code>	Retrieves the current status of the systemwide time-of-day timer-adjustment values.
<code>ksettickd</code>	Sets the current status of the systemwide timer-adjustment values.
<code>kthread_kill</code>	Posts a signal to a specified kernel thread.
<code>loifp</code>	Returns the address of the software loopback interface structure.
<code>longjmpx</code>	Allows exception handling by causing execution to resume at the most recently saved context.
<code>looutput</code>	Sends data through a software loopback interface.
<code>m_adj</code>	Adjusts the size of an <code>mbuf</code> chain.
<code>m_cat</code>	Appends one <code>mbuf</code> chain to the end of another.
<code>m_clattach</code>	Allocates an <code>mbuf</code> structure and attaches an external cluster.
<code>m_clget</code> macro	Allocates a page-sized <code>mbuf</code> structure cluster.
<code>m_clgetm</code>	Allocates and attaches an external buffer.
<code>m_collapse</code>	Guarantees that an <code>mbuf</code> chain contains no more than a given number of <code>mbuf</code> structures.
<code>m_copy</code> macro	Creates a copy of all or part of a list of <code>mbuf</code> structures.
<code>m_copydata</code>	Copies data from an <code>mbuf</code> chain to a specified buffer.
<code>m_copym</code>	Creates a copy of all or part of a list of <code>mbuf</code> structures.
<code>m_free</code>	Frees an <code>mbuf</code> structure and any associated external storage area.
<code>m_freem</code>	Frees an entire <code>mbuf</code> chain.
<code>m_get</code>	Allocates a memory buffer from the <code>mbuf</code> pool.
<code>m_getclr</code>	Allocates and zeros a memory buffer from the <code>mbuf</code> pool.
<code>m_getclust</code> macro	Allocates an <code>mbuf</code> structure from the <code>mbuf</code> buffer pool and attaches a page-sized cluster.

m_getclustm	Allocates an mbuf structure from the mbuf buffer pool and attaches a cluster of the specified size.
m_gethdr	Allocates a header memory buffer from the mbuf pool.
M_HASCL macro	Determines if an mbuf structure has an attached cluster.
m_pullup	Adjusts an mbuf chain so that a given number of bytes is in contiguous memory in the data area of the head mbuf structure.
MTOCL macro	Converts a pointer to an mbuf structure to a pointer to the head of an attached cluster.
MTOD macro	Converts a pointer to an mbuf structure to a pointer to the data stored in that mbuf structure.
M_XMEMD macro	Returns the address of an mbuf cross-memory descriptor.
net_error	Handles errors for AIX communication network interface drivers.
net_start_done	Starts the done notification handler for AIX communications I/O device handlers.
net_wakeup	Wakes up all sleepers waiting on the specified wait channel.
net_xmit	Transmits data using an AIX communications I/O device handler.
net_xmit_trace	Traces transmit packets. This kernel service was added for those network interfaces that choose not to use the net_xmit kernel service to trace transmit packets.
panic	Crashes the system.
pci_cfgrw	Reads and writes PCI bus slot configuration registers.
pfctlinput	Invokes the ctlinput function for each configured protocol.
pffindproto	Returns the address of a protocol switch table entry.
pidsig	Sends a signal to a process.
pgsignal	Sends a signal to a process group.
pio_assist	Provides a standardized programmed I/O exception handling mechanism for all routines performing programmed I/O.
pm_planar_control	Controls power of a specified device on the planar.
pm_register_handle	Registers and unregisters Power Management handle.
pm_register_planar_control_handle	Registers and unregisters a planar control subroutine.
putc	Places a character at the end of a character list.
putcb	Places a character buffer at the end of a character list.
putcbp	Places several characters at the end of a character list.
putcf	Frees a specified buffer.
putcfl	Frees the specified list of buffers.
putcx	Places a character on a character list.
raw_input	Builds a raw_header structure for a packet and sends both to the raw protocol handler.
raw_usrreq	Implements user requests for raw protocols.

rtalloc	Allocates a route.
rtfree	Frees the routing table entry.
rtinit	Sets up a routing table entry, typically for a network interface.
rtredirect	Forces a routing table entry with the specified destination to go through the given gateway.
rtrequest	Carries out a request to change the routing table.
_safe_fetch	Atomically reads a single word variable, issuing an <i>import fence</i> for multiprocessor systems.
schednetisr	Schedules or invokes a network software interrupt service routine.
selnotify	Wakes up processes waiting in a poll or select subroutine or the fp_poll kernel service.
setjmpx	Allows saving the current execution state or context.
setpinit	Sets the parent of the current kernel process to the init process.
tfree	Deallocates a timer request block.
timeout	Schedules a function to be called after a specified interval.
thread_self	Returns the caller's kernel thread ID.
trcgenk	Records a trace event for a generic trace channel.
trcgenkt	Records a trace event, including a time stamp, for a generic trace channel.
tstart	Submits a timer request.
tstop	Cancels a pending timer request.
uexblock	Makes a kernel thread non-runnable when called from a user-mode exception handler.
uexclear	Makes a kernel thread blocked by the uexblock service runnable again.
unlock_enable	Unlocks a simple lock if necessary, and restores the interrupt priority.
unpin	Unpins the address range in system (kernel) address space.
unpinu	Unpins the specified address range in user or system memory.
untimeout	Cancels a pending timer request.
vm_att	Maps a specified virtual memory object to a region in the current address space.
vm_det	Unmaps and deallocates the region in the current address space that contains a given address.
xmdetach	Detaches from a user buffer used for cross-memory operations.
xmemdma	Prepares a page for DMA I/O or processes a page after DMA I/O is complete.
xmemin	Performs a cross-memory move by copying data from the specified address space to kernel global memory.
xmemout	Performs a cross-memory move by copying data from kernel global memory to a specified address space.
xmemunpin	Unpins the specified address range in user or system memory, given a valid cross-memory descriptor.

Kernel Services Available in the Process Environment Only

as_att	Selects, allocates, and maps a region in the specified address space for the specified virtual memory object.
as_det	Unmaps and deallocates a region in the specified address space that was mapped with the as_att kernel service.
as_geth	Obtains a handle to the virtual memory object for the specified address given in the specified address space. The virtual memory object is protected.
as_puth	Indicates that no more references will be made to a virtual memory object that was obtained using the as_geth kernel service.
as_seth	Maps a specified region in the specified address space for the specified virtual memory object
audit_svcbcopy	Appends event information to the current audit event buffer.
audit_svcfinis	Writes an audit record for a kernel service.
audit_svcstart	Initiates an audit record for a system call.
bawrite	Writes the specified buffer's data without waiting for I/O to complete.
bflush	Flushes all write-behind blocks on the specified device from the buffer cache.
binval	Invalidates all of a specified device's data in the buffer cache.
bindprocessor	Binds a process or thread to a processor.
blkflush	Flushes the specified block if it is in the buffer cache.
bread	Reads the specified block's data into a buffer.
breada	Reads in the specified block and then starts I/O on the read-ahead block.
bwrite	Writes the specified buffer's data.
cfgnadd	Registers a notification routine to be called when system-configurable variables are changed.
cfgndel	Removes a notification routine for receiving broadcasts of changes to system configurable variables.
copyin	Copies data between user and kernel memory.
copyinstr	Copies a character string (including the terminating NULL character) from user to kernel space.
copyout	Copies data between user and kernel memory.
creatp	Creates a new kernel process.
delay	Suspends the calling process for the specified number of timer ticks.
devswadd	Adds a device entry to the device switch table.
devswdel	Deletes a device driver entry from the device switch table.
dmp_add	Specifies data to be included in a system dump by adding an entry to the master dump table.
dmp_del	Deletes an entry from the master dump table.
dmp_prinit	Initializes the remote dump protocol.
e_assert_wait	Asserts that the calling thread is going to sleep.
e_block_thread	Blocks the calling thread.
e_sleep, e_sleepl, or e_sleep_thread	Forces the caller to wait for the occurrence of a shared event.
et_wait	Forces the caller to wait for the occurrence of an event.
fp_access	Checks for access permission to an open file.
fp_close	Closes a file.
fp_fstat	Gets the attributes of an open file.
fp_getdevno	Gets the device number and/or channel number for a device.
fp_getf	Retrieves a pointer to a file structure.

fp_hold	Increments the open count for a specified file pointer.
fp_ioctl	Issues a control command to an open device or file.
fp_llseek	Changes the current offset in an open file. Used to access offsets beyond 2GB.
fp_lseek	Changes the current offset in an open file.
fp_open	Opens a regular file or directory.
fp_opendev	Opens a device special file.
fp_poll	Checks the I/O status of multiple file pointers/descriptors and message queues.
fp_read	Performs a read on an open file with arguments passed.
fp_readv	Performs a read operation on an open file with arguments passed in iovec elements.
fp_rwuio	Performs read and write on an open file with arguments passed in a uio structure.
fp_select	Provides for cascaded, or redirected, support of the select or poll request.
fp_write	Performs a write operation on an open file with arguments passed.
fp_writev	Performs a write operation on an open file with arguments passed in iovec elements.
fubyte	Fetches, or retrieves, a byte of data from user memory.
fuword	Fetches, or retrieves, a word of data from user memory.
getadsp	Obtains a pointer to the current process's address space structure for use with the as_att and as_det kernel services.
getblk	Assigns a buffer to the specified block.
geteblk	Allocates a free buffer.
getppidx	Gets the parent process ID of the specified process.
getuerror	Allows kernel extensions to retrieve the current value of the ut_error field.
gfsadd	Adds a file system type to the gfs table.
gfsdel	Removes a file system type from the gfs table.
i_clear	Removes an interrupt handler from the system.
i_init	Defines an interrupt handler to the system, connects it to an interrupt level, and assigns an interrupt priority to the level.
init_heap	Initializes a new heap to be used with kernel memory management services.
initp	Changes the state of a kernel process from idle to ready.
iostadd	Registers an I/O statistics structure used for updating I/O statistics reported by the iostat subroutine.
iostdel	Removes the registration of an I/O statistics structure used for maintaining I/O statistics on a particular device.
iowait	Waits for block I/O completion.
kmod_entrypt	Returns a function pointer to a kernel module's entry point.
kmod_load	Loads an object file into the kernel or queries for an object file already loaded.
kmod_unload	Unloads a kernel object file.
kmsgctl	Provides message queue control operations.
kmsgget	Obtains a message queue identifier.
kmsgrcv	Reads a message from a message queue.
kmsgsnd	Sends a message using a previously defined message queue.
ksettimer	Sets the systemwide time-of-day timer.
kthread_start	Starts a previously created kernel-only thread.
limit_sigs	Changes the signal mask for the calling thread.
lock_alloc	Allocates memory for a simple or complex lock.
lock_clear_recursive	Prevents a complex lock from being acquired recursively.
lock_done	Releases a complex lock.
lock_free	Frees the memory of a simple or complex lock.

lock_init	Initializes a complex lock.
lock_islocked	Tests whether a complex lock is locked.
lock_mine	Checks whether a simple or complex lock is owned by the caller.
lock_read, lock_try_read	Locks a complex lock in shared-read mode.
lock_read_to_write, lock_try_read_to_write	Upgrades a complex lock from shared-read mode to exclusive-write mode.
lock_set_recursive	Prepares a complex lock for recursive use.
lock_write, lock_try_write	Locks a complex lock in exclusive-write mode.
lock_write_to_read	Downgrades a complex lock from exclusive-write mode to shared-read mode.
lockl	Locks a conventional process lock.
lookupvp	Retrieves the vnode that corresponds to the named path.
m_dereg	Deregisters expected mbuf structure usage.
m_reg	Registers expected mbuf usage.
net_attach	Opens a communications I/O device handler.
net_detach	Closes a communications I/O device handler.
net_sleep	Sleeps on the specified wait channel.
net_start	Starts network IDs on a communications I/O device handler.
NLuprintf	Submits a request to print an internationalized message to the controlling terminal of a process.
pin	Pins the address range in the system (kernel) space.
pincl	Manages the list of free character buffers.
pincode	Pins the code and data associated with an object file.
pinu	Pins the specified address range in user or system memory.
prochadd	Adds a systemwide process state-change notification routine.
prochdel	Deletes a process state change notification routine.
purbk	Invalidates a specified block's data in the buffer cache.
rusage_incr	Increments a field of the rusage structure.
setuerror	Allows kernel extensions to set the ut_error field in the u area.
sig_chk	Provides the calling kernel thread the ability to poll for receipt of signals.
sigsetmask	Changes the signal mask for the calling kernel thread.
simple_lock_init	Initializes a simple lock.
simple_lock, simple_lock_try	Locks a simple lock.
simple_unlock	Unlocks a simple lock.
sleep	Forces the calling kernel thread to wait on a specified channel.
subyte	Stores a byte of data in user memory.
suser	Determines the privilege state of a process.
suword	Stores a word of data in user memory.
talloc	Allocates a timer request block before starting a timer request.
thread_create	Creates a new process thread in the calling process.
thread_setsched	Sets process kernel thread scheduling parameters.
thread_terminate	Terminates the calling process kernel thread.
timeoutcf	Allocates or deallocates callout table entries for use with the timeout kernel service.
uexadd	Adds a systemwide exception handler for catching user-mode process exceptions.
uexdel	Deletes a previously added systemwide user-mode exception handler.
ufdcreate	Provides a file interface to kernel services.
uimove	Moves a block of data between kernel space and a space defined by a uio structure.

unlockl	Unlocks a conventional process lock.
unpincode	Unpins the code and data associated with an object file.
uprintf	Submits a request to print a message to the controlling terminal of a process.
uphysio	Performs character I/O for a block device using a uio structure.
ureadc	Writes a character to a buffer described by a uio structure.
uwritec	Retrieves a character from a buffer described by a uio structure.
vfsrele	Points to a virtual file system structure.
vm_cflush	Flushes the processor's cache for a specified address range.
vm_handle	Constructs a virtual memory handle for mapping a virtual memory object with specified access level.
vm_makep	Makes a page in client storage.
vm_mount	Adds a file system to the paging device table.
vm_move	Moves data between a virtual memory object and a buffer specified in the uio structure.
vm_protectp	Sets the page protection key for a page range.
vm_qmodify	Determines whether a mapped file has been changed.
vm_release	Releases virtual memory resources for the specified address range.
vm_releasep	Releases virtual memory resources for the specified page range.
vm_uiomove	Moves data between a virtual memory object and a buffer specified in the uio structure.
vm_umount	Removes a file system from the paging device table.
vm_write	Initiates page-out for a page range in the address space.
vm_writep	Initiates page-out for a page range in a virtual memory object.
vms_create	Creates a virtual memory object of the type and size and limits specified.
vms_delete	Deletes a virtual memory object.
vms_iowait	Waits for the completion of all page-out operations for pages in the virtual memory object.
vn_free	Frees a vnode previously allocated by the vn_get kernel service.
vn_get	Allocates a virtual node and inserts it into the list of vnodes for the designated virtual file system.
waitcfree	Checks the availability of a free character buffer.
w_clear	Removes a watchdog timer from the list of watchdog timers known to the kernel.
w_init	Registers a watchdog timer with the kernel.
w_start	Starts a watchdog timer.
w_stop	Stops a watchdog timer.
xmalloc	Allocates memory.
xmattach	Attaches to a user buffer for cross-memory operations.
xmempin	Pins the specified address range in user or system memory, given a valid cross-memory descriptor.
xmfree	Frees allocated memory.

Index

Numerics

- 64-bit 23
- kernel extension 23

A

- accented characters 188
- asynchronous I/O subsystem
 - AIX subroutines affected by 76
 - changing attributes in 76
 - subroutines 75
- ataide_buf structure (IDE) 291
- ATM LAN Emulation device driver 110
 - configuration parameters 113
 - entry points 118
 - trace and error logging 124
- ATM LANE
 - clients
 - adding 113
- attributes 94

B

- block (physical volumes) 192
- block device drivers
 - I/O kernel services 39
- block I/O buffer cache
 - managing 43
 - supporting user access to device drivers 42
 - using write routines 43
- block I/O buffer cache kernel services 40
- bootlist command
 - altering list of boot devices 97

C

- cfgmgr command
 - configuring devices 90, 97
- character I/O kernel services 40
- chdev command
 - changing device characteristics 97
 - configuring devices 90
- child devices 93
- clients
 - ATM LANE
 - adding 113
- commands
 - crash 303, 305, 334, 383, 388
 - errinstall 548
 - errlogger 553
 - errmsg 548
 - errpt 546, 553
 - errupdate 549, 552, 553
 - nm 377
 - sysdumpdev 301
 - sysdumpstart 302
 - trace 556
 - trcrpt 556, 561

- communications device handlers
 - common entry points 100
 - common status and exception codes 101
 - common status blocks 102
 - interface kernel services 63
 - kernel-mode interface 99
 - mbuf structures and 101
 - types
 - Ethernet 157
 - Fiber Distributed Data Interface (FDDI) 127
 - Forum Compliant ATM LAN Emulation 110
 - Multiprotocol (MPQP) 104
 - PCI Token-Ring High Performance 149
 - SOL (serial optical link) 108
 - Token-Ring (8fa2) 140
 - Token-Ring (8fc8) 131
 - user-mode interface 99
- communications I/O subsystem
 - physical device handler model 100
- compiling
 - when using the kernel debugger 377
- configuration
 - low function terminal interface 185
- cpu command
 - kernel debug program 345
- crash command 303, 305, 334, 383, 388
 - pcb subcommand 327
 - status subcommand 325
 - thread subcommand 328
- crash subcommands 312, 314
 - buf 307
 - buffer 307
 - callout 308
 - cm 308
 - cpu 309
 - dlock 310
 - ds 311
 - du 312
 - dump 312
 - file 312
 - fs 315
 - hide 315
 - inode 315
 - kfp 315, 329
 - knlist 316, 383
 - le 316, 383
 - linkblk 317
 - mblock 318
 - mbuf 318
 - mst 319
 - ndb 319
 - nm 316, 319
 - od 319, 334
 - ppd 320
 - print 321
 - proc 321
 - qrun 322

- crash subcommands 312, 314
 - (continued)
 - queue 322
 - quit 322
 - socket 324
 - stack 325
 - stat 325
 - status 325
 - stream 326
 - tcb 327
 - thread 328
 - trace 315
 - tty 329
 - unhide 329
 - user 329
 - vfs 330
 - vnode 330
 - xmalloc 331
- cross-memory kernel services 55

D

- DASD subsystem
 - device block level description 299
 - device block operation
 - cylinder 300
 - head 300
 - sector 299
 - track 300
- DDS 95
- debugger 301, 332, 390
- device attributes
 - accessing 94
 - modifying 95
- device configuration database
 - configuring 85
 - customized database 84
 - predefined database 84, 92
- device configuration manager
 - configuration hierarchy 85
 - configuration rules 86
 - device dependencies graph 86
 - device methods and 89
 - invoking 87
- device configuration subroutines 98
- device configuration subsystem 84, 85
 - adding unsupported devices 92
 - configuration commands 97
 - configuration database structure
 - components 80
 - device method level 84
 - high-level perspective 83
 - low-level perspective 84
 - configuration subroutines 98
- database configuration procedures 85
- device classifications 79
- device dependencies 93
- device types 87
- object classes in 87
- run-time configuration commands 90
- scope of support 79
- writing device methods for 88

- device dependent structure
 - format 96
 - updating
 - using the Change method 96
- device driver management kernel services 46
- device drivers
 - adding 93
 - device dependent structure 95
 - display 187
 - entry points 186
 - Fibre Channel Protocol (FCP) 269
 - interface 186
 - pseudo
 - low function terminal 186
- device methods
 - Change method and device dependent structure 96
 - Configure method and device dependent structure 95
 - for adding devices 92
 - for changing device states 90
 - for changing the database and not device state 91
 - interfaces to
 - configuration manager 89
 - run-time commands 90
 - invoking 88
 - method types 88
 - source code examples of 88
 - writing 88
- device states 90
- devices
 - child 93
 - dependencies 93
 - SCSI 207
- devnm command
 - naming devices 97
- diacritics 188
- diagnostics
 - low function terminal interface 188
- direct access storage device
 - subsystem 191
- display device driver 187
 - interface 187
- DMA management
 - accessing data in progress 45
 - hiding data 45
 - setting up DMA transfers 44
- DMA management kernel services 42
- dmp_add kernel service 303
- dmp_del kernel service 303
- dump 301, 332, 390
 - system 301

E

- entry points
 - communications physical device handler 100
 - device driver 186
 - IDE adapter device driver 294
 - IDE device driver 294
 - logical volume device driver 196
 - MPQP device handler 104
 - SCSI adapter device driver 227
 - SCSI device driver 227

- entry points (*continued*)
 - SOL device handler 108
- errinstall command 548
- errlogger command 553
- errmsg command 548
- error logging 546
 - adding logging calls 552
 - coding steps 548
- error record template 549
- errpt command 546, 553
- errsave kernel service 546, 552, 553
- errupdate command 549, 552, 553
- Ethernet device driver 157
 - asynchronous status 166
 - configuration parameters 159
 - device control operations 168
 - entry points 164
 - trace and error logging 171
- exception codes
 - communications device handlers 101
- exception handlers
 - implementing
 - in kernel-mode 20, 21, 22
 - in user-mode 23
- exception handling
 - interrupts and exceptions 18
 - modes
 - kernel 18
 - user 23
 - processing exceptions
 - basic requirements 19
 - default mechanism 18
 - kernel-mode 18
- exception management kernel services 63
- execution environments
 - interrupt 8
 - process 8

F

- FDDI device driver 127
 - configuration parameters 127
 - entry points 128
 - trace and error logging 129
- Fiber Distributed Data Interface device driver 127
- Fibre Channel Protocol (FCP) 269
- file descriptor 51
- file systems
 - logical file system 33
 - virtual file system 35
- files
 - /dev/error 546, 553
 - /dev/mem 305
 - /dev/systrctl 557, 559, 560
 - /etc/trcfmt 561, 577
 - /usr/adm/ras/trcfile 557
 - /usr/lib/boot/unix 305
 - sys/buf.h 307
 - sys/dump.h 303, 305
 - sys/erec.h 551
 - sys/err_rec.h 553
 - sys/errids.h 552
 - sys/file.h 312
 - sys/proc.h 301, 321
 - sys/socket.h 324

- files (*continued*)
 - sys/timer.h 308
 - sys/trcctl.h 560
 - sys/trchkid.h 565, 566, 577
 - sys/trcmacros.h 565
 - sys/tty.h 329
 - sys/user.h 301
 - sys/vfs.h 330
 - sys/vnode.h 331
- filesystem 33
- find command for the kernel debug program 348
- fine granularity timer services
 - coding the timer function 68
- Forum Compliant ATM LAN Emulation device driver 110

G

- g-nodes 35
- getattr subroutine
 - modifying attributes 95
- go command for kernel debug program 350
- graphic input device 179

H

- hardware interrupt kernel services 40
- hlpwatch 376

I

- I/O kernel services
 - block I/O 39
 - buffer cache 40
 - character I/O 40
 - DMA management 42
 - interrupt management 40
 - memory buffer (mbuf) 41
- iddebug 371, 372
- IDE subsystem
 - adapter device driver
 - entry points 294
 - ioctl commands 295
 - ioctl operations 296, 297, 298
 - performing dumps 294
 - device communication
 - initiator-mode support 286
 - error processing 294
 - error recovery
 - analyzing returned status 286
 - initiator mode 286
- IDE device driver
 - design requirements 293
 - entry points 294
 - internal commands 288
 - responsibilities relative to adapter device driver 285
- initiator I/O request execution
 - fragmented commands 290
 - gathered write commands 290
 - spanned or consolidated commands 288
- structures
 - ataide_buf structure 291

- IDB subsystem (*continued*)
 - typical adapter transaction sequence 287
- input device, subsystem 179
- input ring mechanism 186
- interface
 - low function terminal subsystem 185
- interrupt execution environment 8
- interrupt management
 - defining levels 44
 - setting priorities 44
- interrupt management kernel services 44
- interrupts
 - management services 40

K

- KDB Kernel Debugger 390
 - example files 542, 544, 546
 - introduction 390
 - subcommands 394
- kernel debug program 355
 - address origin, setting 354
 - address register
 - instruction, increasing 354
 - address register, instruction decreasing 342
 - breakpoints
 - clearing 344
 - listing 343
 - setting 342
 - skipping, restarting after 351
 - cpu command 345
 - create and change values 361
 - data screens, displaying 359
 - data storing 364
 - device drivers, displaying 347
 - displays
 - memory segments 360
 - displays contents of STREAMS 353
 - displays data structure 354
 - displays floating-point registers 349
 - displays internal file system tables 350
 - displays internal STREAMS 346
 - displays reason 357
 - displays STREAMS queues 356
 - displays structure of 64-bit process 353
 - ending the program session 357
 - error messages 388
 - floating point registers, displaying 349
 - formatted process tables, displaying 355
 - formatted trace buffers, displaying 370
 - fullwords, storing into memory 363
 - halfwords, storing 365
 - help screen, displaying 351
 - instruction address register
 - decreasing 342
 - increasing 354
 - loading 333
- kernel debug program 355 (*continued*)
 - memory
 - storing fullwords into 363
 - memory, displaying 345
 - memory location, altering 341
 - per-processor data, displaying 355
 - processor, switching 345
 - program, restarting 350
 - reboots machine 358
 - RS-232 port, switching 367
 - segment registers, displaying 362, 366
 - single-stepping instructions 365
 - stack traceback
 - formatted, tracing 364
 - starting 333
 - statistics
 - diplays 344
 - storage, searching 348
 - storing data 364
 - storing halfwords 365
 - system load list, displaying 352
 - thread command 369
 - thread table, displaying 368
 - timer request blocks, displaying 371
 - translating
 - virtual address to real address 376
 - tty command 372
 - tty structure
 - displaying 371
 - u-area, displaying 372
 - un
 - displays 372
 - user-defined variables, clearing 358
 - user-defined variables, displaying 375
 - user64 area, displaying 373
 - uthread command 374
 - uthread structure, displaying 373
 - variables
 - user-defined, clearing 358
 - variables, user-defined 375
 - virtual memory, displaying 375
- kernel debug program commands
 - alter 341
 - back 342
 - break 342
 - breaks 343
 - buckets 344
 - clear 344
 - cpu 345
 - create and change 361
 - display
 - displaying memory 345
 - internal STREAMS 346
 - displays
 - 64-bit process 353
 - contents of STREAMS 353
 - data structure 354
 - internal file system tables 350
 - internal STREAMS 349
 - memory segmetns 360
 - reason for entering debugger 357
 - STREAMS queues 356
 - drivers 347
 - find 348
- kernel debug program commands (*continued*)
 - float 349
 - go 350
 - help 351
 - loop 351
 - map 352
 - next 354
 - origin 354
 - proc 355
 - quit 357
 - reboots 358
 - reset 358
 - screen 359
 - sr64 362, 366
 - sregs 362
 - st 363
 - stack 364
 - stc 364
 - step 365
 - sth 365
 - swap 367
 - trace 370
 - trb 371
 - tty 371
 - un 372
 - user
 - displaying u-area 372
 - displaying user64 area 373
 - vars 375
 - vmm 375
 - watch 376
 - xlate 376
- kernel debugger
 - accessing global data 384
 - compiler listings 377
 - compiling options 377
 - displaying registers 386
 - entering 334
 - KDB 390
 - LLDB 332
 - map files 378
 - setting breakpoints 338, 381
 - stack trace 387
 - subcommands 335
 - breakpoints 338
 - dereferencing a pointer 337
 - expressions 337
 - reserved variables 336
 - variables 335
- kernel environment 1
 - base kernel services 2
 - creation of kernel processes 11
 - exception handling 18
 - exception processing in
 - exception and process kernel services 63
 - execution environments
 - interrupt 8
 - process 8
 - libraries
 - libcsys 6
 - libsys 7
 - loading system calls and kernel services 3
 - private routines 4

- kernel environment 1 (*continued*)
 - programming
 - kernel threads 9
- kernel environment, runtime 39
- kernel extension binding
 - adding symbols to the /unix name space 2
 - using existing libraries 5
 - using private routines 4
- kernel extension development
 - 64-bit 23
- kernel extension libraries
 - libcsys 6
 - libsys 7
- kernel extension management kernel services 46
- kernel extensions
 - accessing user-mode data
 - using cross-memory services 16
 - using data transfer services 15
 - interrupt priority
 - service times 44
 - loading and binding services 46
 - management services 47
 - serializing access to data structures 16
 - using with system calls 3
- kernel processes
 - accessing data from 12
 - comparison to user processes 11
 - creating 13
 - executing 13
 - handling exceptions 14
 - handling signals 14
 - obtaining cross-memory descriptors 13
 - preempting 14
 - terminating 13
 - using system calls 15
- kernel protection domain 11, 12
- kernel services 39, 46
 - categories
 - atomic operations 50, 51
 - complex locks 49
 - device driver management 46, 47
 - exception management 63
 - I/O 39, 40, 41, 42
 - kernel extension management 46, 47
 - lock allocation 48
 - locking 48
 - logical file system 51
 - memory 53, 54, 55
 - message queue 60
 - network 61, 62, 63
 - process level locks 50
 - process management 63
 - Reliability Availability Serviceability (RAS) 66
 - security 67
 - simple locks 48
 - time-of-day 67
 - timer 67, 68
 - virtual file system 69
 - dmp_add 303
 - dmp_del 303
 - errsave 546, 552, 553

- kernel services 39, 46 (*continued*)
 - loading
 - with system calls 3
 - multiprocessor-safe timer service 69
 - unloading
 - with system calls 4
- kernel threads
 - creating 10
 - executing 10
 - terminating 10

L

- lft 185
- LFT
 - accented characters 188
- libraries
 - libcsys 6
 - libsys 7
- LLDB Kernel Debugger 332
 - commands 338
- locking
 - conventional locks 16
 - kernel-mode strategy 17
 - serializing access to a predefined data structure and 16
- locking kernel services 48
- logical file system
 - component structure
 - file routines 34
 - system calls 34
 - v-nodes 35
 - file system role 33
- logical file system kernel services 51
- logical volume device driver
 - bottom half 196
 - data structures 195
 - physical device driver interface 198
 - pseudo-device driver role 195
 - top half 196
- logical volume manager
 - changing the mwcc_entries variable 200
 - DASD support 191
- logical volume subsystem
 - bad block processing 199
 - logical volume device driver 195
 - physical volumes
 - comparison with logical volumes 191
 - reserved sectors 192
- loop command for the kernel debug program 351
- loopback kernel services 62
- low function terminal
 - configuration commands 186
 - functional description 185
 - interface 185
 - components 186
 - configuration 185
 - device driver entry points 186
 - ioctl 186
 - terminal emulation 185
 - to display device drivers 186
 - to system keyboard 186
 - low function terminal interface
 - AIXwindow support 186

- low function terminal subsystem 185
 - accented characters supported 188
- lsattr command
 - displaying attribute characteristics of devices 97
- lscfg command
 - displaying device diagnostic information 98
- lsconn command
 - displaying device connections 97
- lsdev command
 - displaying device information 97
- lsparent command
 - displaying information about parent devices 97
- LVM 200

M

- macros
 - memory buffer (mbuf) 41
- management services
 - file descriptor 51
- map command 352
- mbuf structures
 - communications device handlers and 101
- memory buffer (mbuf) kernel services 41
- memory buffer (mbuf) macros 41
- memory kernel services
 - memory management 53
 - memory pinning 53
 - user memory access 54
- message queue kernel services 60
- Micro Channel adapters
 - displaying registers 386
- mirror write consistency cache (LVM)
 - changing the value 200
- mkdev command
 - adding devices to the system 97
 - configuring devices 90
- mknod command
 - creating special files 98
- MODS 579
- MPQP device handlers
 - binary synchronous communication
 - message types 104
 - receive errors 105
 - card description
 - hardware specifications 105
 - modem interfaces 107
 - physical interface mapping 108
 - port interfaces 106
 - entry points 104
 - multiprocessor-safe timer services, using 69
- Multiprotocol device handlers 104

N

- network kernel services
 - address family domain 61
 - communications device handler interface 63
 - interface address 62
 - loopback 62

network kernel services (*continued*)
 network interface device driver 61
 protocol 62
 routing 62
nm command 377

O

object data manager 92
ODM 92
odmadd command
 adding devices to predefined
 database 92
optical link device handlers 108

P

partition (physical volumes) 192
PCI Token-Ring Device Driver
 trace and error logging 155
PCI Token-Ring High Device Driver
 entry points 151
PCI Token-Ring High Performance
 configuration parameters 150
PCI Token-Ring High Performance
 Device Driver 149
performance tracing 301, 332, 390
physical volumes
 block 192
 comparison with logical volumes 191
 limitations 192
 partition 192
 reserved sectors 192
 sector layout 192
pinning
 memory 53
ppd command 355
predefined attributes object class
 accessing 94
 modifying 95
printer addition management subsystem
 adding a printer definition 204
 adding a printer formatter 205
 adding a printer type 203
 defining embedded references in
 attribute strings 205
 modifying printer attributes 204
printer formatter
 defining embedded references 205
printers
 unsupported types 203
private routines 4
process execution environment 8
process management kernel services 63
pseudo device driver
 low function terminal 186
putattr subroutine
 modifying attributes 95

R

RCM 187
referenced routines
 for memory pinning 59
 to support address space
 operations 59

referenced routines (*continued*)
 to support cross-memory
 operations 59
 to support pager back ends 59
Reliability Availability Serviceability
(RAS) kernel services 66
rendering context manager 186, 187
restbase command
 restoring customized information to
 configuration database 97
rmdev command
 configuring devices 90
 removing devices from the
 system 97
runtime kernel environment 39

S

sample code
 trace format file 570
savebase command
 saving customized information to
 configuration database 97
sc_buf structure (SCSI) 218
SCSI subsystem
 adapter device driver
 entry points 227
 ioctl commands 235, 237, 239
 ioctl operations 232, 236, 237, 238,
 239
 performing dumps 228
 responsibilities relative to SCSI
 device driver 207
 asynchronous event handling 209
 command tag queuing 218
 device communication
 initiator-mode support 208
 target-mode support 208
 error processing 227
 error recovery
 initiator mode 211
 target mode 214
 initiator I/O request execution
 fragmented commands 216
 gathered write commands 217
 spanned or consolidated
 commands 215
 initiator-mode adapter transaction
 sequence 214
SCSI device driver
 asynchronous event-handling
 routine 210
 closing a device 227
 design requirements 223
 entry points 227
 internal commands 215
 responsibilities relative to adapter
 device driver 207
 using openx subroutine
 options 223
structures
 sc_buf structure 218
 tm_buf structure 227, 232
 target-mode interface 229, 231, 233
 interaction with initiator-mode
 interface 229
security kernel services 67
serial optical link device handlers 108

Small Computer Systems Interface
 subsystem 207
SOL device handlers
 changing device attributes 110
 configuring physical and logical
 devices 109
 entry points 108
 special files interfaces 108
 status and exception codes 101
 status blocks
 communications device handler
 CIO_ASYNC_STATUS 103
 CIO_HALT_DONE 102
 CIO_LOST_STATUS 103
 CIO_NULL_BLK 103
 CIO_START_DONE 102
 CIO_TX_DONE 103
 communications device handlers
 and 102
 status codes
 communications device handlers
 and 101
 storage 191
 stream-based tty subsystem 185
 subroutines
 sysconfig 382
 subsystem
 low function terminal 185
 streams-based tty 185
 subsystem
 graphic input device 179
 sysconfig subroutine 382
sysdumpdev command 301
sysdumpstart command 302
system calls
 accessing kernel data from
 data types 27
 passing parameters 27
 error information 31
 exception handling
 alternatives 30
 default mechanism 29
 extending the kernel with
 call handler actions 26
 execution 25
 in kernel protection domain 26
 in user protection domain 26
 loading
 with kernel services 3
 nesting for kernel-mode use 30
 preempting 28
 response to page faulting 30
 services for all kernel extensions 31
 services for kernel processes only 32
 signal handling in
 asynchronous signals 29
 setjmpx kernel service 29
 signal delivery 28
 stacking saved contexts 29
 wait termination 29
 unloading
 with kernel services 4
 using with kernel extensions 3
system dump 301
formatting 304
including device driver
 information 302

- system dump 301 (*continued*)
 - initiating 301
- system dumpcomponents
 - component dump table 302
 - master dump table 302

T

- terminal emulation
 - low function terminal 185
- thread command
 - kernel debug program 369
- time-of-day kernel services 67
- timer kernel services
 - coding the timer function 69
 - compatibility 68
 - determining the timer service to use 68
 - fine granularity 67
 - reading time into time structure 68
 - watchdog 68
- timer service
 - multiprocessor-safe, using 69
- tm_buf structure (SCSI) 227
- Token-Ring (8fa2) device driver 140, 141
 - configuration parameters 141
 - trace and error logging 147
- Token-Ring (8fc8) device 131
- Token-Ring (8fc8) device driver 132
 - configuration parameters 132
 - trace and error logging 138
- trace command 556
- trace events
 - defining 564
 - event IDs 566
 - determining location of 566
 - format file example 570
 - format file stanzas 568
 - forms 564
 - macros 565
- trace report
 - filtering 578
 - producing 561
 - reading 578
- tracing 554
 - configuring 556
 - controlling 559
 - starting 554, 556
- trcrpt command 556, 561
- tty command
 - kernel debug program 372

U

- user commands
 - configuration 186
- uthread command
 - kernel debug program 374

V

- v-nodes 35
- virtual file system 33
 - configuring 38
 - data structures 36, 37
 - file system role 35

- virtual file system 33 (*continued*)
 - generic nodes (g-nodes) 35
 - header files 37
 - interface requirements 36
 - mount points 35
 - virtual nodes (v-nodes) 35
- virtual file system kernel services 69
- virtual memory management
 - addressing data 57
 - data flushing 57
 - discarding data 57
 - executing data 58
 - installing pager backends 58
 - moving data to or from an object 57
 - objects 56
 - protecting data 58
 - referenced routines
 - for manipulating objects 58, 59
 - virtual memory manager 56
- virtual memory management kernel services 54
- vm_uiomove 55, 57, 59, 591

Vos remarques sur ce document / Technical publication remark form

Titre / Title : Bull Kernel Extensions and Device Support Programming Concepts

N° Référence / Reference N° : 86 A2 36JX 02

Daté / Dated : November 1999

ERREURS DETECTEES / ERRORS IN PUBLICATION

AMELIORATIONS SUGGEREES / SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Vos remarques et suggestions seront examinées attentivement.

Si vous désirez une réponse écrite, veuillez indiquer ci-après votre adresse postale complète.

Your comments will be promptly investigated by qualified technical personnel and action will be taken as required.

If you require a written reply, please furnish your complete mailing address below.

NOM / NAME : _____ Date : _____

SOCIETE / COMPANY : _____

ADRESSE / ADDRESS : _____

Remettez cet imprimé à un responsable BULL ou envoyez-le directement à :

Please give this technical publication remark form to your BULL representative or mail to:

**BULL ELECTRONICS ANGERS
CEDOC
34 Rue du Nid de Pie – BP 428
49004 ANGERS CEDEX 01
FRANCE**

Technical Publications Ordering Form

Bon de Commande de Documents Techniques

To order additional publications, please fill up a copy of this form and send it via mail to:

Pour commander des documents techniques, remplissez une copie de ce formulaire et envoyez-la à :

BULL ELECTRONICS ANGERS
CEDOC
ATTN / MME DUMOULIN
34 Rue du Nid de Pie – BP 428
49004 ANGERS CEDEX 01
FRANCE

Managers / Gestionnaires :
Mrs. / Mme : **C. DUMOULIN** +33 (0) 2 41 73 76 65
Mr. / M : **L. CHERUBIN** +33 (0) 2 41 73 63 96
FAX : +33 (0) 2 41 73 60 19
E-Mail / Courrier Electronique : srv.Cedoc@franp.bull.fr

Or visit our web site at: / Ou visitez notre site web à:

<http://www-frec.bull.com> (Press Room, Technical Literature, Ordering Publications)

CEDOC Reference # N° Référence CEDOC	Qty Qté	CEDOC Reference # N° Référence CEDOC	Qty Qté	CEDOC Reference # N° Référence CEDOC	Qty Qté
__ __ - - - - - [__]		__ __ - - - - - [__]		__ - - - - - [__]	
__ __ - - - - - [__]		__ - - - - - [__]		__ - - - - - [__]	
__ __ - - - - - [__]		__ - - - - - [__]		__ - - - - - [__]	
__ __ - - - - - [__]		__ - - - - - [__]		__ - - - - - [__]	
__ __ - - - - - [__]		__ - - - - - [__]		__ - - - - - [__]	
__ __ - - - - - [__]		__ - - - - - [__]		__ - - - - - [__]	
__ __ - - - - - [__]		__ - - - - - [__]		__ - - - - - [__]	

[__]: **no revision number means latest revision** / pas de numéro de révision signifie révision la plus récente

NOM / NAME : _____ Date : _____

SOCIETE / COMPANY : _____

ADRESSE / ADDRESS : _____

PHONE / TELEPHONE : _____ FAX : _____

E-MAIL : _____

For Bull Subsidiaries / Pour les Filiales Bull :

Identification: _____

For Bull Affiliated Customers / Pour les Clients Affiliés Bull :

Customer Code / Code Client : _____

For Bull Internal Customers / Pour les Clients Internes Bull :

Budgetary Section / Section Budgétaire : _____

For Others / Pour les Autres :

Please ask your Bull representative. / Merci de demander à votre contact Bull.

BULL ELECTRONICS ANGERS
CEDOC
34 Rue du Nid de Pie – BP 428
49004 ANGERS CEDEX 01
FRANCE

ORDER REFERENCE
86 A2 36JX 02

PLACE BAR CODE IN LOWER
LEFT CORNER



Utiliser les marques de découpe pour obtenir les étiquettes.
Use the cut marks to get the labels.

Kernel Extensions
and Device
Support
Programming
Concepts

86 A2 36JX 02

Kernel Extensions
and Device
Support
Programming
Concepts

86 A2 36JX 02

Kernel Extensions
and Device
Support
Programming
Concepts

86 A2 36JX 02