IRIX™ Device Driver Programmer's Guide

Document Number 007-0911-060

CONTRIBUTORS

Written by David Cortesi
Illustrated by Dany Galgani
Edited by Christina Cary
Production by Gloria Ackley
Significant engineering contributions by (in alphabetical order): Rich Altmaier, Brad Eacker, Ben Fathi, Steve Haehnichen, Bruce Johnson, Tom Lawrence, Ben Mahjoor, Charles Marker, Dave Olson, Bhanu Prakash, James Putnam, Sarah Rosedahl, Deepinder Setia, Adam Sweeney, Daniel Yau.
Beta test contributions by: Jeff Stromberg of GeneSys
Cover design and illustration by Rob Aguilar, Rikk Carey, Dean Hodgkinson, Erik Lindholm, and Kay Maitz

IRIX™ Device Driver Programmer's Guide
Document Number 007-0911-060

Contents

List of Examples xxvii

List of Figures xxix

List of Tables xxxi

About This Guide xxxv
Audience xxxvi
What This Guide Contains xxxvi
Other Sources of Information xxxvii
    Developer Program xxxvii
    Internet Resources xxxvii
    Standards Documents xxxviii
    Important Reference Pages xxxviii
    Additional Reading xxxix
Conventions Used in This Guide xl

PART I IRIX Device Integration

1. Physical and Virtual Memory 3
Physical Address Space 4
    Device Addresses 4
    Memory Addresses 5

Addresses of Memory and Devices 5
  CPU Modules 5
    Interrogating the CPU Type 6
  CPU Access to Memory 6
  Processor Operating Modes 8
  Virtual Address Mapping 8
    TLB Misses and TLB Sizes 8
  Address Space Creation 9
  Address Exceptions 9
  CPU Access to Device Registers 10
  Direct Memory Access 11
  Bus Virtual Addresses 12
  Cache Use and Cache Coherency 13
    Cache Coherency in Multiprocessors 14
    Cache Coherency in Uniprocessors 14
The 32-Bit Address Space 14
  Segments of the 32-bit Address Space 15
  Virtual Address Mapping 16
  User Process Space—kuseg 16
  Kernel Virtual Space—kseg2 17
  Cached Physical Memory—kseg0 17
  Uncached Physical Memory—kseg1 18
The 64-Bit Address Space 18
  Segments of the 64-Bit Address Space 18
  Compatibility of 32-Bit and 64-Bit Spaces 20
  Virtual Address Mapping 20
  User Process Space—xkuseg 21
  Supervisor Mode Space—xksseg 22
  Kernel Virtual Space—xkseg 22
  Cache-Controlled Physical Memory—xkphys 22

Device Driver Use of Memory 24
Allowing for 64-Bit Mode 24
Memory Use in User-Level Drivers 25
Access Using a Device Model 25
Access Using mmap() 26
Mapped Access Provided by a Device Driver 26
Memory Use in Kernel-Level Drivers 26
Uncached Memory Access in the Challenge and Onyx Series 27
Uncached Memory Access in the IP26 CPU 27

2. Device Configuration 29
Hardware Inventory 29
Using the Hardware Inventory 29
Contents of the Inventory 30
Displaying the Inventory with hinv 30
Testing the Inventory In Software 30
Creating an Inventory Entry 32
Device Special Files 32
Device Representation 33
Block Versus Character 33
Major Device Number 34
Minor Device Number 35
Defining Device Names 35
IRIX Conventional Device Names 36
The Script MAKEDEV 36
Making Device Files 37
Multiple Names for One Device 37
Configuration Files 38
Master Configuration Database 38
System Configuration Files 39
System Tuning Parameters 39
X Display Manager Configuration 40

3. Device Control Software 41

User-Level Device Control 41

EISA Mapping Support 42

VME Mapping Support 42

User-Level DMA From the VME Bus 43

User-Level Control of SCSI Devices 43

Managing External Interrupts 43

User-Level Interrupt Management 44

Memory-Mapped Access to Serial Ports 45

Kernel-Level Device Control 45

Kinds of Kernel-Level Drivers 45

Typical Driver Operations 46

Overview of Device Open 46

Overview of Device Control 48

Overview of Programmed Kernel I/O 49

Overview of Memory Mapping 50

Overview of DMA I/O 52

Upper and Lower Halves 54

Driver Upper Half 54

Driver Lower Half 54

Relationship Between Halves 55

Layered Drivers 55

Combined Block and Character Drivers 55

Drivers for Multiprocessors 56

Loadable Drivers 57

PART II Device Control From Process Space

4. User-Level Access to VME and EISA 61

VME Programmed I/O 62

Mapping a VME Device Into Process Address Space 62

Learning VME Device Addresses 62

Opening a Device Special File 63

Using the mmap() Function 64

Map Size Limits 64

VME PIO Access 65

VME PIO Bandwidth 66

EISA Programmed I/O 67

Mapping an EISA Device Into Memory 67

Learning EISA Device Addresses 67

Opening a Device Special File 68

Using the mmap() Function 68

EISA PIO Bandwidth 69

VME User-Level DMA 70

Using the udmalib Functions 71

Buffer Allocation for User DMA 71

Allocation of Descriptors 72

Advantages of User DMA 73

DMA Engine Bandwidth 73

Example User DMA Function 74

5. User-Level Access to SCSI Devices 77

Overview of the dsreq Driver 78

Generic SCSI Device Special Files 78

Form of Names in /dev/scsi 79

Names of SCSI Devices on a SCSI Bus 79

Names of SCSI Devices on the Jag (VME Bus) Controller 80

Major and Minor Device Numbers in /dev/scsi 80

Creating Additional Names in /dev/scsi 81

Relationship to Other Device Special Files 81

The dsreq Structure 82
    Values for ds_flags 83
    Data Transfer Options 85
    Return Codes and Status Values 85
Testing the Driver Configuration 88
Using the Special DS_RESET and DS_ABORT Calls 89
    Using DS_ABORT 89
    Using DS_RESET 90
Using dslib Functions 90
    dslib Functions 90
    Using dsopen() and dsclose() 92
    Issuing a Request With doscsireq() 93
    SCSI Utility Functions 93
        Using filldsreq() 93
        Using fillg0cmd() and fillg1cmd() 94
        Using ds_vtostr() and ds_ctostr() 95
    Using Command-Building Functions 96
        inquiry12()—Issue an Inquiry Command 96
        modeselect15()—Issue a Group 0 Mode Select Command 96
        modesense1a()—Send a Group 0 Mode Sense Command 97
        read08() and readextended28()—Issue a Read Command 98
        readcapacity25()—Issue a Read Capacity Command 99
        requestsense03()—Issue a Request Sense Command 100
        reserveunit16() and releaseunit17()—Control Logical Units 100
        senddiagnostic1d()—Issue a Send Diagnostic Command 101
        testunitready00—Issue a Test Unit Ready Command 102
        write0a() and writeextended2a()—Issue a Write Command 102
Example dslib Program 103

6. **Control of External Interrupts** 113
External Interrupts in Challenge and Onyx Systems 114
Generating Outgoing Signals 114
Receiving Incoming External Interrupts 115
Detecting Invalid External Interrupts 116
Setting the Expected Pulse Width 117
Setting the Stuck Pulse Width 118
Receiving Interrupts 118

7. **User-Level Interrupts** 121
Overview of ULI 121
The User Level Interrupt Handler 122
Restrictions on the ULI Handler 122
Planning for Concurrency 123
Debugging With Interrupts 123
Declaring Global Variables 123
Using Multiple Devices 124
Setting Up 124
Opening the Device Special File 124
Locking the Program Address Space 125
Registering the Interrupt Handler 126
Registering an External Interrupt Handler 127
Registering a VME Interrupt Handler 127
Interacting With the Handler 128
Achieving Mutual Exclusion 129
Sample Program 129

PART III Kernel-Level Drivers

8. Structure of a Kernel-Level Driver 135

Summary of Driver Structure 136

Entry Point Naming and lboot 136

Driver Name Prefix 136

Kernel Switch Tables 137

Entry Point Summary 138

Driver Flag Constant 140

Flag D_MP 141

Flag D_WBACK 141

Flag D_MT 142

Flag D_OLD 142

Initialization Entry Points 143

When Initialization Is Performed 143

Initialization of Loadable Drivers 143

Entry Point init() 144

Entry Point edtinit() 144

Entry Point start() 145

Open and Close Entry Points 146

Entry Point open() 146

Use of the Device Number 147

Use of the Open Type 147

Use of the Open Flag 148

Use of the cred_t Object 148

Saving the Size of a Block Device 149

Saving the User ABI 149

Entry Point close() 149

Control Entry Point 150

Choosing the Command Numbers 151

Supporting 32-Bit and 64-Bit Callers 151

User Return Value 151

Data Transfer Entry Points 151
  Entry Points read() and write() 152
    Data Transfer for a PIO Device 152
    Calling Entry Point strategy() From Entry Point read() or write() 153
  Entry Point strategy() 154
Poll Entry Point 155
  Use and Operation of poll(2) 155
    Use of pollwakeup() 156
    Use of pollwakeup() Without Interrupts 156
  Entry Point poll() 156
Memory Map Entry Points 158
  Concepts and Use of mmap() 158
    Use of mmap() 158
    Persistent Mappings 159
  Entry Point map() 159
  Entry Point mmap() 161
  Entry Point unmap() 162
Interrupt Entry Point 163
  Associating Interrupt to Driver 163
  Entry Point intr() 164
    Mutual Exclusion 165
    Performance and Latency 165
    Completing Block I/O 166
    Completing Character I/O 166
    Calling pollwakeup() 166
Support Entry Points 167
  Entry Point unload() 167
  Entry Point halt() 168
  Entry Point size() 169
  Entry Point print() 169

Handling 32-Bit and 64-Bit Execution Models 170

Planning for Multiprocessor Use 171

The Multiprocessor Environment 171

Uniprocessor Assumptions 171

Protecting Common Data 172

Sleeping and Waking 173

Synchronizing Within Upper-Half Functions 173

Running on CPU 0 174

Serializing on a Single Lock 174

Serializing on a Lock Per Device 174

Coordinating Upper-Half and Interrupt Entry Points 175

Coordinating Through the buf_t 175

Coordination in a Character Driver 175

Converting a Uniprocessor Driver 176

Example Conversion Problem 176

9. Device Driver/Kernel Interface 179

Important Data Types 180

The Device Number Types 180

Use of the Device Numbers 180

Device Number Functions 181

External and Internal Numbers 181

Structure uio_t 182

Data Location and the iovec_t 182

Use of the uio_t 183

Structure buf_t 183

Fields of buf_t 183

Using the Logical Block Number 184

Buffer Location and b_flags 184

Lock and Semaphore Types 185

Important Header Files 185
Memory Allocation 187
  General-Purpose Allocation 187
  Allocating Objects of Specific Kinds 189
    Allocating pollhead Objects 189
    Allocating Semaphores and Locks 189
    Allocating buf_t Objects and Buffers 190
  Suballocation Functions 191
Transferring Data 192
  General Data Transfer 192
    Block Copy Functions 193
    Byte and Word Functions 194
  Transferring Data Through a uio_t Object 194
Managing Virtual and Physical Addresses 195
  Testing Device Physical Addresses 195
  Managing Mapped Memory 196
  Working With Page and Sector Units 197
  Setting Up a DMA Transfer 198
    Converting Physical Addresses 199
    Managing Buffer Virtual Addresses 199
    Managing Memory for Cache Coherency 200
    DMA Buffer Alignment 201
    Maximum DMA Transfer Size 202
User Process Administration 202
  Sending a Process Signal 203

Waiting and Mutual Exclusion 204
Mutual Exclusion Compared to Waiting 204
Basic Locks 205
Long-Term Locks 207
Using Mutex Locks 207
Using Sleep Locks 209
Reader/Writer Locks 211
Priority Level Functions 213
Waiting for Time to Pass 214
Time Units 215
Timer Support 215
Short-Term Delay Support 216
Waiting for Memory to Become Available 216
Waiting for Block I/O to Complete 217
How the strategy() Entry Point Is Called 217
Strategies of the strategy() Entry Point 217
Waiting for a General Event 218
Using sleep() and wakeup() 219
Using Synchronization Variables 220
Semaphores 222
Using a Semaphore for Mutual Exclusion 223
Using a Semaphore for Waiting 224

10. Building and Installing a Driver 225
Defining Device Numbers 226
Selecting a Major Number 226
Selecting Minor Numbers 227
Defining Device Special Files 227
Static Definition of Device Special Files 227
Dynamic Definition of Device Special Files 227

Compiling and Linking 228
    Using /var/sysgen/Makefile.kernio 228
    Compiler Variables 229
    Compile Options, 32-Bit Kernel 230
    Compile Options, 64-Bit Kernel 231
Configuring a Nonloadable Driver 232
    How Names Are Used in Configuration 232
    Placing the Object File in /var/sysgen/boot 233
    Describing the Driver in /var/sysgen/master.d 233
        Descriptive Line 233
        Listing Dependencies 234
        Stubs Section 235
        Variables Section 235
    Configuring a Kernel 236
    Generating a Kernel 237
Configuring a Loadable Driver 237
    Public Global Variables 237
    Compile Options for Loadable Drivers 238
    Master File for Loadable Drivers 238
        Descriptive Line 238
    Registration 239
    Loading 239
    Unloading 240
Configuring for a Dynamic Major Number 241

11. Testing and Debugging a Driver 243
Preparing the System for Debugging 243
Placing symmon in the Volume Header 244
Enabling Debugging in irix.sm 245
Including Symbols in the Kernel Image 245
Including idbg in the Kernel Image 246
Including Lock Metering in the Kernel Image 246
Generating a Debugging Kernel 247
Specifying a Separate System Console 247
Verifying the Debugging Tools 248
Producing Diagnostic Displays 249
Using cmn_err 249
Displaying to the System Log 249
Displaying to the Circular Message Buffer 250
Using cmn_err() Through Macros 250
Using printf() 251
Using ASSERT 252
Using symmon 252
How symmon Is Entered 253
Using symmon in a Uniprocessor Workstation 253
Using symmon in a Multiprocessor Workstation 253
Entering symmon at Boot Time 254
Commands of symmon 255
Syntax of Command Elements 255
Commands for Symbol Conversion and Lookup 256
Commands to Control Execution Flow 257
Commands to Manage Virtual Memory 259
Commands to Display Memory 260
Utility Commands 261

Using idbg 262
  Loading and Invoking idbg 262
    Invoking idbg for Interactive Use 262
    Invoking idbg with a Log File 263
    Invoking idbg for a Single Command 263
  Commands of idbg 264
  Commands to Display Memory and Symbols 265
  Commands to Display Process Information 265
  Commands to Display Locks and Semaphores 267
  Commands to Display I/O Status 267
  Commands to Display buf_t Objects 268
  Commands to Display STREAMS Structures 268
  Commands to Display Network-Related Structures 269
Using icrash 269

12. Driver Example 271
Installing the Example Driver 271
  Obtaining the Source Files 272
  Compiling the Example Driver 272
  Configuring the Example Driver 273
  Creating Device Special Files 275
  Verifying Driver Operation 275
Example Driver Source Files 278
  Descriptive File 278
  System File 279
  Header File 279
  Source File 281

PART IV VME Device Drivers

13. VME Device Attachment 297

Overview of the VME Bus 298
- VME History 298
- VME Features 298
  - VME Address Spaces 299
  - Master and Slave Devices 299
  - VME Transactions 300

VME Bus in Silicon Graphics Systems 300
- The VME Bus Controller 301
- VME PIO Operations 302
- VME DMA Operations 302
- Operation of the DMA Engine 303

VME Bus Addresses and System Addresses 304
- User-Level and Kernel-Level Addressing 304
- PIO Addressing and DMA Addressing 305
- PIO Addressing in Challenge and Onyx Systems 306
- PIO Addressing in the Crimson Series 306
  - Crimson Mapping of A16 Space 307
  - Crimson Mapping of A24 Space 307
  - Crimson Mapping of A32 Space 307
- DMA Addressing 308
  - DMA Addressing in the Challenge and Onyx Systems 308
  - Direct DMA Addressing in the Crimson Series 308
  - Mapped DMA Addressing in the Crimson Series 309

Configuring VME Devices 310
- Configuring Device Addresses 310
- Configuring the System Files 311
  - Coding the VECTOR Statement 311
  - Using the IPL Statement 312

VME Hardware in Challenge and Onyx Systems 313
VME Hardware Architecture 313
Main System Bus 313
Ibus 314
Bus Interfacing 314
Maximum Latency 314
VME Bus Numbering 315
VMEbus Channel Adapter Module (VCAM) Board 315
VMECC 315
F Controller ASIC 317
VMEbus Interrupt Generation 317
VME Interface Features and Restrictions 318
DMA Multiple Address Mapping 319
VME Interrupt Priority 321
VME Hardware Features and Restrictions 321
Designing a VME Bus Master for Challenge and Onyx Systems 322

14. Services for VME Drivers 325
Kernel Services for VME 325
Mapping PIO Addresses 325
Testing the PIO Map 327
Using the Mapped Address 328
Using the PIO Map in Functions 328
Fixed PIO Maps 328
Unfixed PIO Maps 329
Mapping DMA Addresses 330
Using a DMA Map 331
Allocating an Interrupt Vector Dynamically 332
Allocating a Vector 333
Releasing a Vector 334
Vector Errors 334
Supporting Early IO4 Cache Problems 334
Sample VME Device Driver 334

PART V SCSI Device Drivers

15. SCSI Device Drivers 351

SCSI Support in Silicon Graphics Systems 352

- SCSI Hardware Support 352
- IRIX Kernel SCSI Support 353
  - Host Adapter Drivers 353
  - SCSI Device Drivers 354

Host Adapter Facilities 354

- Purpose of the Host Adapter Driver 354
- Host Adapter Concepts 355
  - Target Numbers 356
  - Logical Unit Numbers (LUNs) 356
- Overview of Host Adapter Functions 357
- How the Host Adapter Functions Are Found 358
  - Using the Function Vector Tables 358
  - Learning the Adapter Type Number 359
  - Learning the Adapter Number 359
- Using scsi_info() 360
- Using scsi_alloc() 361
- Using scsi_free() 362
- Using scsi_command() 363
  - Input to scsi_command() 363
  - Command Execution 365
  - Values Returned in a scsi_request Structure 366
- Using scsi_abort() 368
- Using scsi_reset() 369

Designing a SCSI Driver 369

- SCSI Driver Initialization 369
- Opening a SCSI Device 370
- Accessing a SCSI Device 370
- Configuring a SCSI Driver 370

Example SCSI Device Driver 370
Designing a Host Adapter Driver 375
Overview of Host Adapter Driver Architecture 375
Host Adapter Initialization 375
Initializing the Hardware 376
Acquiring a Type Number 376
Storing Entry Point Addresses 376
SCSI Reference Data 377
SCSI Error Messages 377
SCSI Error Message Tables 378
Adapter Error Codes (Table scsi_adaperrs_tab) 378
SCSI Sense Codes (Table scsi_key_msgtab) 379
Additional Sense Codes (Table scsi_addit_msgtab) 380
WD93 States and Phases 384

PART VI Network Drivers

16. Network Device Drivers 389
Overview of Network Drivers 390
Application Interfaces 391
Protocol Stack Interfaces 391
Device Driver Interfaces 392
Network Driver Interfaces 392
Kernel Facilities 393
Principal ifnet Header Files 393
Debugging Facilities 394
Information Sources 394
Network Inventory Entries 396

Multiprocessor Considerations 396
Ineffective spl() Functions 397
Multiprocessor Locking Macros 397
Compilation Flags for MP TCP/IP 397
Mutual Exclusion Macros 398
Macro Use 399
Input Queueing Example 399
Interrupt Handler Example 400
Example ifnet Driver 401

PART VII EISA Drivers

17. EISA Device Drivers 427
The EISA Bus in Silicon Graphics Systems 428
EISA Bus Overview 428
EISA Request Arbitration 430
EISA Interrupts 430
EISA Data Transfers 430
EISA Address Spaces 430
EISA Locked Cycles 431
EISA Byte Ordering 431
EISA Product Identifier 431
EISA Support in Indigo^2 and Challenge M Series 434
Available Card Slots 434
EISA Address Mapping 434
Interrupt Priority Scheduling 435
EISA Configuration 435
Configuring the Hardware 435
Configuring IRIX 436
Using the iospace Parameters 437
Using the probe and exprobe Parameters 437
Using the module Parameter 438

Kernel Functions for EISA Support 439
Mapping PIO Addresses 439
Testing the PIO Map 441
Using the Mapped Address 441
Using the PIO Map in Functions 441
Allocating IRQs and Channels 442
Allocating and Programming an IRQ 442
Allocating a DMA Channel 444
Programming Bus-Master DMA 444
Programming Slave DMA 446
Sample EISA Driver Code 447
Initialization Sketch 447
Complete EISA Character Driver 449

PART VIII GIO Drivers

18. GIO Device Drivers 511
GIO Bus Overview 512
GIO Bus Varieties 512
Card Form Factors and Compatibility 512
GIO Bus Address Spaces 513
Configuring a GIO Device 514
GIO VECTOR Line 514

Writing a GIO Driver 515
GIO-Specific Kernel Functions 515
Function setgiovector() 515
Function setgioconfig() 516
splgio0, splgio1, splgio2 517
GIO Driver edtinit() Entry Point 517
GIO Driver Interrupt Handler 519
Using PIO 519
Using DMA 521
DMA To Multiple Pages 521
DMA With Scatter/Gather Capability 521
DMA Without Scatter/Gather Support 523
Memory Parity Workarounds 526
Example GIO Driver 528

PART IX STREAMS Drivers

19. STREAMS Drivers 541
Driver Exported Names 542
Streamtab Structure 542
Driver Flag Constant 542
Initialization Entry Points 543
Entry Point open() 543
Entry Point close() 544
Put Functions wput() and rput() 544
Service Functions rsrv() and wsrv() 545
Building and Debugging 546
Special Considerations for Multiprocessing 547

Special Considerations for IRIX 549
  Extension of Poll and Select 549
  Support for Pipes 549
  Service Scheduling 550
  Supplied STREAMS Modules 550
  No #idefs 551
  Different I/O Hardware Model 551
  Different Network Model 551
  Support for CLONE Drivers 552
    Using the CLONE Driver 552
    Recognizing a Clone Request Independently 553
    Responding to a Clone Request 553
Summary of Standard STREAMS Functions 554
STREAMS Modules for X Input Devices 556
  The X Input Subsystem 556
  Shared Memory Input Queue 557
  IDEV Interface 558
  Input Device Naming 558
  Opening Input Devices 559
    Starting Up the Server 559
    Opening from a Client 560
  Device Controls 560
    Where Controls Are Stored 560
    Control Syntax 561

A. Silicon Graphics Driver/Kernel API 563
Driver Exported Names 564
Kernel Data Structures and Declarations 565
Kernel Functions 566

B. Challenge DMA with Multiple IO4 Boards 581
The IO4 Problem 581
  Software Fix 582
  Software Not Affected 582
Fixing the IO4 Problem 583

Glossary 585

Index 597

List of Examples

Example 2-1 Testing the Hardware Inventory in a Shell Script 31

Example 2-2 Function Returning Type Code for CPU Module 31

Example 4-1 Opening and Using a Hypothetical VME Device 65

Example 4-2 User-Level DMA Access to VME 74

Example 5-1 Testing the Generic SCSI Configuration 89

Example 5-2 Code of the testunitread00() Function 102

Example 5-3 Program That Uses dslib Functions 103

Example 6-1 Function to Test and Set External Interrupt Pulse Width 117

Example 7-1 Hypothetical ULI Program 129

Example 8-1 Hypothetical pfxread() entry in a Character/Block Driver 153

Example 8-2 *pfx*poll() Code for Hypothetical Driver 157

Example 8-3 Edited Fragment of flash_map() 161

Example 8-4 Hypothetical Call to pollwakeup() 166

Example 8-5 Entry Point pfxprint() 169

Example 8-6 Uniprocessor Upper-Half Wait Logic 176

Example 8-7 Uniprocessor Interrupt Logic 177

Example 9-1 LIFO Queue Using Basic Locks 206

Example 9-2 Skeleton Code for Use of SV_WAIT 221

Example 11-1 Verifying Presence of symmon 244

Example 11-2 Setting Kernel putbuf Size 250

Example 11-3 Debugging Macros Using cmn_err() 251

Example 11-4 More Elaborate Debugging Macro 251

Example 11-5 Invoking idbg Interactively 262

Example 11-6 Invoking idbg with a Log File 263

Example 11-7 Invoking idbg for a Single Command 263

Example 12-1 Compiling the Example Driver for a 32-bit Kernel 272

Example 12-2 Displaying Simulated Volume Header Using idbg 274

Example 12-3 Install Command to Create Device Special File 275
Example 12-4 Applying prtvtoc to a RAM Drive of 2 MB 276
Example 12-5 Making a Filesystem on a RAM Drive 276
Example 12-6 Mounting a RAM Drive Filesystem 277
Example 14-1 Comparing pio_badaddr() to pio_badaddr_val() 327
Example 14-2 Example VME Character Driver 335
Example 15-1 Storing the Adapter Type Number in pfxedtinit() 359
Example 15-2 Extracting an Adapter Number From a Minor Device Number 360
Example 15-3 Macro to Encapsulate a Call to scsi_alloc() 360
Example 16-1 Input Queueing Using Locking Macros 399
Example 16-2 Interrupt Handling Using Locking Macros 400
Example 16-3 Skeleton ifnet Driver 401
Example 17-1 Sketch of EISA Initialization 447
Example 17-2 Master File /var/sysgen/rap for RAP-10 Driver 449
Example 17-3 Configuration File /var/sysgen/rap.sm for RAP-10 Driver 450
Example 17-4 Installation Script for RAP-10 Driver 450
Example 17-5 Program to Test RAP-10 Driver 450
Example 17-6 Complete EISA Character Driver for RAP-10 452
Example 18-1 GIO Driver edtinit() Entry Point 518
Example 18-2 Hypothetical PIO Routine for GIO 520
Example 18-3 Strategy Code for Hypothetical Scatter/Gather GIO Device 522
Example 18-4 Strategy() Code for GIO Device Without Scatter/Gather 524
Example 18-5 Disabling SysAD Parity Checking During PIO 527
Example 18-6 Complete Driver for Hypothetical GIO Device 528
Example 19-1 Testing Pipe Configuration 549

List of Figures

Figure 1-1 CPU Access to Memory 7
Figure 1-2 CPU Access to Device Registers 10
Figure 1-3 Device Access to Memory 11
Figure 1-4 Device Access Through a Bus Adapter 12
Figure 1-5 The 32-Bit Address Space 15
Figure 1-6 MIPS 32-Bit Virtual Address Format 16
Figure 1-7 Main Parts of the 64-Bit Address Space 19
Figure 1-8 MIPS 64-Bit Virtual Address Format 21
Figure 1-9 Address Decoding for Physical Memory Access 23
Figure 3-1 Overview of Device Open 47
Figure 3-2 Overview of Device Control 48
Figure 3-3 Overview of Programmed Kernel I/O 49
Figure 3-4 Overview of Memory Mapping 50
Figure 3-5 Overview of DMA I/O 52
Figure 5-1 Bit Assignments in SCSI Device Minor Numbers 80
Figure 13-1 Relationship of VME Bus to System Bus 301
Figure 13-2 VMECC, the VMEbus Adapter 316
Figure 13-3 I/O Address to System Address Mapping 320
Figure 13-4 VMECC Contribution to VME Handshake Cycle Time 322
Figure 16-1 Overview of Network Architecture 390
Figure 17-1 High-Level Overview of EISA Bus in Indigo^2 429
Figure 17-2 Encoding of the EISA Manufacturer ID 433
Figure 18-1 The SysAD Bus in Relation to GIO 526

List of Tables

Table 1-1 CPU Modules and System Names 5
Table 1-2 Number of TLB Entries by Processor Type 9
Table 1-3 Cache Algorithm Selection 23
Table 4-1 VME Bus PIO Bandwidth 66
Table 4-2 EISA Bus PIO Bandwidth (32-Bit Slave, 33-MHz GIO Clock) 70
Table 4-3 EISA Bus PIO Bandwidth (16-Bit Slave, 33-MHz GIO Clock) 70
Table 4-4 VME Bus Bandwidth, DMA Engine, D32 Transfer 73
Table 5-1 Fields of the dsreq Structure 82
Table 5-2 Flag Values for ds_flags 83
Table 5-3 Return Codes From SCSI Operations 85
Table 5-4 SCSI Status Codes 87
Table 5-5 SCSI Message Byte Values 87
Table 5-6 Fields of the dsconf Structure 88
Table 5-7 dslib Function Summary 90
Table 5-8 Lookup Tables in dslib 95
Table 6-1 Functions for Outgoing External Signals 114
Table 6-2 Functions for Incoming External Interrupts 115
Table 8-1 Entry Points in Alphabetic Order 138
Table 8-2 Use of Driver Entry Points 139
Table 9-1 Functions to Manipulate Device Numbers 181
Table 9-2 Header Files Often Used in Device Drivers 185
Table 9-3 Functions for Kernel Virtual Memory 188
Table 9-4 Functions for Allocating pollhead Structures 189
Table 9-5 Functions for Allocating buf_t Objects and Buffers 190
Table 9-6 Functions for Suballocation 191
Table 9-7 Functions for General Data Transfer 192
Table 9-8 Functions Moving Data Using uio_t 194

Table 9-9	Functions to Test Physical Addresses 195
Table 9-10	Functions to Manipulate a vhandl_t Object 196
Table 9-11	Functions to Convert Bytes to Sectors or Pages 198
Table 9-12	Functions Related to Physical Memory 199
Table 9-13	Functions to Map Buffer Pages 199
Table 9-14	Functions Related to Cache Coherency 200
Table 9-15	Functions for User Process Management 202
Table 9-16	Functions for Basic Locks 205
Table 9-17	Functions for Mutex Locks 207
Table 9-18	Functions for Sleep Locks 209
Table 9-19	Functions for Reader/Writer Locks 211
Table 9-20	Functions to Set Interrupt Levels 213
Table 9-21	Functions for Timed Delays 214
Table 9-22	Functions for Synchronizing Block I/O 217
Table 9-23	Functions for Synchronization: sleep/wakeup 219
Table 9-24	Functions for Synchronization: Synchronization Variables 220
Table 9-25	Functions for Semaphores 222
Table 10-1	Compiler Variables Tested by System Header Files 229
Table 10-2	Compiler Options for 32-Bit Kernel Modules 230
Table 10-3	Compiler Options for 64-Bit Kernel Modules 231
Table 11-1	Commands for Symbol Conversion and Lookup 256
Table 11-2	Commands to Control Execution 257
Table 11-3	Commands to Manage Virtual Memory 259
Table 11-4	Commands to Display Memory 260
Table 11-5	Utility Commands 261
Table 11-6	Commands to Display Memory and Symbols 265
Table 11-7	Commands to Display Process Information 265
Table 11-8	Commands to Display Locks and Semaphores 267
Table 11-9	Commands to Display I/O Status 267
Table 11-10	Commands to Display buf_t Objects 268
Table 11-11	Commands to Display STREAMS Structures 268
Table 11-12	Commands to Display Network-Related Structures 269
Table 13-1	VME A16 Addressing in Crimson Systems 307

Table 13-2 VME A24 Addressing in Crimson Systems 307
Table 13-3 VME A32 Addressing in Crimson Systems (One Bus) 307
Table 13-4 VME A32 Addressing in Crimson Systems (Two Buses) 308
Table 13-5 Accessible VME Addresses in Each System 310
Table 14-1 Functions to Create and Use PIO Maps 326
Table 14-2 Functions That Operate on DMA Maps 330
Table 14-3 Functions to Manage Interrupt Vector Values 332
Table 15-1 Host Adapter Driver Classes 355
Table 15-2 Host Adapter Function Summary 357
Table 15-3 Input Fields of the scsi_request Structure 363
Table 15-4 Values for the sr_flags Field of a scsi_request 364
Table 15-5 Values Returned From a SCSI Command 366
Table 15-6 Software Status Values From a SCSI Request 366
Table 15-7 SCSI Status Bytes 367
Table 15-8 Host Adapter Status After a SCSI Request 368
Table 15-9 Adapter Error Codes 378
Table 15-10 Primary Sense Key Error Table 379
Table 15-11 Additional Sense Code Table 380
Table 15-12 SCSI State Error Messages 384
Table 16-1 Important Reference Pages Related to Network Drivers 395
Table 16-2 Mutual Exclusion Macros for ifnet Drivers 398
Table 17-1 Functions to Create and Use PIO Maps 439
Table 17-2 Functions for IRQ and Channel Allocation 442
Table 17-3 Functions That Operate on DMA Maps 445
Table 17-4 Functions for EISA DMA 446
Table 18-1 GIO Slot Names and Addresses 513
Table 19-1 Multiprocessing STREAMS Functions 548
Table 19-2 Kernel Entry Points 554
Table A-1 Driver Exported Names 564
Table A-2 Device Driver Interface Objects 565
Table A-3 STREAMS Driver Interface Objects 566
Table A-4 Kernel Functions 566

About This Guide

This guide describes the ways in which hardware devices are integrated into and controlled from a Silicon Graphics® computer system running the IRIX™ operating system version 6.2 and above.

Three general classes of device-control software exist in an IRIX system: process-level drivers, kernel-level drivers, and STREAMS drivers.

- A process-level driver executes as part of a user-initiated process. Examples include the use of programmed I/O (PIO) to the VME bus, and control of external interrupts in a Challenge™ system.
- A kernel-level driver is loaded as part of the IRIX kernel and executes in the kernel address space, controlling one device in response to calls to its read, write, and ioctl (control) entry points.
- A STREAMS driver is dynamically loaded into the kernel address space to monitor or modify a stream of data passing between a device and a user process.

All three classes are discussed in this guide, although the greatest amount of attention is given to kernel-level drivers.

Note: This edition applies only to IRIX 6.2 and later. If you are working with an earlier release (4.x, 5.2, 5.3, 6.0.x, or 6.1), you should use the version of this manual appropriate to that release.

Audience

In order to write a process-level driver you must be an experienced C programmer with a thorough understanding of the use of IRIX system services and, of course, detailed knowledge of the device to be managed.

In order to write a kernel-level driver or a STREAMS driver you must be an experienced C programmer who knows UNIX® system administration, and expecially IRIX system administration, and who understands the concepts of UNIX device management.

What This Guide Contains

This guide is divided into the following major parts.

Part I, "IRIX Device Integration"	How devices are attached to Silicon Graphics computers, configured to IRIX, and initialized at boot time.
Part II, "Device Control From Process Space"	Details of user-level interrupt handling, programmed I/O of VME and EISA devices, SCSI control using dslib, and external interrupts.
Part III, "Kernel-Level Drivers"	How kernel-level drivers are designed, compiled, loaded, and tested. Survey of kernel services for drivers.
Part IV, "VME Device Drivers"	Kernel-level drivers for the VME bus.
Part V, "SCSI Device Drivers"	Kernel-level drivers for the SCSI bus.
Part VI, "Network Drivers"	Kernel-level drivers for network interfaces.
Part VII, "EISA Drivers"	Kernel-level drivers for the EISA bus.
Part VIII, "GIO Drivers"	Kernel-level drivers for the GIO bus.
Part IX, "STREAMS Drivers"	Design of STREAMS drivers.

In the printed book, you can locate these parts using the part-tabs printed in the margins. Using IRIX Insight™, each part is a top-level division in the clickable table of contents, or you can jump to any part by clicking the blue cross-references in the list above.

Other Sources of Information

Developer Program

Information and support are available through the Silicon Graphics Developer Program. The Developer Toolbox CDROM contains numerous code examples. To join the program, contact the Developer Response Center at (800) 770-3033 or send e-mail to devprogram@sgi.com.

Internet Resources

A great deal of useful material can be found on the internet. Some starting points are in the following list.

SGI patches, examples, and other material.	ftp://ftp.sgi.com
Network of pages of information about Silicon Graphics and MIPS® products	http://www.sgi.com
Text of all Internet RFC documents.	ftp://ds.internic.net/rfc/
Computer graphics pointers at the UCSC Perceptual Science Labororatory.	http://mambo.ucsc.edu/psl/cg.html
Pointers to binaries and sources at The National Research Council of Canada's Institute For Biodiagnostics.	http://zeno.ibd.nrc.ca:80/~sgi/
A Silicon Graphics "meta page" at the Georgia Institute of Technology College of Computing.	http://www.cc.gatech.edu/service/sgimeta.html
Dazzling Silicon Graphics "meta page" at NASA in Huntsville, AL.	http://chernobog.msfc.nasa.gov/SGI/html/SGI.html
Complete SCSI-2 standard in HTML.	http://abekas.com:8080/SCSI2/
IEEE Catalog and worldwide ordering information.	http://stdsbbs.ieee.org:70/0/pub/htmlfiles/stctoc.htm
MIPS processor manuals in HTML form.	http://www.mips.com/

Standards Documents

The following documents are the official standard descriptions of buses:

- *EISA Technical Reference*, available from BCPR Services, Inc., 1400 L Street NW, Washington, DC 20005.
- *Intel 82350D Reference*, Intel™ order number 290377 (EISA reference implementation chip set).
- *ANSI/IEEE standard 1014-1987* (VME Bus), available from IEEE Customer Service, 445 Hoes Lane, PO Box 1331, Piscataway, NJ 08855-1331 (but see also "Internet Resources" on page xxxvii).

Important Reference Pages

The following reference pages contain important details about software tools and practices that you need.

getinvent(3)	The interface to the inventory database
hinv(1)	The use of the inventory display command
intro(7)	The conventions used for special device filenames
MAKEDEV(1)	The use of the program that creates device special files
master(4)	Syntax of files in */var/sysgen/master.d*
prom(1)	Commands of the "miniroot" and other features of the boot PROM, which you use to bring up the system when testing a new device driver
system(4)	Syntax of files in */var/sysgen/system/*.sm*
udmalib(3)	Functions for performing user-level DMA from VME.
uli(3)	Functions for registering and using a user-level interrupt handler.
usrvme(7)	Naming conventions for mappable VME device special files.

Additional Reading

The following books, obtainable from Silicon Graphics, can be helpful when designing or testing a device driver.

- *MIPS Compiling and Performance Tuning Guide*, document number 007-2479-001, tells how to use the C compiler and related tools.
- *MIPSpro Assembly Language Programmer's Guide*, document number 007-2418-001, tells how to compile assembly-language modules.
- *MIPSpro 64-Bit Porting and Transition Guide*, document number 007-2391-001, documents the implications of the 64-bit execution mode for user programs.
- *Topics in IRIX Programming*, document number 008-2478-002, documents some of the sophisticated services offered by the IRIX kernel to user-level programs.
- *MIPS R4000 User's Manual* (2nd ed.) by Joe Heinrich, document number 007-2489-001, gives detailed information on the MIPS instruction set and hardware registers for the processor used in many Silicon Graphics computer systems (also available as HTML on http://www.mips.com/).
- *MIPS R10000 User's Manual* by Joe Heinrich gives detailed information on the MIPS instruction set and hardware registers for the processor used in certain high-end systems. Available only in HTML form from http://www.mips.com/.
- *IRIX Administration: System Configuration and Operation*, document number 007-2859-001, describes the basic adminstrative tools for configuring, operating, and tuning IRIX.
- *IRIX Administration: Disks and File Systems*, document number 007-2825-001, describes the configuration of new disk subsystems and the management of logical volumes and file systems.
- *IRIX Administration: Peripheral Devices*, document number 007-2861-001, describes the adminstration of tapes, printers, and other devices.

The following books, obtainable from bookstores or libraries, can also be helpful.

- Egan, Janet I., and Thomas J. Teixeira. *Writing a UNIX Device Driver*. John Wiley & Sons, 1992.
- Leffler, Samuel J., et alia. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Palo Alto, California: Addison-Wesley Publishing Company, 1989.
- A. Silberschatz, J. Peterson, and P. Galvin. *Operating System Concepts*, Third Edition. Addison Wesley Publishing Company, 1991.
- Heath, Steve. *VMEbus User's Handbook*. CRC Press, Inc, 1989. ISBN 0-8493-7130-9.
- *Device Driver Reference, UNIX SVR4.2*, UNIX Press 1992.
- *UNIX System V Release 4 Programmer's Guide*, UNIX SVR4.2. UNIX Press, 1992.
- *STREAMS Modules and Drivers, UNIX SVR4.2*, UNIX Press 1992. ISBN 0-13-066879.

Conventions Used in This Guide

Special terms and special kinds of words are indicated with the following typographical conventions:

Data structures, variables, function arguments, and macros.	The *dsiovec* structure has members *iov_base* and *iov_len*. Use the *IOVLEN* macro to access them.
Kernel and library functions and functions in examples.	When successful, **v_mapphys**() returns 0.
Driver entry point names that must be completed with a unique prefix string.	The **munmap**() system function calls the *pfx***unmap**() entry point.
Files and directories.	Device special files are in */dev*, and are created using the */dev/MAKEDEV* script.
First use of terms defined in the glossary (see "Glossary" on page 585).	The *inode* of a *device special file* contains the *major device number.*
Literal quotes of code examples.	The SCSI driver's prefix is `scsi_`.

PART ONE

IRIX Device Integration

Chapter 1: Physical and Virtual Memory
An overview of physical memory, virtual address space management, and device addressing in Silicon Graphics/MIPS systems.

Chapter 2: Device Configuration
How IRIX locates devices, and how devices are represented in software.

Chapter 3: Device Control Software
A survey of the ways in which you can control devices under IRIX, from user-level processes and from kernel-level drivers of different kinds.

Chapter 1

Physical and Virtual Memory

This chapter gives an overview of the management of physical and virtual memory in the MIPS® R4*x*00®, R8000™, and R10000™ processors. Access to physical devices is included in this topic, because device registers and bus attachments are accessed using physical memory addresses.

When you are designing a kernel-level driver, this information helps you understand the operation of the kernel functions that you call on, and the constraints on their operations. This information is only of academic interest if you intend to control a device from a user-level process.(See Chapter 3, "Device Control Software," for the difference between these two types of drivers.)

The following main topics are covered in this chapter.

- "Physical Address Space" on page 4 describes the range and meaning of address numbers on the hardware bus.
- "Addresses of Memory and Devices" on page 5 summarizes the hardware architecture by which the CPU accesses memory.
- "The 32-Bit Address Space" on page 14 describes the divisions of the 32-bit virtual address space and their uses.
- "The 64-Bit Address Space" on page 18 describes the divisions of the 64-bit virtual address space and their uses.
- "Device Driver Use of Memory" on page 24 describes the techniques and rules for how kernel-level device drivers allocate and use memory.

Note: This chapter tells only enough about memory access and cache management to explain the rules of the driver/kernel interface. For complete details on the MIPS hardware processors, see the hardware manuals listed under "Additional Reading" on page xxxix.

Physical Address Space

Physical addresses are used to select RAM, ROM, device registers, and bus attachments. Physical addresses start at 0 and can (in some systems) go as high as 2^{40}. This range includes 1.1e12 unique numbers, or 1,024 *gigabytes* (GB), or 1 *terabyte* (TB).

Software never uses physical addresses directly. Kernel-level software can access physical memory and devices using indirect addressing discussed later.

Device Addresses

The MIPS processor architecture has no I/O instructions. Certain ranges of physical addresses are reserved as device addresses. That is, an attempt to access a reserved address is decoded by the hardware as an access to a particular device or bus attachment.

Each Silicon Graphics computer model has a particular set of device addresses. The choice of device addresses is part of the architecture of the whole computer system; it is not designed into the processor chip.

For example, the EISA bus attachment in the Indigo^{2}™ responds to physical memory addresses beginning at 0x000A 0000. (The EISA bus is discussed in detail in Part VII, “EISA Drivers.”) To see some actual device addresses, examine VECTOR statements in files in */var/sysgen/system*.

In some cases, the relationship between device addresses and physical addresses can be changed dynamically. For example, in the Silicon Graphics Challenge and Onyx™ systems, different 8 MB portions of the VME bus address space can be mapped dynamically to different blocks of physical addresses. (VME bus details are in Part IV, “VME Device Drivers.”)

Memory Addresses

Some physical addresses are decoded to select memory hardware. Physical memory includes ROM as well as RAM. Each block of physical memory has a range of physical addresses. The physical addresses where RAM or ROM can be found depend on the particular computer system.

Physical memory does not necessarily occupy sequential addresses. There can be (and often are) gaps, ranges of physical addresses that do not relate to either memory or devices, between ROM addresses and RAM addresses. In most systems, all RAM is given a single sequential span of physical addresses. However, this is not a requirement. Blocks of RAM addresses can also be separated by gaps that are not populated with memory. Since all software uses virtual addresses, software always sees a sequential range of addresses without gaps.

Addresses of Memory and Devices

Each Silicon Graphics computer system has one or more CPU modules. The CPU reads memory or a device by placing an address on a system bus, and receiving data back from the addressed memory or device. Access to memory can pass through multiple levels of cache.

CPU Modules

A CPU is a hardware module containing a MIPS processor chip such as the R8000, together with system interface chips and possibly a secondary cache. Silicon Graphics CPU modules have model designation of the form IP*nn*; for example, the IP22 module is used in the Indy™ workstation. The CPU modules supported by IRIX 6.2 are listed in Table 1-1.

Table 1-1 CPU Modules and System Names

Module	MIPS Processor	System Families
IP17	R4000	Crimson™
IP19	R4*x*00	Challenge (other than S model), Onyx
IP20	R4*x*00	Indigo®

Table 1-1 (continued) CPU Modules and System Names

Module	MIPS Processor	System Families
IP21	R8000	POWER Challenge™, POWER Onyx™
IP22	R4*x*00	Indigo^2, Indy, Challenge S
IP25	R10000	POWER Challenge R10000
IP26	R8000	POWER Indigo^2™

Modules with the same IP designation can be ordered in a variety of clock speeds, and they can differ in other ways. Also, the choice of graphics hardware is independent of the CPU model. However, all these CPUs are identical as seen from software.

Interrogating the CPU Type

At the interactive command line, you can determine which CPU module a system uses with the command

```
hinv -c processor
```

Within a shell script, it is more convenient to process the terse output of

```
uname -m
```

(See the uname(1) and hinv(1) reference pages.)

Within a program, you can get the CPU model using the **getinvent()** function. For an example, see "Testing the Inventory In Software" on page 30.

CPU Access to Memory

The CPU generates the address of data that it needs—the address of an instruction to fetch, or the address of an operand of an instruction. It requests the data through a mechanism that is depicted in simplified form in Figure 1-1.

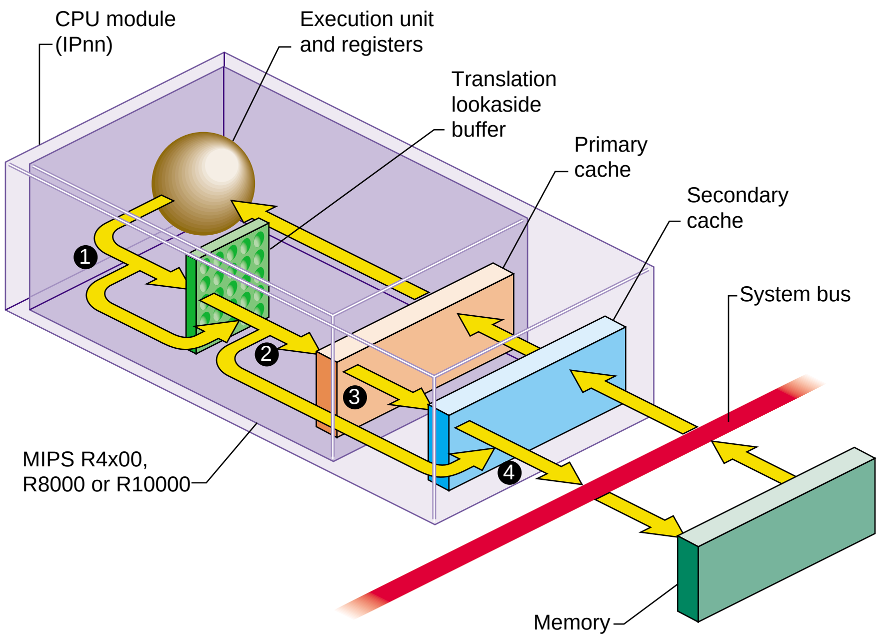
CPU module (IPnn)
Execution unit and registers
Translation lookaside buffer
Primary cache
Secondary cache
System bus
1
2
3
4
MIPS R4x00, R8000 or R10000
Memory

Figure 1-1 CPU Access to Memory

1. The address of the needed data is formed in the processor execution or instruction-fetch unit. Most addresses are then mapped from virtual to real through the Translation Lookaside Buffer (TLB). Certain ranges of addresses are not mapped, and bypass the TLB.
2. Most addresses are presented to the *primary cache*, the cache in the processor chip. If a copy of the data with that address is found, it is returned immediately. Certain address ranges are never cached; these addresses pass directly to the bus.
3. When the primary cache does not contain the data, the address is presented to the secondary cache. If it contains a copy of the data, the data is returned immediately. The size and the architecture of the secondary cache differ from one CPU model to another, and some CPUs do not have a secondary cache.
4. The address is placed on the system bus. The memory module that recognizes the address places the data on the bus.

Processor Operating Modes

The MIPS processor under IRIX operates in one of two modes: kernel and user. The processor enters the more privileged kernel mode when an interrupt, a system instruction, or an exception occurs. It returns to user mode only with a “Return from Exception” instruction.

Certain instructions cannot be executed in user mode. Certain segments of memory can be accessed only in kernel mode, and other segments only in user mode.

Virtual Address Mapping

The MIPS processor contains an array of Translation Lookaside Buffer (TLB) entries that map, or translate, virtual addresses to physical ones. Most memory accesses are first mapped by reference to the TLB. This permits the IRIX kernel to implement *virtual memory* for user processes, and permits it to relocate parts of the kernel itself. The translation scheme is summarized in the following sections and covered in detail in the hardware manuals listed under “Additional Reading” on page xxxix.

TLB Misses and TLB Sizes

Each TLB entry describes a segment of memory containing two adjacent *pages*. When the input address falls in a page described by a TLB entry, the TLB supplies the physical memory address for that page. The translated address, now physical instead of virtual, is passed on to the cache, as shown in Figure 1-1 on page 7.

When the input address is not covered by any active TLB entry, the MIPS processor takes a “TLB miss” interrupt to an IRIX kernel routine. The kernel routine inspects the address. When the address has a valid translation to some page in the address space, the kernel loads a TLB entry to describe that page, and restarts the instruction.

The size of the TLB is important for performance. The size of the TLB in different processors is shown in Table 1-2.

Table 1-2 Number of TLB Entries by Processor Type

Processor Type	Number of TBL Entries
R3000 (not supported by IRIX 6.2)	64
R4x00	96
R8000	384
R10000	128

Address Space Creation

There are not sufficient TLB entries to describe all the address space of every process. The IRIX kernel creates a page table for each process, containing one entry for each virtual memory page in the address space of that process. Whenever an executing program refers to an address for which there is no current TLB entry, the processor traps to the handler for the TLB miss exception. The exception handler loads one TLB entry from the appropriate page table entry of the current process, in order to describe the needed virtual address. Then it resumes execution with the failed instruction.

The kernel maintains a page table in kernel memory for each process, and a page table for the kernel virtual address space as well. In order to extend a virtual address space, the kernel takes the following two steps.

- It allocates unused page table entries to describe the needed pages. This defines the virtual addresses the pages will have.
- It allocates page frames in memory to contain the pages themselves, and puts their physical addresses in the page table entries.

Address Exceptions

When the CPU requests an invalid address—because the processor is in the wrong mode, or an address does not translate to a valid location in the address space, or an address refers to hardware that does not exist in the system—an addressing exception occurs. The processor traps to a particular address in the kernel.

An addressing exception can also be detected while handling a TLB miss—if there is no page table entry assigned for the desired address, that address is not part of the address space of the processs.

When a user-mode process caused the addressing exception, the kernel sends the process a SIGSEGV (see the signal(5) reference page), usually causing a segmentation fault. When kernel-level code such as a device driver caused the addressing exception, the kernel executes a “panic,” taking a crash dump and shutting down the system.

CPU Access to Device Registers

The CPU accesses a device register using the mechanism illustrated in Figure 1-2. Access to device registers is always uncached. It is not affected by considerations of cache coherency in any system (see “Cache Use and Cache Coherency” on page 13).

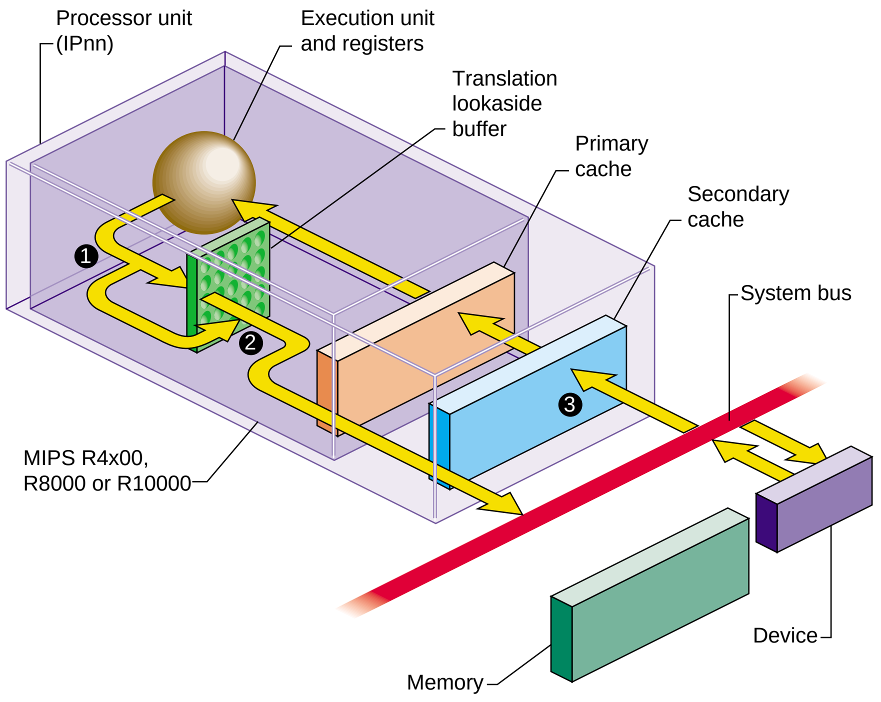
Processor unit (IPnn)
Execution unit and registers
Translation lookaside buffer
Primary cache
Secondary cache
System bus
1
2
3
MIPS R4x00, R8000 or R10000
Device
Memory

Figure 1-2 CPU Access to Device Registers

1. The address of the device is formed in the Execution unit. It may or may not be an address that is mapped by the TLB.
2. A device address, after mapping if necessary, always falls in one of the ranges that is not cached, so it passes directly to the system bus.
3. The device or bus attachment recognizes its physical address and responds with data.

Direct Memory Access

Some devices can perform *direct memory access (DMA)*. In order to read or write a sequence of memory addresses, the device has to be told of the proper physical address range to use. This is done by storing target address numbers into device registers from the CPU. When the device's DMA address registers are loaded, it can access memory through the system bus, as shown in Figure 1-3.

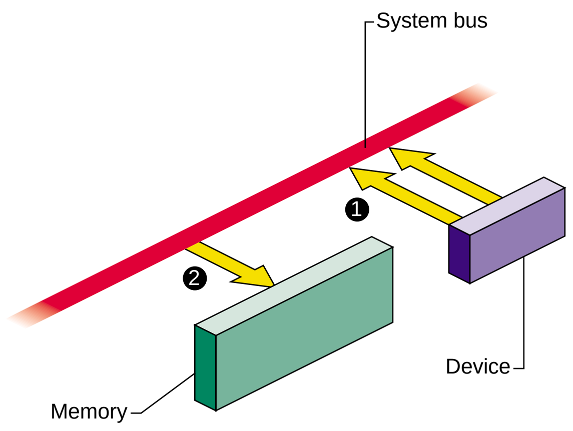

Figure 1-3 Device Access to Memory

1. The device places the next physical address, and data, on the system bus.
2. The memory module stores the data.

When a device is programmed with an invalid physical address, the result is a bus error interrupt.

Bus Virtual Addresses

Figure 1-3 is too simple for some devices that are attached through a bus adapter. A bus adapter connects a bus of a different type to the system bus, as shown in Figure 1-4.

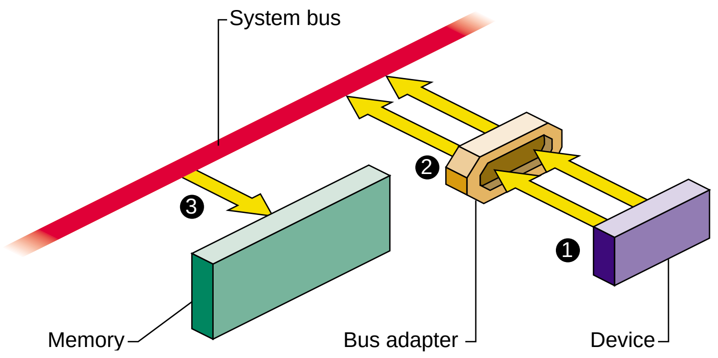

Figure 1-4 Device Access Through a Bus Adapter

For example, the VME adapter connects a VME bus to the system bus. Multiple VME devices can be plugged into the VME bus, and can use the VME bus to read and write. The VME bus adapter translates the VME bus protocol into the system bus protocol. (For details on the VME bus adapter, see Chapter 13, "VME Device Attachment.")

Another example of a bus adapter is the EISA bus adapter that connects EISA devices to a GIO bus in an Indigo^2 workstation. (For details, see Chapter 17, "EISA Device Drivers.")

One task of a bus adapter is to translate between the physical addresses used on the system bus, and the addressing scheme used within the proprietary bus. Translation is most visible with VME bus devices. The VME bus protocol defines several different address space conventions: A16, 16-bit addresses; A32, 32-bit addresses; and so on. A device on the VME bus uses addresses of this form. But VME addresses have no direct relationship to the physical addresses used on the system bus.

The bus adapter translates the addresses used on the proprietary bus to corresponding addresses on the system bus. Considering Figure 1-4, the operation of a DMA device is as follows:

1. The device places a bus virtual address and data on the proprietary (EISA or VME) bus.
2. The bus adapter translates the address to a meaningful physical address, and places that address and the data on the system bus.
3. The memory modules stores the data.

The translation of bus virtual to physical addresses is fixed in some systems. For example, the device in EISA slot 1 on an Indigo^2 is always translated to physical addresses 0x0001 0000 through 0x0001 FFFF.

In most systems, however, the bus translation is programmed by the IRIX kernel to suit the devices that are configured into the system. For example, the VME bus adapter in a Challenge or Onyx system can be programmed to place 15 different "windows" of VME address space at different locations in physical address space. There is no fixed relation between a given VME bus address and a physical address.

Cache Use and Cache Coherency

The primary and secondary caches shown in Figure 1-1 on page 7 are essential to CPU performance. There is an order of magnitude difference in the speed of access between cache memory and main memory. Execution speed remains high only as long as a very high proportion of memory accesses are satisfied from the primary or secondary cache.

The use of caches means that there are often multiple copies of data: a copy in main memory, a copy in the secondary cache (when one is used) and a copy in the primary cache. Moreover, a multiprocessor system has multiple CPU modules like the one shown, and there can be copies of the same data in the cache of *each* CPU.

The problem of *cache coherency* is to ensure that all cache copies of data are true reflections of the data in main memory. Different Silicon Graphics systems use different hardware designs to achieve cache coherency.

In most cases, cache coherence is achieved by the hardware, without any effect on software. In a few cases, specialized software, such as a kernel-level device driver, must take specific steps to maintain cache coherency.

Cache Coherency in Multiprocessors

Multiprocessor systems have more complex cache coherency protection because it is possible to have data in multiple caches. In a multiprocessor system, the hardware ensures that cache coherency is maintained under all conditions, including DMA input and output, without action by the software. However, in some systems the cache coherency hardware works correctly only when a DMA buffer is aligned on a cache-line-sized boundary. You ensure this by using the KM_CACHEALIGN flag when allocating buffer space with **kmem_alloc()** (see the kmem_alloc(D3) reference page).

Note: In one specific hardware configuration of Challenge and Onyx systems, a hardware problem can cause cache coherency errors that a device driver must deal with; see Appendix B, "Challenge DMA with Multiple IO4 Boards."

Cache Coherency in Uniprocessors

In some uniprocessor systems, it is possible for the CPU cache to have newer information than appears in memory. This is a problem only when a device is going to perform DMA. In these systems, a device driver calls a kernel function to ensure that all cached data has been written to memory prior to DMA output (see the dki_cache_wb(D3) reference page). The device driver calls a kernel function to ensure that the CPU receives the latest data following a DMA input (see the dki_cache_inval(D3) reference page). In a multiprocessor these functions do nothing, but it is always safe to call them.

The 32-Bit Address Space

The MIPS processors can operate in one of two address modes: 32-bit and 64-bit. The choice of address mode is independent of other features of the instruction set architecture such as the number of available registers and the precision of integer arithmetic (for example, programs compiled to the n32 binary interface use 32-bit addresses but 64-bit integers). The implications for user programs are documented in manuals listed under "Additional Reading" on page xxxix.

The addressing mode can be switched dynamically; for example, the IRIX kernel can operate with 64-bit addresses, but the kernel can switch to 32-bit address when it dispatches a user program that was compiled for that mode. The 32-bit address space is the range of all addresses that can be used when in 32-bit mode. This space is discussed first because it is simpler and more familiar than the 64-bit space.

Segments of the 32-bit Address Space

When operating in 32-bit mode, the MIPS architecture uses addresses that are 32-bit unsigned integers from 0x0000 0000 to 0xFFFF FFFF. However, this address space is not uniform. The MIPS hardware divides it into segments, and treats each segment differently. The ranges are shown graphically in Figure 1-5.

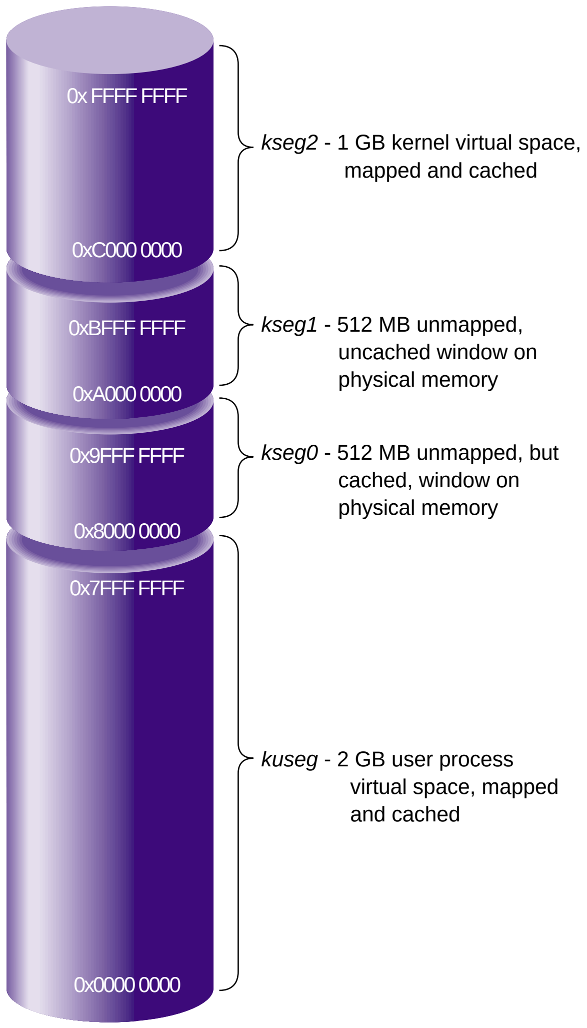
0x FFFF FFFF
0xC000 0000
kseg2 - 1 GB kernel virtual space, mapped and cached
0xBFFF FFFF
0xA000 0000
kseg1 - 512 MB unmapped, uncached window on physical memory
0x9FFF FFFF
0x8000 0000
kseg0 - 512 MB unmapped, but cached, window on physical memory
0x7FFF FFFF
0x0000 0000
kuseg - 2 GB user process virtual space, mapped and cached

Figure 1-5 The 32-Bit Address Space

The address segments differ in three characteristics:

- whether access to an address is mapped; that is, passed through the translation lookaside buffer (TLB)
- whether an address can be accessed when the CPU is operating in user mode or in kernel mode
- whether access to an address is cached; that is, looked up in the primary and secondary caches before it is sent to main memory

Virtual Address Mapping

In the mapped segments, each 32-bit address value is treated as shown in Figure 1-6.

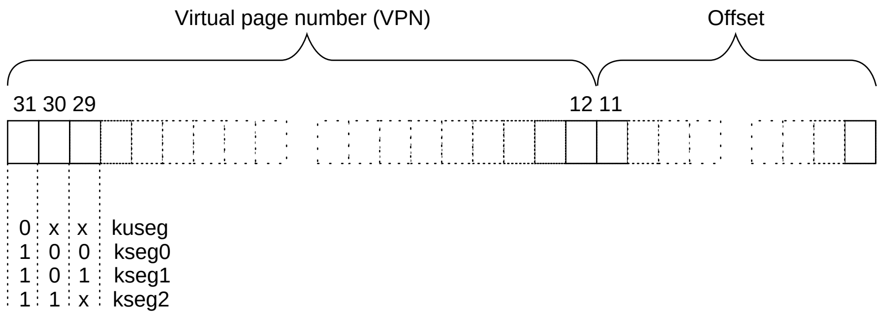

Figure 1-6 MIPS 32-Bit Virtual Address Format

The three most significant bits of the address choose the segment among those drawn in Figure 1-5. When bit 31 is 0, bits 30:12 select a *virtual page number* (VPN) from 2^{19} possible pages in the address space of the current user process. When bits 31:30 are 11, bits 29:12 select a VPN from 2^{18} possible pages in the kernel virtual address space.

User Process Space—kuseg

The total 32-bit address space is divided in half. Addresses with a most significant bit of 0 constitute the 2 GB user process space. When executing in user mode, only addresses in *kuseg* are valid; an attempt to use an address with bit 31=1 causes an addressing exception.

Access to *kuseg* is always mapped through the TLB. The kernel creates a unique address space for each user process. Of the 2^{19} possible pages in an address space, most are typically unassigned—few processes ever occupy more than a fraction of *kuseg*—and many are shared pages of program text from dynamic shared objects (DSOs) that are mapped into the address space of every process that needs them.

Kernel Virtual Space—kseg2

When bits 31:30 are 11, access is to kernel virtual memory. Only code that is part of the kernel can access this space. References to this space are translated through the TLB. The kernel uses the TLB to map kernel pages in memory as required, possibly in noncontiguous locations. Although pages in kernel space are mapped, they are always associated with real memory. Kernel memory is never paged to secondary storage.

This is the space in which the IRIX kernel allocates such objects as stacks, user page tables, and per-process data that must be accessible on context switches. This area contains automatic variables declared by loadable device drivers. It is the space in which kernel-level device drivers allocate memory. Since kernel space is mapped, addresses in *kseg2* that are apparently contiguous need not be contiguous in physical memory. However, a device driver can can allocate space that is both logically and physically contiguous, when that is required (see for example the kmem_alloc(D3) reference page).

Cached Physical Memory—kseg0

When address bits 31:29 contain 100, access is directed to physical memory through the cache. If the addressed location is not in the cache, bits 28:0 are placed on the system bus as a physical memory address, and the data presented by memory or a device is returned.

Since only 29 bits are available for mapping physical memory, only 512 MB of physical memory space can be accessed through this segment in 32-bit mode (some of which must be reserved for device addressing). It is possible to gain cached access to wider physical addresses by mapping through the TLB into *kseg2*, but systems that need access to more physical memory typically run in 64-bit mode (see “Cache-Controlled Physical Memory—xkphys” on page 22).

Kseg0 contains the exception address to which the MIPS processor branches it when it detects an exception such as an addressing exception or TLB miss.

Uncached Physical Memory—kseg1

When address bits 31:29 contain 101, access is directly to physical memory, bypassing the cache. Bits 28:0 are placed on the system bus for memory or device transfer.

The kernel refers to *kseg1* when interacting with I/O devices because loads or stores from device registers should not pass through cache memory. The kernel also uses *kseg1* when operating on certain data structures that might be *volatile*. Kernel-level device drivers sometimes need to write to uncached memory, and must take special precautions when doing so (see “Uncached Memory Access in the IP26 CPU” on page 27).

Portions of *kseg0* or *kseg1* can be mapped into *kuseg* by the **mmap()** function. This is covered at more length under “Memory Use in User-Level Drivers” on page 25.

The 64-Bit Address Space

The 64-bit mode is an upward extension of 32-bit mode. All MIPS processors from the R4000 on support 64-bit mode. However, this mode was not used in Silicon Graphics software until IRIX 6.0 was released.

Segments of the 64-Bit Address Space

When operating in 64-bit mode, the MIPS architecture uses addresses that are 64-bit unsigned integers from 0x0000 0000 0000 0000 to 0xFFFF FFFF FFFF FFFF. This is an immense span of numbers—if it were drawn to a scale of 1 millimeter per terabyte, the drawing would be 16.8 kilometers long (just over 10 miles).

The MIPS hardware divides the address space into segments based on the most significant bits, and treats each segment differently. The ranges are shown graphically in Figure 1-7. These major segments define only a fraction of the 64-bit space. Most of the possible addresses are undefined and cause an addressing exception (segmentation fault) if used.

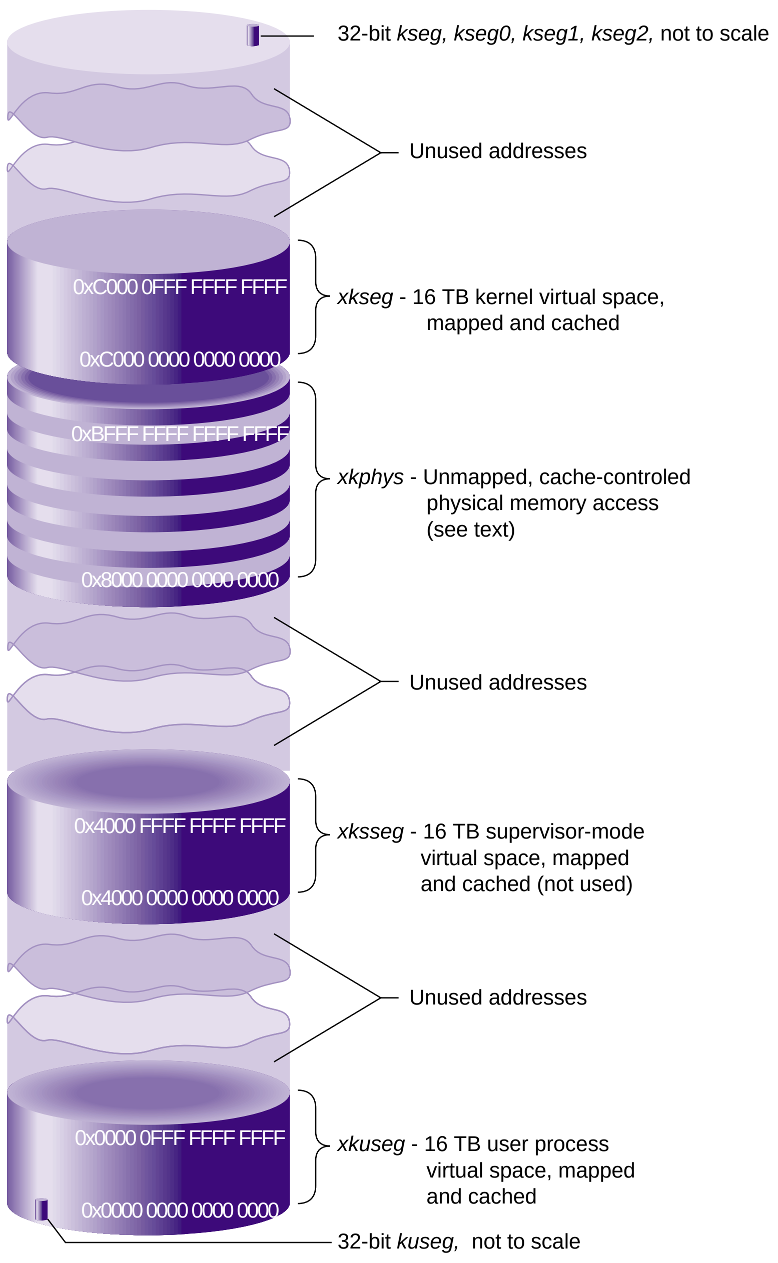
32-bit *kseg, kseg0, kseg1, kseg2,* not to scale
Unused addresses
0xC000 0FFF FFFF FFFF
0xC000 0000 0000 0000
xkseg - 16 TB kernel virtual space, mapped and cached
0xBFFF FFFF FFFF FFFF
0x8000 0000 0000 0000
xkphys - Unmapped, cache-controled physical memory access (see text)
Unused addresses
0x4000 FFFF FFFF FFFF
0x4000 0000 0000 0000
xksseg - 16 TB supervisor-mode virtual space, mapped and cached (not used)
Unused addresses
0x0000 0FFF FFFF FFFF
0x0000 0000 0000 0000
xkuseg - 16 TB user process virtual space, mapped and cached
32-bit *kuseg,* not to scale

Figure 1-7 Main Parts of the 64-Bit Address Space

As in the 32-bit space, these major segments differ in three characteristics:

- whether access to an address is mapped; that is, passed through the translation lookaside buffer (TLB)
- whether an address can be accessed when the CPU is operating in user mode or in kernel mode.
- whether access to an address is cached; that is, looked up in the primary and secondary caches before it is sent to main memory

Compatibility of 32-Bit and 64-Bit Spaces

The MIPS-3 instruction set (which is in use when the processor is in 64-bit mode) is designed so that when a 32-bit instruction is used to generate or to load an address, the 32-bit operand is automatically sign-extended to fill the high-order 32 bits.

As a result, any 32-bit address that falls in the user segment *kuseg*, and which must have a sign bit of 0, is extended to a 64-bit integer with 32 high-order 0 bits. This automatically places the 32-bit *kuseg* in the bottom of the 64-bit *xkuseg*, as shown in Figure 1-7.

A 32-bit kernel address, which must have a sign bit of 1, is automatically extended to a 64-bit integer with 32 high-order 1 bits. This places all kernel segments shown in Figure 1-5 at the extreme top of the 64-bit address space. However, these 32-bit kernel spaces are not used by a kernel operating in 64-bit mode.

Virtual Address Mapping

In the mapped segments, each 64-bit address value is treated as shown in Figure 1-8.

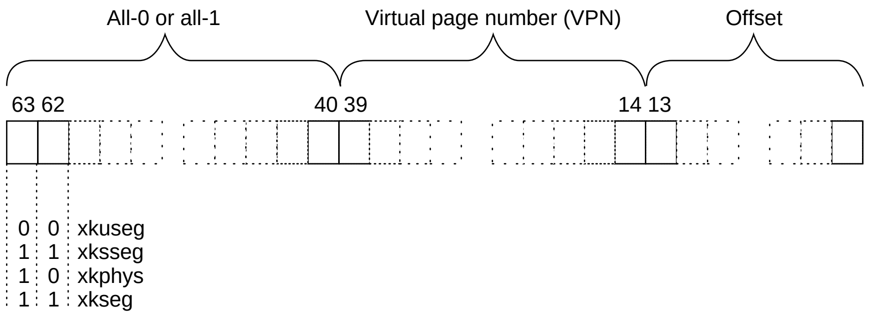

Figure 1-8 MIPS 64-Bit Virtual Address Format

The two most significant bits select the major segment (compare these to the address boundaries in Figure 1-7). Bits 61:40 must all be 0. (In principle, references to 32-bit kernel segments would have bits 61:40 all 1, but these segments are not used in 64-bit mode.)

The size of a page of virtual memory is a compile-time parameter when the kernel is created. In IRIX 6.2, the page size in a 32-bit kernel is 4 KB and in a 64-bit kernel is 16 KB. (Either size could change in later releases, so always determine it dynamically. In a user-level program, call the **getpagesize()** function (see the getpagesize(2) reference page). In a kernel-level driver, use the **ptob()** kernel function (see the ptob(D3) reference page) or the constant NBPP declared in *sys/immu.h*.)

When the page size is 16 KB, bits 13:0 of the address represent the offset within the page, and bits 39:14 select a VPN from the 2^{26}, or 64 M, pages in the virtual segment..

User Process Space—xkuseg

The first 16 TB of the address space are devoted to user process space. Access to *xkuseg* is always mapped through the TLB. The kernel creates a unique address space for each user process. Of the 2^{26} possible pages in a process's address space, most are typically unassigned, and many are shared pages of program text from dynamic shared objects (DSOs) that are mapped into the address space of every process that needs them.

Supervisor Mode Space—xksseg

The MIPS architecture permits three modes of operation: user, kernel, and supervisor. When operating in kernel or supervisor mode, the 2 TB space beginning at 0x4000 0000 0000 0000 is accessible. IRIX does not employ the supervisor mode, and does not use *xksseg*. If *xksseg* were used, it would be mapped and cached.

Kernel Virtual Space—xkseg

When bits 63:62 are 11, access is to kernel virtual memory. Only code that is part of the kernel can access this space, a 2 TB segment starting at 0xC000 0000 0000 0000. References to this space are translated through the TLB, and cached. The kernel uses the TLB to map kernel pages in memory as required, possibly in noncontiguous locations. Although pages in kernel space are mapped, they are always associated with real memory. Kernel pages are never paged to secondary storage.

This is the space in which the IRIX kernel allocates such objects as stacks, per-process data that must be accessible on context switches, and user page tables. This area contains automatic variables declared by loadable device drivers. It is the space in which kernel-level device drivers allocate memory. Since kernel space is mapped, addresses in *kseg2* that are apparently contiguous need not be contiguous in physical memory. However, a device driver can can allocate space that is both logically and physically contiguous, when that is required (see for example the kmem_alloc(D3) reference page).

Cache-Controlled Physical Memory—xkphys

One-quarter of the 64-bit address space—all addresses with bits 63:62 containing 10—are devoted to special access to the 1 TB physical address space. In 64-bit mode this space replaces the *kseg0* and *kseg1* spaces used in 32-bit mode. Addresses in this space are interpreted as shown in Figure 1-9.

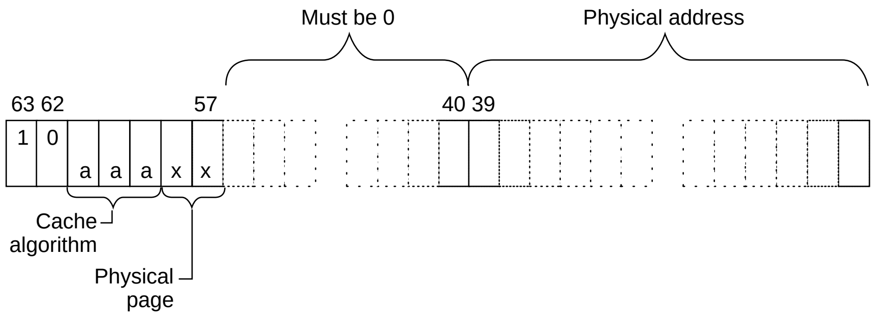

Figure 1-9 Address Decoding for Physical Memory Access

Bits 39:0 select a physical address in a 1 TB range. As a result, a system operating in 64-bit mode can access a much larger physical address space than the 512 MB space allowed by *kseg0*. This permits more physical memory to be installed, and it gives more freedom in assigning device and bus addresses.

Bits 57:40 must always contain 0. Bits 61:59 select the hardware cache algorithm to be used. The only values defined for these bits are summarized in Table 1-3.

Table 1-3 Cache Algorithm Selection

Address 61:59	Algorithm	Meaning
010	Uncached	This is the 64-bit equivalent of *kseg1* in 32-bit mode—uncached access to physical memory.
110	Cacheable coherent exclusive on write	This is the 64-bit equivalent of *kseg0* in 32-bit mode—cached access to physical memory, coherent access in a multiprocessor.
011	Cacheable non-coherent	Data is cached; on a cache miss the processor issues a non-coherent read (one without regard to other CPUs).
100	Cacheable coherent exclusive	Data is cached; on a read miss the processor issues a coherent read exclusive.

Table 1-3 (continued) Cache Algorithm Selection

Address 61:59	Algorithm	Meaning
101	Cacheable coherent update on write	Same as 110, but updates memory on a store hit in cache.
111	Uncached Accelerated	Same as 010, but the cache hardware is permitted to defer writes to memory until it has collected a larger block, improving write utilization.

Only the 010 (uncached) and 110 (cached) algorithms are implemented on all systems. The others may or may not be implemented on particular systems.

Bits 58:59 must be 00 unless the cache algorithm is 010 (uncached) or 111(uncached accelerated). Then bits 58:59 can in principle be used to select four other properties to qualify the operation. No present Silicon Graphics computer system supports these properties, so bits 58:59 always contain 00 at this time.

Portions of *xkphys* and *xkseg* can be mapped to user process space by the **mmap()** function. This is covered in more detail under “Memory Use in User-Level Drivers” on page 25.

Device Driver Use of Memory

Memory use by device drivers is simpler than the details in this chapter suggest. The primary complication for the designer is the use of 64-bit addresses, which may be unfamiliar.

Allowing for 64-Bit Mode

You must take account of a number of considerations when porting an existing C program to an environment where 64-bit mode is used, or might be used. For details see the *MIPSpro 64-Bit Porting and Transition Guide* listed on page xxxix.

The most common problems arise because the size of a pointer and of a long int changes between a program compiled with the -64 option and one compiled -32. When pointers, longs, or types derived from longs, are used in structures, the field offsets differ between the two modes.

When all programs in the system are compiled to the same mode, there is no problem. This is the case for a system in which the kernel is compiled to 32-bit mode: only 32-bit user programs are supported. However, a kernel compiled to 64-bit mode executes user programs in 32-bit or 64-bit mode. A structure prepared by a 32-bit program—a structure passed as an argument to **ioctl()**, for example—does not have fields at the offsets expected by a 64-bit kernel device driver. For more on this specific problem, see "Handling 32-Bit and 64-Bit Execution Models" on page 170.

The header files *sgidefs.h* and *sys/types.h* define type names that you can use to declare structures that always have the same size.

Memory Use in User-Level Drivers

When you control a device from a user process, your code executes entirely in user process space, and has no direct access to any of the other spaces described in this chapter.

Depending on the device and other considerations, you may use the **mmap()** function to map device registers into the address space of your process (see the mmap(2) reference page). When the kernel maps a device address into process space, it does it using the TLB mechanism. From **mmap()** you receive a valid address in process space. This address is mapped through a TLB entry to an address in segment that accesses uncached physical memory. When your program refers to this address, the reference is directed to the system bus and the device.

Portions of kernel virtual memory (*kseg0* or *xkseg*) can be accessed from a user process. Access is based on the use of device special files (see the mem(7) reference page). Access is done using two models, a device model and a memory map model.

Access Using a Device Model

The device special file */dev/mem* represents physical memory. A process that can open this device can use **lseek()** and **read()** to copy physical memory into process virtual memory. If the process can open the device for output, it can use **write()** to patch physical memory.

The device special file */dev/kmem* represents kernel virtual memory (*kseg0* or *xkseg*). It can be opened, read and written similarly to */dev/mem*. Clearly both of these devices should have file permissions that restrict their use even for input.

Access Using mmap()

The **mmap()** function allows a user process to map an open file into the process address space (see the mmap(2) reference page). When the file that is mapped is */dev/mem*, the process can map a specified segment of physical memory. The effect of **mmap()** is to set up a page table entry and TLB entry so that access to a range of virtual addresses in user space is redirected to the mapped physical addresses in cached physical memory (*kseg0* or the equivalent segment of *xkphys*).

The */dev/kmem* device, representing kernel virtual memory, cannot be used with **mmap()**. However, a third device special, */dev/mmem* (note the double "m"), represents access to only those addresses that are configured in the file */var/sysgen/master.d/mem*. As distributed, this file is configured to allow access to the free-running timer device and, in some systems, to graphics hardware.

For an example of mapped access to physical memory, see the example code in the syssgi(2) reference page related to the SGI_QUERY_CYCLECNTR option. In this operation, the address of the timer (a device register) is mapped into the process's address space using a TLB entry. When the user process accesses the mapped address, the TLB entry converts it to an address in *kseg1*/*xkphys*, which then bypasses the cache.

Mapped Access Provided by a Device Driver

A kernel-level device driver can provide mapped access to device registers or to memory allocated in kernel virtual space. An example of such a driver is shown in Part III, "Kernel-Level Drivers."

Memory Use in Kernel-Level Drivers

When you control a device from a kernel-level driver, your code executes in kernel virtual space. The allocation of memory for program text, local (stack) variables, and static global variables is handled automatically by the kernel. Besides designing data structures so they have a consistent size, you have to consider these special cases:

- dynamic memory allocation for data and for buffers
- getting addresses of device registers for the device you control
- transferring data between kernel space and user process space

The kernel supplies utility functions to help you deal with each of these issues, all of which are discussed in Chapter 9, "Device Driver/Kernel Interface."

Uncached Memory Access in the Challenge and Onyx Series

Access to uncached memory is not supported. The Challenge and Onyx systems have coherent caches; cache coherency is maintained by the hardware, even under access from CPUs and concurrent DMA. There is never a need (and no approved way) to access uncached memory in these systems.

Note: In certain specific hardware configurations, a device driver may have to deal with cache issues in a Challenge or Onyx system. See Appendix B, "Challenge DMA with Multiple IO4 Boards."

Uncached Memory Access in the IP26 CPU

The IP26 CPU module is used in the Silicon Graphics Power Indigo^2 workstation and the Power Challenge M workstation. Both are deskside workstations using the R8000 processor chip.

Late in the design of these systems, the parity-based memory that had been planned for them was replaced with ECC memory (error-correcting code memory, which can correct for single-bit errors on the fly). ECC memory is also used in large multiprocessor systems from Silicon Graphics, where it has no effect on performance.

Owing to the hardware design of the IP26, ECC memory could be added with no impact on the performance of cached memory access, but uncached memory access can be permitted only when the CPU is placed in a special, "slow" access mode.

In some cases a kernel-level device driver must be sure that stored data has been written into main memory, rather than being held in the cache. There are two ways to ensure this:

- Store the data into cached memory, then use the **dki_dcache_wb()** function to force a range of cached addresses to be written to memory. This method works in all systems including the IP26; however, the function call is an expensive one when the amount of data is small.
- Write directly to uncached memory using addresses in *kseg1*. This works in all systems, but in the IP26 (only) it will fail unless the CPU is first put into "slow" mode.

In order to put the CPU into “slow” mode, call the function **ip26_enable_ucmem()**. As soon as the uncached store is complete, return the system to “fast” mode by calling **ip26_return_ucmem()**. (See the ip26_ucmem(D3) reference page.) While the CPU is in “slow” mode, several clock cycles are added to every memory access, so do not keep it in “slow” mode any longer than necessary.

These functions can be called in any system. They do nothing unless the CPU is an IP26. Alternatively, you could save the current CPU type using a function like the one shown in Example 2-2 on page 31, and call the functions only when that function returns INV_IP26BOARD.

Chapter 2

Device Configuration

This chapter discusses how IRIX establishes the inventory of available hardware, and how devices are represented to software.

This information is essential when your work involves attaching a new device or a new class of devices to IRIX. The information is helpful background material when you intend to control a device from a user-level process.

The following primary topics are covered in this chapter.

- "Hardware Inventory" on page 29 describes the hardware inventory table displayed by the *hinv* command and how the inventory is initialized.
- "Device Special Files" on page 32 describes the system of filenames in */dev* and how they are created.
- "Configuration Files" on page 38 summarizes the files used for system generation and kernel configuration.

Hardware Inventory

During IRIX bootstrap, each device driver probes the hardware attachments for which it is responsible, and adds information to a hardware inventory table. The hardware inventory table is kept in kernel virtual memory. It is available to users and to programs.

Using the Hardware Inventory

The hardware inventory is used by users, administrators, and programmers.

Contents of the Inventory

Using database terminology, the hardware inventory consists of a single table with the following columns:

Class	A code for the class of device; for example, audio, disk, processor, or network.
Type	A code for the type of device within its class; for example, FPU and CPU types within the processor class.
Controller	When applicable, the number of the controller, board, or attachment.
Unit	When applicable, the logical unit or device within a Controller number.
State	A descriptive number, such as the CPU model number.

Displaying the Inventory with hinv

The *hinv* command formats all or selected rows of the inventory table for display (see the hinv(1) reference page), translating the numbers to readable form. The user or system administrator can use command options to select a class of entries or certain specific device types by name. The class or type can be qualified with a unit number and a controller number. For example,

```
hinv -c disk -b 1 -u 4
```

displays information about disk 4 on controller 1.

You can use *hinv* to check the result of installing new hardware. The new hardware should show up in the report after the system is booted following installation, provided that the associated device driver was called and was written correctly.

A full inventory report (*hinv -v*) is almost mandatory documentation for a software problem report, either submitted by your user to you, or by you to Silicon Graphics.

Testing the Inventory In Software

Within a shell script, you can test the output of *hinv* most conveniently in the command exit status. The command sets exit status of 0 when it finds or reports any items. It sets status of 1 when it finds no items. The code in Example 2-1 could be used in a shell script to test the existence of a disk controller.

Example 2-1 Testing the Hardware Inventory in a Shell Script

```
if hinv -s -c disk -b 1;
   then ;
   else echo No second disk controller;
fi ;
```

You can access the inventory table in a C program using the functions documented in the getinvent(3) reference page. The only access method supported is a sequential scan over the table, viewing all entries. Three functions permit access:

- **setinvent()** initializes or reinitializes the scan to the first row.
- **getinvent()** returns the next table row in sequence.
- **endinvent()** releases storage allocated by **setinvent()**.

These functions use static variables and should only be used by a single process within an address space. Reentrant forms of the same functions, which can safely be used in a multithreaded process, are also available (see getinvent(3)). Example 2-2 demonstrates the use of these functions.

The format of one inventory table row is declared as type *inventory_t* in the *sys/invent.h* header file. This header file also supplies symbolic names for all the class and type numbers that can appear in the table, as well as containing commentary explaining the meanings of some of the numbers.

Example 2-2 Function Returning Type Code for CPU Module

```
#include <stddef.h> /* for NULL */
#include <invent.h> /* includes sys/invent.h */
int getIPtypeCode()
{
   inv_state_t * pstate = NULL;
   inventory_t * work;
   int ret = 0;
   setinvent_r(&pstate);
   do {
      work = getinvent_r(pstate);
      if ( (INV_PROCESSOR == work->inv_class)
      &&   (INV_CPUBOARD == work->inv_type) )
         ret = work->inv_state;
   } while (!ret)
   endinvent_r(pstate); /* releases pstate-> */
   return ret;
}
```

Creating an Inventory Entry

Device drivers supplied by Silicon Graphics add information to the hardware inventory table when they are called at their *pfx***init()** or *pfx***edtinit()** entry points. One of these entry points is called by the IRIX kernel during bootstrap. (The small distinction between the two entry points is discussed in “Initialization Entry Points” on page 143.)

The function that adds a row to the inventory table is **add_to_inventory()**. Its prototype is declared in the include file *sys/invent.h*. The function takes arguments that are scalar values corresponding to the fields of the *inventory_t* structure.

Note: In IRIX 6.2, the only valid inventory types and classes are those declared in *sys/invent.h*. Only those numbers can be decoded and displayed by the *hinv* command, which prints an error message if it finds an unknown device class, and which prints nothing at all for an unknown device type within a known class. There is no provision for adding new device-class or device-type values for third-party devices.

Warning: The driver interface to the hardware inventory will change in a release following IRIX 6.2. The add_to_inventory() function of IRIX 6.2 is not formally part of the device driver API, and a different function or functions will be used in a future release.

Device Special Files

Devices are represented within IRIX as objects in the file system, specifically device special file nodes in the */dev* directory. These special file nodes are, in some cases, created automatically during the bootstrap process, and in some cases created manually by the system administrator.

Device Representation

The IRIX record of a file’s existence is sometimes called an *inode*. The device special files consist of inodes only, with no associated data. The fields of the inode are used to encode the following critical information about a device:

Filename	Programs use the name of a device file to open the device using **open()**.
Permissions, Owner ID, Group ID	The file access permissions, owner ID, and group ID of a device file establish which users can read and which can write to that device.
Block or Character	A device file belongs to one of two classes, block or character, visible as the first letter of an *ls -l* display.
Major device number	A code for the device driver that controls this device.
Minor device number	A code specifying the unit or position of this device under its controller.

All this information is visible in a display produced by *ls -l*. The major and minor numbers are shown in the column used for file size for regular files. Examine the output of a command such as

```
ls -l /dev/* | more
```

A device special file can be used the same as a regular file in most IRIX commands; for example, a device file can be the target of a symbolic link, the destination of redirected input or output, and so on.

Block Versus Character

IRIX supports two classes of device. A *block device* such as a disk drive transfers data in fixed size blocks between the device and memory, and usually has some ability to reposition the medium so as to read or write the same data again. The driver for a block device typically has to manage buffering, and may schedule I/O operations in a different sequence than they are requested.

A *character device* such as a printer accepts or returns data as a stream of bytes, and usually acts as a sink or source of data—the medium cannot be repositioned and read again. The driver for a character device typically transfers data as soon as it is requested and completes one operation before accepting another request. Character devices are also called *raw* devices, because their input is not buffered.

Major Device Number

The *major device number* recorded in the device special inode selects the device driver to service this device. When a device is opened, IRIX selects the driver to handle the device based on the major device number. Each device driver supports one or more specific major numbers. There are two unrelated ranges of major numbers, one for character device drivers and one for block device drivers.

The possible major numbers are declared and given names in the file *sys/major.h*. When you create a new kernel-level device driver you must choose a major number for it—a number not used by any other driver. Numbers 60-79 are not used by Silicon Graphics. (See "Selecting a Major Number" on page 226.)

In IRIX releases through 5.2 (and 6.0.x, which is based on 5.2), major numbers were limited to the range 0 through 254. Beginning with releases 5.3 and 6.1, the IRIX inode structure permits major numbers to have up to 14 bits of precision. However, major numbers are currently restricted to at most 9 bits to conserve kernel data space.

In order to use this limit symbolically, use the name L_MAXMAJ defined in *sys/sysmacros.h*. When you declare a variable for a major device number in a program, use type *major_t* declared in *sys/types.h*.

Normally a device driver services only one major number. However, it is possible to designate the same device driver to service more than one major number. In this case, the driver may need to discover the major number at execution time. The **getemajor()** function returns the number in use for a given request (see the getemajor(D3) reference page).

Minor Device Number

The *minor device number* is passed to the device driver as an argument when the driver is called. (The major and minor numbers are passed together in a long integer called a *dev_t*.) The minor device number is interpreted only by the device driver, so it can be a simple logical unit number, or it can contain multiple, encoded bit fields. For example:

- The IRIX tape device driver uses the minor device number to encode the options for rewind or no-rewind, byte-swap or nonswap, and fixed or variable blocking, along with the logical unit number.
- The IRIX disk device drivers encode the disk partion number into the minor device number along with a disk unit number. Both disk and tape devices encode the SCSI adapter number in the minor number.
- The IRIX generic SCSI driver encodes the adapter (bus) number, target (control unit) number, and logical unit number into the minor number (see “Generic SCSI Device Special Files” on page 78).

The IRIX inode structure permits minor numbers to have up to 18 bits of precision. In order to use this limit symbolically, use the name L_MAXMIN defined in *sys/sysmacros.h*. When you declare a variable for a minor device number in a program, use type *minor_t* declared in *sys/types.h*.

With STREAMS drivers, the minor device number can be chosen arbitrarily during a CLONE open—see “Support for CLONE Drivers” on page 552.

Defining Device Names

The device special files related to Silicon Graphics device drivers are created by execution of the script */dev/MAKEDEV*. Additional device special files can be created with administrator commands.

IRIX Conventional Device Names

The device drivers distributed with IRIX depend on certain conventions for device names. These conventions are spelled out in the following reference pages: intro(7), dks(7), dsreq(7), and tps(7). For example, the components of a disk device name in */dev/dsk* include

`dks`*c*	Constant prefix "dks" followed by bus adapter number *c*.
`d`*u*	Constant letter "d" followed by disk SCSI ID number *u*.
`l`*n*	Optionally, letter "l" (ell) and logical unit number *n* (used only when disk u controls multiple drives).
`s`*p* or **`vh`** or **`vol`**	Constant letter "s" and partition number *p*, or else "vh" for volume header, or "vol" for (entire) volume.

Programs throughout the system rely on the conventions for these device names. In addition, by convention the associated major and minor numbers agree with the names. For example, the logical unit and partition numbers that appear in a disk name are also encoded into the minor number.

The Script MAKEDEV

The conventions for all the IRIX device special names are written into the script */dev/MAKEDEV*. This is a make file, but unlike most make files, it is not used to compile executable programs. It contains the logic to prepare device special names and their associated major and minor numbers and file permissions.

The MAKEDEV script is executed during IRIX startup from a script in */etc/rc2.d*. It is executed after all device drivers have been initialized, so it can use the output of the *hinv* command to construct device names to suit the actual configuration.

The system administrator can invoke MAKEDEV to construct device special files. Administrator use of MAKEDEV is described in *IRIX Administration: System Configuration and Operation*.

Making Device Files

You or a system administrator can create device special files explicitly using the commands *mknod* or *install*. Either command can be used in a make file such as you might create as part of the installation script for a product.

For details of these commands, see the install(1) and mknod(1M) reference pages, and *IRIX Administration: System Configuration and Operation*. The following is a hypothetical example of *install*:

```
# install -m 644 -u root -g sys -root /dev -chr 62,0
```

The *-chr* option specifies a character device, and *62,0* are the major and minor device numbers, respectively.

Tip: The *mknod* command is portable, being used in most UNIX systems. The *install* command is unique to IRIX, and has a number of features and uses beyond those of *mknod*. Examples of both can be found by reading */dev/MAKEDEV*.

Multiple Names for One Device

It is possible to point to the same device with more than one device special filename. This is done in the distributed IRIX system for several reasons:

- To supply default names for devices with specific names. For example, the default device */dev/tapens* is a link to the first device file in */dev/rmt/**.
- To pass different parameters to the device driver. For example, the same tape device appears multiple times in */dev/rmt/tps**, with different combinations of *nr* (norewind), *ns* (nonswapped), and *v* (variable block) suffixes. The minor number for each name encodes these options for the same unit number.
- To supply both block and character drivers for the same device. For example, each disk device appears in */dev/dsk/** as a block device, and again in */dev/rdsk/** as a character device.

Configuration Files

IRIX uses a number of configuration files to supplement its knowledge of devices and device drivers. This is a summary of the files. The use of each file for device driver purposes is described in more detail in other chapters. (The uses of these files for other system administration tasks is covered in *IRIX Administration: System Configuration and Operation*.)

Most configuration files used by the IRIX kernel are located in the directory */var/sysgen*. Files used by the X11 display system are generally in */usr/lib/X11*. With regard to device drivers, the important files are:

*/var/sysgen/master.d.**	Descriptions of the attributes of kernel modules
*/var/sysgen/boot/**	Kernel object modules
/var/sysgen/system/.sm*	Device configuration information
*/var/sysgen/mtune/**	Values and limits of tunable parameters
/var/sysgen/stune	New values for tunable parameters
*/usr/lib/X11/input/config/**	Initialization commands for Xdm input modules

Master Configuration Database

Every configurable module of the kernel (this includes kernel-level device drivers and some other service modules) is represented by a single file in the directory */var/sysgen/master.d*.

A file in *master.d* describes the attributes of a module of the kernel which is to be loaded at boot time. The general syntax of the file is documented in detail in the master(4) reference page. Only a subset of the syntax is used to describe a device driver module. In general, the *master.d* file specifies device driver attributes such as:

- the driver's *prefix*, a name that qualifies all its entry points
- whether it is a block, character, or STREAMS driver
- the major number serviced by the driver
- whether the driver can be loaded dynamically as needed
- whether the driver is multiprocessor-aware
- which of the possible driver entry points the driver supplies

For each module described in a *master.d* file there should be a corresponding object module in */var/sysgen/boot*. The creation of device driver modules and the syntax of *master.d* files is covered in detail in Chapter 10, "Building and Installing a Driver."

System Configuration Files

The files */var/sysgen/system/*.sm* direct the *lboot* command in loading the modules of the kernel at boot time. Although there are normally several files with the names of subsystems, all the files are treated as one single file. The contents of the files direct *lboot* in loading components that are described by files in */var/sysgen/master.d*, and in probing for devices to see if they exist.

The exact syntax of these files is documented in the system(4) reference page. The use of the VECTOR lines to probe for hardware is covered in this book in the context of each type of attachment. For example, probing for VME devices is covered under "Configuring the System Files" on page 311.

System Tuning Parameters

The IRIX kernel supports a variety of tunable parameters, some of which can be interrogated by device drivers. The current values of the parameters are recorded in files in */var/sysgen/mtune/** (one file per major subsystem).

You or the system administrator can view the current settings using the *systune* command (see the systune(1M) reference page). The system administrator can use *systune* to request changes in parameters. Some changes take effect at once; others are recorded in a modified kernel that is loaded the next time the system boots.

To retrieve certain tuning parameters from within a kernel-level device driver, include the header file *sys/var.h*.

The use of *systune* and its related files is covered in *IRIX Administration: System Configuration and Operation*.

X Display Manager Configuration

Most files related to the configuration of the X Display Manager *Xdm* are held in */var/X11*. These files are documented in reference pages such as xdm(1) and in the programming manuals related to the X Windows System™.

One set of files, in */usr/lib/X11/input/config*, controls the initialization of nonstandard input devices. These devices use STREAMS modules, and their configuration is covered in Chapter 19, "STREAMS Drivers."

Chapter 3

Device Control Software

IRIX provides for two general methods of controlling devices, at the user level and at the kernel level. This chapter describes the architecture of these two software levels and points out the different abilities of each. This is important background material for understanding all types of device control. The chapter covers the following main topics:

- "User-Level Device Control" on this page summarizes five methods of device control for user-initiated processes.
- "Kernel-Level Device Control" on page 45 sets the concepts needed to understand kernel-level drivers.

User-Level Device Control

In IRIX terminology, a *user-level* process is one that is initiated by a user (possibly the superuser). A user-level process runs in an address space of its own, with no access to the address space of other processes or to the kernel's address space, except through explicit memory-sharing agreements.

In particular, a user-level process has no access to physical memory (which includes access to device registers) unless the kernel allows the process to share part of the kernel's address space. (For more on physical memory, see Chapter 1, "Physical and Virtual Memory.")

There are several ways in which a user-level process can control devices, which are summarized in the following topics:

- "EISA Mapping Support" on page 42 summarizes PIO access to the EISA bus.
- "VME Mapping Support" on page 42 summarizes PIO access to the VME bus.
- "User-Level DMA From the VME Bus" on page 43 summarizes DMA I/O managed from a user-level process.
- "User-Level Control of SCSI Devices" on page 43 summarizes DMA and command access to the SCSI bus.

- "Managing External Interrupts" on page 43 summarizes access to the external interrupt ports on Challenge and Onyx systems.
- "User-Level Interrupt Management" on page 44 summarizes the handling of some interrupts in a user-level process.

EISA Mapping Support

In systems that support the EISA bus (Indigo^2 Maximum Impact and Indigo^2, Challenge M, and their Power versions), IRIX contains a kernel-level device driver that supports memory-mapping EISA bus addresses into the address space of a user process (see "Overview of Memory Mapping" on page 50).

You can write a program that maps a portion of the EISA bus address space into the program address space. Then you can load and store from device registers directly.

For more details of PIO to the EISA bus, see Chapter 4, "User-Level Access to VME and EISA."

VME Mapping Support

In systems that support the VME bus (Onyx, Challenge DM, Challenge L, Challenge XL, and their Power versions), IRIX contains a kernel-level device driver that supports mapping of VME bus addresses into the address space of a user process (see "Overview of Memory Mapping" on page 50).

You can write a program that maps a portion of the VME bus address space into the program address space. Then you can load and store from device registers directly.

For more details of PIO to the VME bus, see Chapter 4, "User-Level Access to VME and EISA."

User-Level DMA From the VME Bus

The Challenge L, Challenge XL, and Onyx systems and their Power versions contain a DMA engine that manages DMA transfers from VME devices, including VME slave devices that normally cannot do DMA.

The DMA engine in these systems can be programmed directly from code in a user-level process. Software support for this facility is contained in the *udmalib* package.

For more details of user DMA, see Chapter 4, "User-Level Access to VME and EISA" and the udmalib(3) reference page.

User-Level Control of SCSI Devices

IRIX contains a special kernel-level device driver whose purpose is to give user-level processes the ability to issue commands and read and write data on the SCSI bus. By using **ioctl()** calls to this driver, a user-level process can interrogate and program devices, and can initiate DMA transfers between memory buffers and devices.

The low-level programming used with the dsreq device driver is eased by the use of a library of utility functions documented in the dslib(3) reference page.

For more details on user-level SCSI access, see Chapter 5, "User-Level Access to SCSI Devices."

Managing External Interrupts

The Challenge L, Challenge XL, and Onyx systems and their Power versions have four external-interrupt output jacks and four external-interrupt input jacks on their back panels. In these systems, the device special file */dev/ei* represents a device driver that manages access to these external interrupt ports.

Using **ioctl()** calls to this device (see “Overview of Device Control” on page 48), your program can

- enable and disable the detection of incoming external interrupts
- set the strobe length of outgoing signals
- strobe, or set a fixed level, on any of the four output ports

In addition, library calls are provided that allow very low-latency detection of an incoming signal.

For more information on external interrupt management, see Chapter 6, “Control of External Interrupts” and the ei(7) reference page.

User-Level Interrupt Management

A new facility in IRIX 6.2 allows you to receive and handle device interrupts in a function in a user-level program you write.

Your program calls a library function to register the interrupt-handling function. When the device generates an interrupt, the kernel branches directly into your handler. Because the handler runs as a subroutine of the kernel, it can use only a very limited set of system and library functions. However, it can refer to variables in the process address space, and it can wake up a process that is blocked, waiting for the interrupt to occur.

Combined with PIO, user-level interrupts allow you to test most of the logic of a device driver for a new device in user-level code.

In IRIX 6.2, support for user-level interrupts is limited to VME devices and to external interrupts in the Challenge L, Challenge XL,and Onyx systems and their POWER versions.

For more details on user-level interrupts, see Chapter 7, “User-Level Interrupts” and the uli(3) reference page.

Memory-Mapped Access to Serial Ports

The Audio/Serial Option (ASO) board for the Challenge and Onyx series provides six high-performance serial ports, each of which can be set to run at speeds as high as 115,200 bits per second. The features and administration of the Audio/Serial Option board are described in the *Audio/Serial Option User's Guide* (document 007-2645-001).

The serial ports of the ASO board can be accessed in the usual way, by opening a file to a device in the */dev/tty** group of names. However, for the minimum of latency and overhead, a program can open a device in the */dev/aso_mmap* directory. These device files are managed by a device driver that permits the input and output ring buffers for the port to be mapped directly into the user process address space. The user-level program can spin on the input ring buffer pointers and detect the arrival of a byte of data in microseconds after the device driver stores it.

The details of the memory-mapping driver for ASO ports are spelled out in the asoserns(7) reference page (available only when the ASO feature has been installed).

Kernel-Level Device Control

IRIX supports the conventional UNIX architecture in which a user process uses a kernel service to request a data transfer, and the kernel calls on a device driver to perform the transfer.

Kinds of Kernel-Level Drivers

There are three distinct kinds of kernel-level drivers:

- A *character device driver* transfers data as a stream of bytes of arbitrary length. A character device driver is called as a direct result of a user process issuing a system function call such as **read()** or **ioctl()**.
- A *block device driver* transfers data in blocks of fixed size, and is called from the kernel to support filesystem or paging operations.

- A STREAMS driver is not a device driver, but rather can be dynamically installed to operate on the flow of data to and from any character device driver.

Overviews of the operation of STREAMS drivers are found in Chapter 19, “STREAMS Drivers.” The rest of this discussion is on character and block device drivers.

Typical Driver Operations

There are five different kinds of operations that a device driver can support:

- The open interaction is supported by all drivers; it initializes the connection between a process and a device.
- The control operation is supported by character drivers; it allows the user process to modify the connection to the device or to control the device.
- Programmed I/O (PIO) is used by character drivers to transfer small quantities of data synchronously.
- Memory mapping enables the user process to perform PIO for itself.
- Direct memory access (DMA) is used by block device drivers to transfer larger quantities of data asynchronously under device control.

The following topics present a conceptual overview of the relationship between the user process, the kernel, and the kernel-level device driver. The software architecture that supports these interactions is documented in detail in Part III, “Kernel-Level Drivers,” especially Chapter 8, “Structure of a Kernel-Level Driver.”

Overview of Device Open

Before a user process can use a kernel-controlled device, the process must open the device as a file. A high-level overview of this process, as it applies to a character device driver, is shown in Figure 3-1.

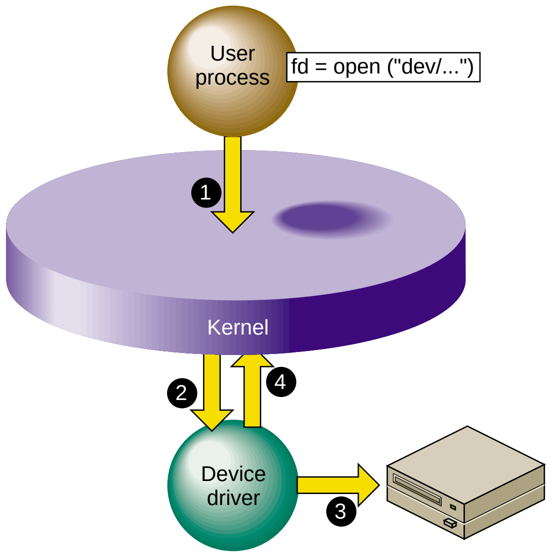

Figure 3-1 Overview of Device Open

The steps illustrated in Figure 3-1 are:

1. The user process calls the **open()** kernel function, passing the name of a device special file (see "Device Special Files" on page 32 and the open(2) reference page).
2. The kernel notes the device major and minor numbers from the inode of the device special file (see "Device Representation" on page 33). The kernel uses the major device number to select the device driver, and calls the driver's open entry point, passing the minor number and other data.
3. The device driver verifies that the device is operable, and prepares whatever is needed to operate it.
4. The device driver returns a return code to the kernel, which returns either an error code or a *file descriptor* to the process.

It is up to the device driver whether the device can be used by only one process at a time, or by more than one process. If the device can support only one user, and is already in use, the driver returns the EBUSY error code.

The **open()** interaction on a block device is similar, except that the operation is initiated from the filesystem code responding to a **mount()** request, rather than coming from a user process **open()** request (see the mount(1) reference page).

There is also a **close()** interaction so a process can terminate its connection to a device.

Overview of Device Control

After the user process has successfully opened a character device, it can request control operations. Figure 3-2 shows an overview of this operation.

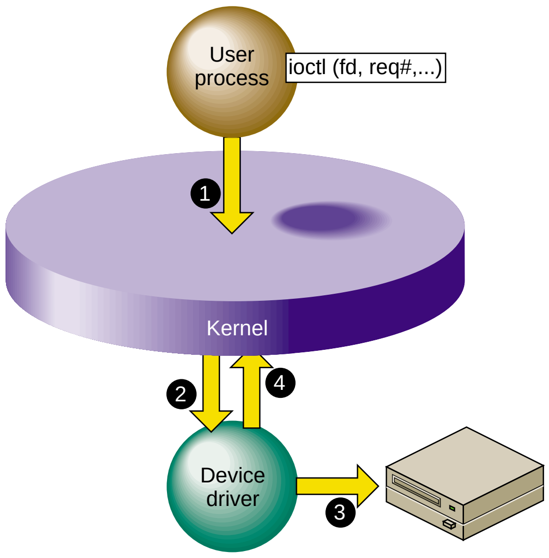

Figure 3-2 Overview of Device Control

The steps illustrated in Figure 3-2 are:

1. The user process calls the **ioctl()** kernel function, passing the file descriptor from open and one or more other parameters (see the ioctl(2) reference page).
2. The kernel uses the major device number to select the device driver, and calls the device driver, passing the minor device number, the request number, and an optional third parameter from **ioctl()**.
3. The device driver interprets the request number and other parameter, notes changes in its own data structures, and possibly issues commands to the device.

4. The device driver returns an exit code to the kernel, and the kernel (then or later) redispatches the user process.

Block device drivers are not asked to provide a control interaction. The user process is not allowed to issue **ioctl()** for a block device.

The interpretation of ioctl request codes and parameters is entirely up to the device driver. For examples of the range of ioctl functions, you might review some reference pages in volume 7, for example, termio(7), ei(7), and arp(7P).

Overview of Programmed Kernel I/O

Figure 3-3 shows a high-level overview of data transfer for a character device driver that uses programmed I/O.

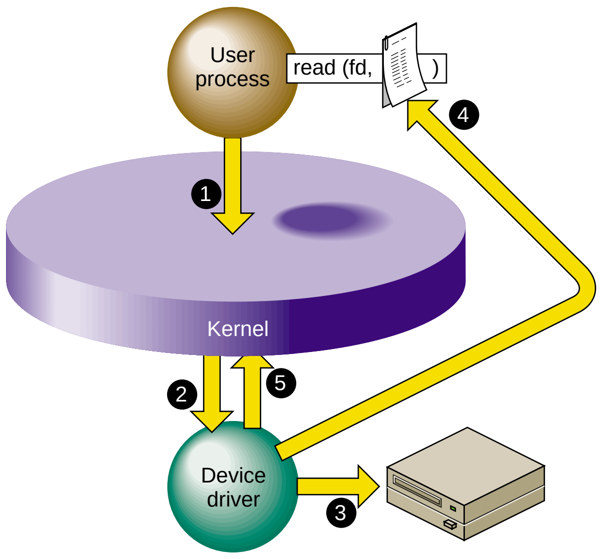

Figure 3-3 Overview of Programmed Kernel I/O

The steps illustrated in Figure 3-3 are:

1. The user process invokes the **read()** kernel function for the file descriptor returned by **open()** (see the read(2) and write(2) reference pages).
2. The kernel uses the major device number to select the device driver, and calls the device driver, passing the minor device number and other information.

3. The device driver directs the device to operate by storing into its registers in physical memory.
4. The device driver retrieves data from the device registers and uses a kernel function to store the data into the buffer in the address space of the user process.
5. The device driver returns to the kernel, which (then or later) dispatches the user process.

The operation of **write()** is similar. A kernel-level driver that uses programmed I/O is conceptually simple since it is basically a subroutine of the kernel.

Overview of Memory Mapping

It is possible to allow the user process to perform I/O directly, by mapping the physical addresses of device registers into the address space of the user process. Figure 3-4 shows a high-level overview of this interaction.

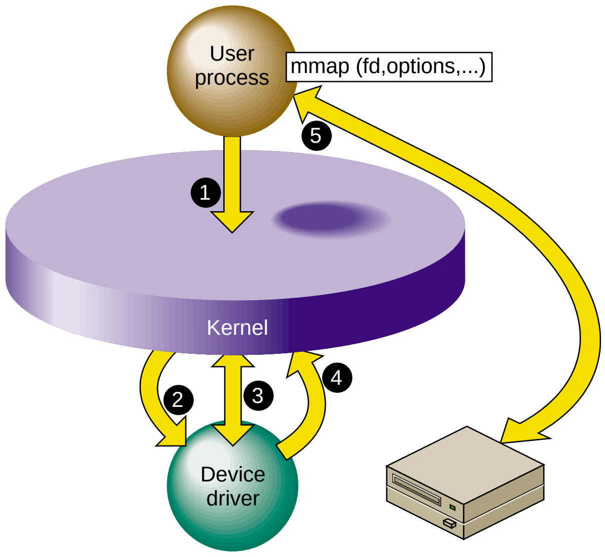

Figure 3-4 Overview of Memory Mapping

The steps illustrated in Figure 3-4 are:

1. The user process calls the **mmap()** kernel function, passing the file descriptor from open and various other parameters (see the mmap(2) reference page).
2. The kernel uses the major device number to select the device driver, and calls the device driver, passing the minor device number and certain other parameters from **mmap()**.
3. The device driver validates the request and uses a kernel function to map the necessary range of physical addresses into the address space of the user process.
4. The device driver returns an exit code to the kernel, and the kernel (then or later) redispatches the user process.
5. The user process accesses data in device registers by accessing the virtual address returned to it from the **mmap()** call.

Memory mapping can be supported by either a character or a block device driver. When a block device driver supports it, the memory mapping request comes from the filesystem when it responds to an **mmap()** call to map a file into memory. The filesystem calls the driver to map different pages of the file into memory, as required.

Memory mapping by a character device driver has the purpose of making device registers directly accessible to the process as memory addresses.

The Silicon Graphics device drivers for the VME and EISA buses support memory mapping. This enables user-level processes to perform PIO to devices on these buses, as described under “EISA Mapping Support” on page 42 and “VME Mapping Support” on page 42.

A memory-mapping character device driver is very simple; it needs to support only **open()**, **mmap()**, and **close()** interactions. Data throughput can be higher when PIO is performed in the user process, since the overhead of the **read()** and **write()** system calls is avoided.

It is possible to write a kernel-level driver that only maps memory, and controls no device at all. Such drivers are called *pseudo-device* drivers. For examples of psuedo-device drivers, see the prf(7) and imon(7) reference pages.

Overview of DMA I/O

Block devices and block device drivers normally use DMA (see "Direct Memory Access" on page 11). With DMA, the driver can avoid the time-consuming process of transferring data between memory and device registers. Figure 3-5 shows a high-level overview of a DMA transfer.

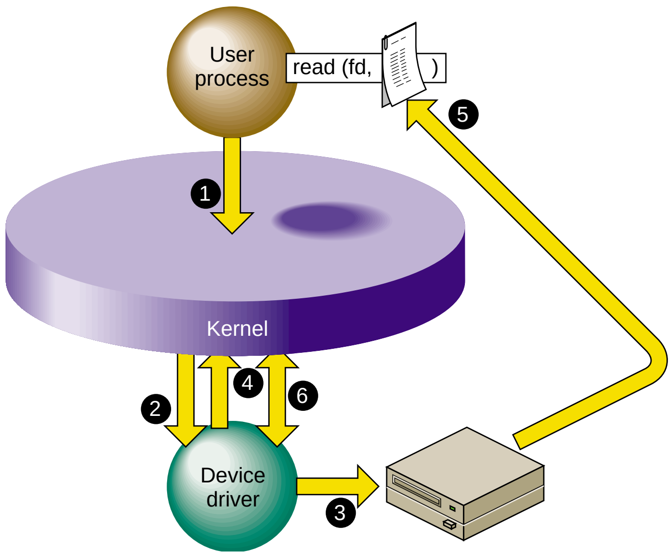

Figure 3-5 Overview of DMA I/O

The steps illustrated in Figure 3-5 are:

1. The user process invokes the **read()** kernel function for a normal file descriptor (not necessarily a device special file). The filesystem (not shown) asks for a block of data.
2. The kernel uses the major device number to select the device driver, and calls the device driver, passing the minor device number and other information.
3. The device driver uses kernel functions to locate the filesystem buffer in physical memory; then programs the device with target addresses by storing into its registers.
4. The device driver returns to the kernel after telling it to block the user process from running.

5. The device itself stores the data to the physical memory locations that represent the buffer in the user process address space. During this time the kernel may dispatch other processes.
6. When the device presents a hardware interrupt, the kernel invokes the device driver. The driver notifies the kernel that the user process can now resume execution. The filesystem code moves the requested data into the user process buffer.

DMA is fundamentally asynchronous. There is no necessary timing relation between the operation of the device performing its operation and the operation of the various user processes. A DMA device driver has a more complex structure because it must deal with such factors as

- programming a device to store into a page buffer in physical memory
- scheduling operations, such as blocking a user process and waking it up when the operation is complete
- asynchronous interrupts from the device
- the possibility that requests from other processes can occur while the device is operating
- the possibility that a device interrupt can occur while the driver is handling a request

When a DMA driver permits multiple processes to open its device, it must be able to cope with the possibility that it can receive several requests from different processes while the device is busy handling one operation. This implies that the driver must implement some method of queuing requests until they can be serviced in turn.

The mapping between physical memory and process address space can be complicated. For example, the buffer can span multiple pages, and the pages need not be in contiguous locations in physical memory. If the device supports *scatter/gather,* it can be programmed with the starting addresses and lengths of each page in the buffer. If it does not support scatter/gather, the device driver has to program a separate DMA operation for each page or part of a page—or else has to obtain a contiguous buffer in the kernel address space, do the I/O from that buffer, and copy the data from that buffer to the process buffer.

The reward for the extra complexity of DMA is the possibility of much higher performance. The device can store or read data from memory at its maximum rated speed, while other processes can execute in parallel.

Upper and Lower Halves

When a device can produce hardware interrupts, its kernel-level device driver has two distinct logical parts, called the “upper half” and the “lower half” (although the upper “half” usually has much more than half the code).

Driver Upper Half

The upper half of a driver comprises all the parts that are invoked as a result of user process calls: the driver entry points that execute in response to **open()**, **close()**, **ioctl()**, **mmap()**, **read()** and **write()**.

These parts of the driver are called on behalf of a specific process. This is referred to as “having user context,” which means that they are executed under the identity of a specific process.

As a result, code in the upper half of the driver is allowed to request kernel services that can be delayed, or “sleep.” For example, code in the upper half of a driver can call **kmem_alloc()** to request memory in kernel space, and can specify that if memory is not available, the driver can sleep until memory is available. Also, code in the upper half can wait on a semaphore until some event occurs, or can seize a lock knowing that it may have to sleep until the lock is released.

In each case, the entire kernel does not “sleep.” The driver upper half sleeps under the identity of the user process; the kernel dispatches other processes to run. When the blocking condition is removed—when memory is available, the semaphore is posted, or the lock is released—the driver is scheduled for execution and resumes.

Driver Lower Half

The lower half of a driver comprises the code that is called to respond to a hardware interrupt. An interrupt can occur at almost any time, including large parts of the time when the kernel is executing other services, including driver upper halves, and even driver lower halves for devices with lower-priority interrupts.

The kernel is not in a known state when executing a driver lower half, and there is no process context. Several things follow from this fact:

- It is very important that the interrupt be handled in the absolute minimum of time, since it may be delaying a kernel service or even the handling of a lower-priority interrupt.
- The lower-half code may not use any kernel service that can sleep (because there is no dispatchable process to be blocked and dispatched again later). Every authorized kernel service is documented as to whether it can sleep or not.

Relationship Between Halves

Each half has its proper kind of work. In general terms, the upper half performs all validation and preparation, including allocating and deallocating memory and copying data between address spaces. It initiates the first device operation of a series and queues other operations. Then it waits on a semaphore.

The lower half verifies the correct completion of an operation. If another operation is queued, it initiates that operation. Then it posts the semaphore to awaken the upper half, and exits.

Layered Drivers

IRIX allows for "layered" device drivers, in which one driver operates the actual hardware and the driver at the higher layer presents the programming interface. This approach is implemented for SCSI devices: actual management of the SCSI bus is delegated to a set of Host Adapter drivers. Drivers for particular kinds of SCSI devices call the Host Adapter driver through an indirect table to execute SCSI commands. SCSI drivers and Host Adapter drivers are discussed in detail in Chapter 15, "SCSI Device Drivers."

Combined Block and Character Drivers

A block device driver is called indirectly, from the filesystem, and it is not allowed to support the **ioctl()** entry point. In some cases, block devices can also be thought of as character devices. For example, a block device might return a string of diagnostic information, or it might be sensitive to dynamic control settings.

It is possible to support *both* block and character access to a device: block access to support filesystem operations, and character access in order to allow a user process (typically one started by a system administrator) to read, write, or control the device directly.

For example, the Silicon Graphics disk device drivers support both block and character access to disk devices. This is why you can find every disk device represented as a block device in the */dev/dsk* directory and again as a character device in */dev/rdsk* (“r” for “raw,” meaning character devices).

Drivers for Multiprocessors

Many Silicon Graphics computers have multiple CPUs that execute concurrently. The CPUs share access to the single main memory, including a single copy of the kernel address space. In principle, all CPUs can execute in the kernel code simultaneously. In principle, the upper half of a device driver could be entered simultaneously by as many different processes are there are CPUs in the system (up to 36 in a Challenge or Onyx system).

A device driver written for a uniprocessor system cannot tolerate concurrent execution by multiple CPUs. For example, a uniprocessor driver has scalar variables whose values would be destroyed if two or more processes updated them concurrently.

In order to make uniprocessor drivers work in multiprocessors, IRIX by default uses only CPU 0 to execute calls to upper-half code of character and STREAMS drivers. This ensures that at most one process executes in any upper half at one time. (Network and block device drivers do not receive this service.)

It is not difficult to design a kernel-level driver to execute safely in any CPU of a multiprocessor. Each critical data object must be protected by a lock or semaphore, and particular techniques must be used to coordinate between the upper and lower halves. These techniques are discussed in “Planning for Multiprocessor Use” on page 171.

When you have made a driver multiprocessor-safe, you compile it with a particular flag value that IRIX recognizes. From then on, the driver upper half is executed on any CPU of a multiprocessor. This can improve performance, since processes that use the driver are not required to wait for CPU 0 to be available.

Loadable Drivers

Some drivers are needed whenever the system is running, but others are needed only occasionally. IRIX allows you to create a kernel-level device driver or STREAMS driver that is not loaded at boot time, but only later when it is needed.

A loadable driver has the same purposes as a nonloadable one, and uses the same interfaces to do its work. A loadable driver can be configured for automatic loading when its device is opened. Alternatively it can be loaded on command using the *ml* program (see the ml(1) and mload(4) reference pages).

A loadable driver remains in memory until its device is no longer in use, or until the administrator uses *ml* to unload it. A loadable driver remains in memory indefinitely, and cannot be unloaded, unless it provides a *pfx***unload()** entry point (see “Entry Point unload()” on page 167).

There are some small differences in the way a loadable driver is compiled and configured (see “Configuring a Loadable Driver” on page 237).

One operational difference is that a loadable driver is not available in the miniroot, the standalone system administration environment used for emergency maintenance. If a driver might be required in the miniroot, it can be made nonloadable, or it can be configured for “autoregistration” (see “Registration” on page 239).

PART TWO

Device Control From Process Space

Chapter 4: User-Level Access to VME and EISA
How a user-level process can access and control devices on the VME and EISA buses.

Chapter 5: User-Level Access to SCSI Devices
How a user-level process can execute commands and transfer data to a SCSI device.

Chapter 6: Control of External Interrupts
How a user-level process creates or responds to external interrupt signals in the Challenge and Power Challenge systems.

Chapter 7: User-Level Interrupts
How a user-level process can trap and respond to device interrupts with low latency and the fewest context switches.

Chapter 4

User-Level Access to VME and EISA

Programmed I/O (PIO) refers to loading and storing data between program variables (actually, CPU registers) and device registers. This is done by setting up a memory mapping of a VME or EISA device into the process address space, so that the program can treat device registers as if they were volatile memory locations. This chapter discusses the methods of setting up this mapping, and the performance that can be obtained. The main topics are as follows:

- "VME Programmed I/O" on page 62 discusses PIO mapping of VME devices.
- "EISA Programmed I/O" on page 67 discusses PIO mapping of EISA devices.
- "VME User-Level DMA" on page 70 discusses the use of the DMA engine in a Challenge or Onyx system.

Normally, PIO programs are designed in synchronous fashion; that is, the process issues commands to the device and then polls the device to find out when the action is complete. However, it is possible for a user process to receive interrupts from a mapped VME device—see Chapter 7, "User-Level Interrupts."

A user-level process can perform DMA transfers from a VME bus master or (in the Challenge or Onyx series) a VME bus slave, directly into the process address space. The use of these features is covered under "VME User-Level DMA" on page 70.

VME Programmed I/O

For an overview of the VME bus and its hardware implementation in Silicon Graphics systems, see Chapter 13, “VME Device Attachment.”

Mapping a VME Device Into Process Address Space

As discussed in “Device Addresses” on page 4, an I/O device is represented as an address, or range of addresses, in the physical address space. A kernel-level device driver has the ability to set up a mapping between the address of a device register and an arbitrary location in the address space of a user-level process. When this has been done, the device register appears to be a variable in memory. The program can assign values to it, or refer to it in expressions.

Learning VME Device Addresses

In order to map a VME device for PIO, you must know the following points:

- the VME bus number on which the device resides

  The Crimson series supports two VME busses, numbered 0 and 1.

  Challenge and Onyx systems support as many as five VME buses. The first is number 0. Use the *hinv* command to display the numbers of others (and see “VME Bus Numbering” on page 315).
- the VME address space in which the device resides

  This will be either A16, A24, or A32—the A64 space is not supported for PIO.
- the VME address space modifier the device uses—either supervisory (s) or nonprivileged (n)
- the VME bus addresses associated with the device

  This must be a sequential range of VME bus addresses that spans all the device registers you need to map.

This information is normally supplied by the manufacturer of a third-party VME device. You can find these values for Silicon Graphics equipment by examining the */var/sysgen/system/irix.sm* file, in which each configured VME device is specified by a VECTOR line. When you examine a VECTOR line, note the following parameter values:

bustype	Specified as *VME* for VME devices. The VECTOR statement can be used for other types of buses as well.
adapter	The number of the VME bus where the device is attached.
iospace, *iospace2*, *iospace3*	Each *iospace* group specifies the VME address space and modifier, the starting bus address, and the size of a segment of VME address space used by this device.

Within each *iospace* parameter group you find keywords and numbers for the address space, modifier, and addresses for a device. The following is an example of a VECTOR line:

```
VECTOR: bustype=VME module=cdsio ipl=5 ctlr=0 adapter=0
iospace=(A24S,0xF00000,0x10000) probe_space=(A24S,0xF0FFFF,1)
```

This example specifies a VME device (*bustype=VME*) on bus 0 (*adapter=0*). The device resides in the A24 address space in supervisory mode (*iospace=(A24S...)*). Its first VME bus address is 0xF0 0000 and it covers a span of 0x01 0000 (64K) addresses—in other words, 0xF0 0000 through 0xF0 FFFF.

For third-party VME devices, look for a VECTOR line supplied by the manufacturer, usually stored in some file in */var/sysgen/system*.

Opening a Device Special File

When you know the device addresses, you can open a device special file that represents the correct range of addresses. The device special files for VME mapping are found in */dev/vme*.

The naming convention for these files is documented in the usrvme(7) reference page. Briefly, each file is named **vme***B***a***SM*, where

B	is one or two digits for the bus number, for example 0 or 53
S	is two digits for the address space, 16, 24, or 32
M	is the modifier, either *s* for supervisory or *n* for nonprivileged

The device special file for the device described by the example VECTOR line in the preceding section would be */dev/vme/vme0a24s*.

In order to map a device on a particular bus and address space, you must open the corresponding file in */dev/vme*.

Using the mmap() Function

When you have successfully opened the device special file, you use the file descriptor as the primary input parameter in a call to the **mmap()** system function.

This function is documented for all its many uses in the mmap(2) reference page. For purposes of mapping a VME device into memory, the parameters should be as follows (using the names from the reference page):

addr	Should be NULL to permit the kernel to choose an address in user process space.
len	The length of the span of VME addresses, as documented in the *iospace* group in the VECTOR line.
prot	PROT_READ for input, PROT_WRITE for output, or the logical sum of those names when the device will be used for both input and output.
flags	MAP_SHARED, with the addition of MAP_PRIVATE if this mapping is not to be visible to child processes created with the **sproc()** function (see the sproc(2) reference page).
fd	The file descriptor returned from opening the device special file in */dev/vme*.
off	The starting VME bus address, as documented in the *iospace* group in the VECTOR line.

The value returned by **mmap()** is the virtual address that corresponds to the starting VME bus address. When the process accesses that address, the access is implemented by data transfer to the VME bus.

Map Size Limits

There are limits to the amount and location of VME bus address space that can be mapped for PIO. The system architecture can restrict the span of mappable addresses, and kernel resource constraints can impose limits.

In all systems that support the VME bus it is possible to map all of A16 space.

In a Silicon Graphics Crimson system, the upper 8 MB of A24 space, from 0x80 0000 to 0xFF FFFF can be mapped. Only a fixed part of the A32 space can be mapped. A total of 256 MB of addresses is divided between the supervisor and nonprivileged A32 spaces on one or two VME buses. For details of the VME bus addresses, see “Crimson Mapping of A32 Space” on page 307. VME devices that respond to addresses in the ranges shown in that section can be mapped for PIO.

In the Silicon Graphics Challenge and Onyx systems, all of A24 and A32 space can be used for PIO mappings, but there is a limit on the size of each map. Each bus mapping uses a hardware register that can span as much as 8 MB of contiguous VME bus addresses—so a single **mmap()** call can map at most 8 MB. There are as many as 12 mapping registers for each bus, so by making successive **mmap()** calls for adjacent 8 MB blocks of VME space you can map up to 96 MB of VME space into user process space from a single bus.

VME PIO Access

Once a VME device has been mapped into memory, your program reads from the device by referencing the mapped address, and writes to the device by storing into the mapped address. Example 4-1 displays a sketch of a hypothetical function that maps a device and copies one register into another.

Example 4-1 Opening and Using a Hypothetical VME Device

```
#define SPECFILE "/dev/vme/vme1a16n"
typedef unsigned short int busdata; /* device data item */
typedef volatile busdata busreg;    /* device register */
#define MAPSIZE 8*sizeof(vmereg)
#define BUSADDR 0xff00
int vmefunc()
{
   busreg *mapped;
   busdata sample;
   int specfd = open(SPECFILE,O_RDWR);
   if (-1 == specfd) return error;
   mapped = mmap(NULL,      /* kernel pick address */
                 REGSIZE,   /* size of mapped area */
                 PROT_READ|PROT_WRITE, /* protection flags */
                 MAP_SHARED, /* mapping flags */
                 specfd,     /* special file */
                 BUSADDR)    /* file offset */
   if (!mapped) return error;
   sample = busreg[0];       /* read A16N at 0xff00 */
   busreg[1] = sample;       /* write A16N at 0xff02 */
}
```

A PIO read is synchronous at the hardware level. The CPU executes a register-load instruction that does not complete until data has been returned from the device, up the system bus, to the CPU (see “CPU Access to Device Registers” on page 10). This can take 1 or 2 microseconds in a Challenge system.

A PIO write is not necessarily synchronous at the hardware level. The CPU executes a register-store instruction that is complete as soon as the physical address and data have been placed on the system bus. The actual VME write operation on the VME bus can take 1 or more microseconds to complete. During that time the CPU can execute dozens or even hundreds more instructions from cache memory.

VME PIO Bandwidth

On a Challenge L or Onyx system, the maximum rate of PIO output is approximately 750K writes per second. The maximum rate of PIO input is approximately 250K reads per second. The corresponding data rate depends on the number of bytes transferred on each operation, as summarized in Table 4-1.

Table 4-1 VME Bus PIO Bandwidth

Data Unit Size	Read	Write
D8	0.25 MB/second	0.75 MB/second
D16	0.5 MB/second	1.5 MB/second
D32	1 MB/second	3 MB/second

Note: The numbers in Table 4-1 were obtained by doing continuous reads, or continuous writes, to a device in the Challenge chassis. When reads and writes alternate, add approximately 1 microsecond for each change of direction. The use of a repeater to extend to an external card cage would add 200 nanoseconds or more to each transfer.

EISA Programmed I/O

For an overview of the EISA bus and its implementation in Silicon Graphics systems, see Chapter 17, "EISA Device Drivers."

Mapping an EISA Device Into Memory

As discussed in "Device Addresses" on page 4, an I/O device is represented as an address or range of addresses in the physical address space. A kernel-level device driver has the ability to set up a mapping between the physical address of a device register and an arbitrary location in the address space of a user-level process. When this has been done, the device register appears to be a variable in memory—the program can assign values to it, or refer to it in expressions.

Learning EISA Device Addresses

In order to map an EISA device for PIO, you must know the following points:

- which EISA bus adapter the device is on

  In all present Silicon Graphics systems, there is only one EISA bus adapter, and its number is 0.
- whether you need access to the EISA bus memory or I/O address space
- the address and length of the desired registers within the address space

You can find all these values by examining files in the */var/sysgen/system* directory, especially the */var/sysgen/system/irix.sm* file, in which each configured EISA device is specified by a VECTOR line. When you examine a VECTOR line, note the following parameter values:

bustype	Specified as *EISA* for EISA devices. The VECTOR statement can be used for other types of buses as well.
adapter	The number of the bus where the device is attached (0).
iospace, *iospace2*, *iospace3*	Each *iospace* group specifies the address space, starting bus address, and the size of a segment of bus address space used by this device.

Within each *iospace* parameter group you find keywords and numbers for the address space and addresses for a device. The following is an example of a VECTOR line (which must be a single physical line in the system file):

```
VECTOR: bustype=EISA module=if_ec3 ctlr=1
iospace=(EISAIO,0x1000,0x1000)
exprobe_space=(r,EISAIO, 0x1c80,4,0x6010d425,0xffffffff)
```

This example specifies a device that resides in the I/O space at offset 0x1000 (the slot-1 I/O space) for the usual length of 0x1000 bytes. The *exprobe_space* parameter suggests that a key device register is at 0x1c80.

Opening a Device Special File

When you know the device addresses, you can open a device special file that represents the correct range of addresses. The device special files for EISA mapping are found in */dev/eisa*.

The naming convention for these files is as follows: Each file is named **eisa***B***a***M*, where

B	is a digit for the bus number (0)
M	is the modifier, either *io* or *mem*

The device special file for the device described by the example VECTOR line in the preceding section would be */dev/vme/eisa0aio*.

In order to map a device on a particular bus and address space, you must open the corresponding file in */dev/eisa*.

Using the mmap() Function

When you have successfully opened the device special file, you use the file descriptor as the primary input parameter in a call to the **mmap()** system function.

This function is documented for all its many uses in the mmap(2) reference page. For purposes of mapping EISA devices, the parameters should be as follows (using the names from the reference page):

addr	Should be NULL to permit the kernel to choose an address in user process space.

len	The length of the span of bus addresses, as documented in the *iospace* group in the VECTOR line.
prot	PROT_READ, or PROT_WRITE, or the logical sum of those names when the device is used for both input and output.
flags	MAP_SHARED, with the addition of MAP_PRIVATE if this mapping is not to be visible to child processes created with the **sproc()** function (see the sproc(2) reference page).
fd	The file descriptor from opening the device special file in */dev/eisa*.
off	The starting bus address, as documented in the *iospace* group in the VECTOR line.

The value returned by **mmap()** is the virtual memory address that corresponds to the starting bus address. When the process accesses that address, the access is implemented by data transfer to the EISA bus.

Note: When programming EISA PIO, you must always be aware that EISA devices generally store 16-bit and 32-bit values in "small-endian" order, with the least-significant byte at the lowest address. This is opposite to the order used by the MIPS CPU under IRIX. If you simply assign to a C unsigned integer from a 32-bit EISA register, the value will appear to be byte-inverted.

EISA PIO Bandwidth

The EISA bus adapter is a device on the GIO bus. The GIO bus runs at either 25 MHz or 33 MHz, depending on the system model. Each EISA device access takes multiple GIO cycles, as follows:

- The base time to do a native GIO read (of up to 64 bits) is 1 microsecond.
- A 32-bit EISA slave read adds 15 GIO cycles to the base GIO read time; hence one EISA access takes 19 GIO cycles, best case.
- A 4-byte access to a 16-bit EISA device requires 10 more GIO cycles to transfer the second 2-byte group; hence a 4-byte read to a 16-bit EISA slave requires 25 GIO cycles.
- Each wait state inserted by the EISA device adds four GIO cycles.

Table 4-2 summarizes best-case (no EISA wait states) data rates for reading and writing a 32-bit EISA device, based on these considerations.

Table 4-2 EISA Bus PIO Bandwidth (32-Bit Slave, 33-MHz GIO Clock)

Data Unit Size	Read	Write
1 byte	0.68 MB/sec	1.75 MB/sec
2 byte	1.38 MB/sec	3.51 MB/sec
4 bytes	2.76 MB/sec	7.02 MB/sec

Table 4-3 summarizes the best-case (no wait state) data rates for reading and writing a 16-bit EISA device.

Table 4-3 EISA Bus PIO Bandwidth (16-Bit Slave, 33-MHz GIO Clock)

Data Unit Size	Read	Write
1 byte	0.68 MB/sec	1.75 MB/sec
2 byte	1.38 MB/sec	3.51 MB/sec
4 bytes	2.29 MB/sec	4.59 MB/sec

VME User-Level DMA

A DMA engine is included as part of each VME bus adapter in a Challenge or Onyx system. The DMA engine is unique to the Challenge architecture. It performs efficient, block-mode, DMA transfers between system memory and VME bus slave cards—cards that would normally be capable of only PIO transfers.

The DMA engine greatly increases the rate of data transfer compared to PIO, provided that you transfer at least 32 contiguous bytes at a time. The DMA engine can perform D8, D16, D32, D32 Block, and D64 Block data transfers in the A16, A24, and A32 bus address spaces.

Using the udmalib Functions

All DMA engine transfers are initiated by a special device driver. However, you do not access this driver through open/read/write system functions. Instead, you program it through a library of functions. The functions are documented in the udmalib(3) reference page. They are used in the following sequence:

1. Call **dma_open()** to initialize action to a particular VME card.
2. Call **dma_allocbuf()** to allocate storage to use for DMA buffers.
3. Call **dma_mkparms()** to create a descriptor for an operation, including the buffer, the length, and the direction of transfer.
4. Call **dma_start()** to execute each transfer. This function does not return until the transfer is complete.

The **dma_start()** function takes the VME bus address of the particular slave device register that will provide or accept a series of data items. Before starting a DMA transfer, and possibly between transfers, you may need to program the VME controller with other commands. You would do this using PIO (see "VME Programmed I/O" on page 62).

Tip: The **dma_start()** function operates synchronously, polling the VME adapter hardware to find out when the DMA transfer is complete. In order to get parallel execution, consider calling **dma_start()** from a separate process.

Buffer Allocation for User DMA

A buffer allocated by **dma_allocbuf()** is rounded up to a multiple of the memory page size, and is locked in memory to avoid page-faults during the DMA transfer. There is some overhead in creating a buffer, so for best performance the program should allocate the required buffers of the necessary size during initialization. However, if the total size of the buffers is a significant fraction of the available real memory, the large number of locked pages can hurt system performance.

You can only use the allocated buffer for DMA; it is not possible to provide your own buffer (for example, a buffer in a shared memory arena) for use by the DMA engine. When the data is produced by one process and written by another, this design can mean that the data has to be copied from an application buffer to the DMA buffer.

Tip: One way to avoid copying is to call **dma_allocbuf()** early, during program setup, before creating subprocesses using **sproc()**. Processes made with **sproc()** share their parent process address space, including buffer space created by **dma_allocbuf()**, so you can have a process generating or consuming data in one part of the allocated buffer space while a different process executes **dma_start()** to write or read data in a different part. It is of course essential to synchronize the use of the different buffer segments so that each area is used by only one process at a time.

Allocation of Descriptors

Each call to **dma_mkparms()** allocates a small block of memory and fills it with constants that describe a particular transfer operation. The primary input to **dma_mkparms()** is a *vme_parms_t* object (declared in *udmalib.h*) containing the following important fields:

vp_block	1 for a block-mode transfer; 0 for a normal transfer.
vp_datumsz	The width of transfer units: VME_DS_BYTE 8-bit transfers) VME_DS_HALFWORD (16-bit transfers) VME_DS_WORD (32-bit transfers), or VME_DS_DBLWORD (64-bit transfers).
vp_dir	The direction of transfer: either VME_READ (from the VME bus) or VME_WRITE (to the VME bus).
vp_throt	For a block-mode transfer, the number of bytes to transfer in one burst without yielding the bus. Set to either VME_THROT_256 (the usual size, supported by most VME slaves that allow block transfer) or VME_THROT_2048 to allow up to 2,048 bytes per burst.
vp_release	The bus arbitration mode: VME_REL_RWD to release the bus as soon as the transfer is over, or VME_REL_ROR to release only when another bus master wants the bus. Use VME_REL_ROR for best speed when no bus masters are on the bus. Use VME_REL_RWD when other bus masters may be present and active.
vp_addrmod	The address modifier value that selects the target VME address space. Names of the form VME_*AMOD are declared for these values in *sys/vmereg.h*, for example VME_A16NPAMOD for the nonpriviliged A16 space.

During initialization you call **dma_mkparms()** to create a descriptor for each unique combination of parameters that your program will use. In each call to **dma_start()**, you pass the descriptor that contains the appropriate set of parameters. A descriptor can be used in multiple **dma_start()** calls.

Advantages of User DMA

The **dma_start()** function and its related functions operate in user space; they do not make system calls to the kernel. This has two important effects. First, overhead is reduced, since there are no mode switches between user and kernel, as there are for **read()** and **write()**. This is important, because the DMA engine is often used for frequent, small inputs and outputs.

Second, **dma_start()** does not block the calling process, in the sense of suspending it and possibly allowing another process to use the CPU. It waits in a test loop until the operation is complete. As you can infer from Table 4-4, typical transfer times range from 50 to 250 microseconds. You can calculate the approximate duration of a call to **dma_start()** based on the amount of data and the operational mode.

You can use the udmalib functions to access a VME Bus Master device, if the device can respond in slave mode. However, this would normally be less efficient than using the Master device's own DMA circuitry.

While you can initiate only one DMA engine transfer per bus, it is possible to program a DMA engine transfer from each bus in the system, concurrently.

DMA Engine Bandwidth

The maximum performance of the DMA engine for D32 transfers is summarized in Table 4-4. Performance with D64 Block transfers is somewhat less than twice the rate shown in Table 4-4. Transfers for larger sizes are faster because the setup time is amortized over a greater number of bytes.

Table 4-4 VME Bus Bandwidth, DMA Engine, D32 Transfer

Transfer Size	Read	Write	Block Read	Block Write
32	2.8 MB/sec	2.6 MB/sec	2.7 MB/sec	2.7 MB/sec
64	3.8 MB/sec	3.8 MB/sec	4.0 MB/sec	3.9 MB/sec
128	5.0 MB/sec	5.3 MB/sec	5.6 MB/sec	5.8 MB/sec
256	6.0 MB/sec	6.7 MB/sec	6.4 MB/sec	7.3 MB/sec
512	6.4 MB/sec	7.7 MB/sec	7.0 MB/sec	8.0 MB/sec

Table 4-4 (continued) VME Bus Bandwidth, DMA Engine, D32 Transfer

Transfer Size	Read	Write	Block Read	Block Write
1024	6.8 MB/sec	8.0 MB/sec	7.5 MB/sec	8.8 MB/sec
2048	7.0 MB/sec	8.4 MB/sec	7.8 MB/sec	9.2 MB/sec
4096	7.1 MB/sec	8.7 MB/sec	7.9 MB/sec	9.4 MB/sec

Note: The throughput that can be achieved in VME DMA is very sensitive to several factors:

- The other activity on the VME bus.
- The blocksize (larger is better).
- Other overhead in the loop requesting DMA operations.

  The loop used to generate the figures in Table 4-4 contained no activity except calls to **dma_start()**.
- the response time of the target VME board to a read or write request, in particular the time from when the VME adapter raises Data Strobe (DS) and the time the slave device raises Data Acknowledge (DTACK).

  For example, if the slave device takes 500 ns to raise DTACK, there will always be fewer than 2 M data transfers per second.

Example User DMA Function

The hypothetical function displayed in Example 4-2 is called to perform a series of DMA transfers from a specified device address. The code does not reflect any device initialization; all setup is presumably done by the caller. The code does not show the processing of the data, which is done in the hypothetical function **processOneBuffer()**.

Example 4-2 User-Level DMA Access to VME

```
/*
|| This function assumes that any device programming needed to
|| prepare the device for input has been done (using PIO) before
|| the function is called.  The bus number and device address
|| are function parameters.
*/
```

```
#define BLOCK_SIZE_TO_USE 4096
#include <udmalib.h>
#include <sys/vmereg.h>

extern void processOneBuffer(void *pBuffer);

int
readBlocks(int iBusNum, __uint32_t uiDevAddress)
{
    udmaid_t   *hEngine;    /* handle returned by dma_open */
    void       *pDMAbuffer; /* pointer from dma_allocbuf */
    vmeparms_t sParms;      /* operation parms for dma_mkparms */
    udmaprm_t  *hParms;     /* handle returned by dma_mkparms */
    int        iStartCode;  /* return code of dma_start */
/*
|| Open the DMA engine. Terminate if it won’t open.
*/
    hEngine = dma_open(DMA_VMEBUS, iBusNum);
    if (!hEngine)
    {
        perror(“dma_open”);
        return(-1);
    }
/*
|| Allocate a special buffer for I/O. If that fails,
|| release the engine and terminate.
*/
    hDMAbuffer = dma_allocbuf(hEngine, BLOCK_SIZE_TO_USE);
    if (!hDMAbuffer)
    {
        perror(“dma_allocbuf”);
        dma_close(hEngine);
        return(-2);
    }
/*
|| Set up the VME parameters and “make” them.  A different set
|| of parameters is needed for each combination of vmeparms_t
|| values, buffer, and size.  This example uses only one set.
*/
    sParms.vp_block = 0;             /* this device does not do blocks */
    sParms.vp_datumsz = VME_DS_WORD; /* this is a 32-bit device */
    sParms.vp_dir = VME_READ;        /* input operation */
    sParms.vp_throt = VME_THROT_256; /* smaller burst size */
    sParms.vp_release = VME_REL_ROR; /* release on request */
    sParms.vp_addrmod = VME_A32NPAMOD; /* address modifier */
    hParms = dma_mkparms(hEngine, &sParms, pDMAbuffer, BLOCK_SIZE_TO_USE);
    if (!hParms)
    {
        perror("dma_mkparms");
        dma_freebuf(pDMAbuffer);
        dma_close(hEngine);
        return(-3);
    }
/*
|| Read and process blocks until error.
*/
    for(iStartCode=0;iStartCode==0;)
    {
         iStartCode = dms_start(hEngine, uiDevAddress, hParms);
         if (!iStartCode)
            processOneBuffer(pDMAbuffer);
    }
/*
|| Clean up and exit.
*/
    dma_freeparms(hEngine, hParms);
    dma_freebuf(pDMAbuffer);
    dma_close(hEngine);
    return 0;
}
```

Chapter 5

User-Level Access to SCSI Devices

IRIX contains a programming library, called *dslib*, that allows you to control SCSI devices from a user-level process. This chapter documents the functions in dslib, including the following topics:

- "Overview of the dsreq Driver" on page 78 gives a summary of the features and use of the generic SCSI device driver.
- "Generic SCSI Device Special Files" on page 78 documents the format of the names and major and minor numbers of generic SCSI files.
- "The dsreq Structure" on page 82 gives details of the request structure that is the primary input to the generic SCSI driver.
- "Testing the Driver Configuration" on page 88 documents the use of the DS_CONF **ioctl()** operation.
- "Using dslib Functions" on page 90 describes the functions that make it simpler to use the generic SCSI driver.
- "Using the Special DS_RESET and DS_ABORT Calls" on page 89 describes two special functions of the generic SCSI driver.

You must understand the SCSI interface in order to command a SCSI device. For several SCSI information resources, see "Other Sources of Information" on page xxxvii.

If you are specifically interested in using audio data from a CDROM or DAT drive, you should use the special-purpose libraries for CDROM and DAT that are included in the IRIS Digital Media Development Environment. These libraries are built upon the generic SCSI driver, but provide convenient, audio-oriented functions. For more information on these libraries, see the *IRIS Digital Media Programming Guide*, document number 008-1799-040.

If your interest is in controlling SCSI devices at the kernel level, see Part V, "SCSI Device Drivers."

Overview of the dsreq Driver

IRIX includes a generic SCSI device driver, the *dsreq* driver, through which a user-level program can issue SCSI commands to SCSI devices. This is a character device driver that supports only **open()**, **close()** and **ioctl()** operations (see "Kinds of Kernel-Level Drivers" on page 45, and also the open(2), close(2) and ioctl(2) reference pages).

The formal documentation of the *dsreq* driver is found in the ds(7) reference page. In order to invoke its services, you prepare a *dsreq* data structure describing the operation and pass it to the device driver using an **ioctl()** call. The device driver issues the SCSI command you specify, and sleeps until it has completed. Then it returns the status in the *dsreq* structure.

You can request operations for input and output as well as issuing control and diagnostic commands. The *dsreq* structure for input and output operations specifies a buffer in memory for data transfer. The *dsreq* driver handles the task of locking the buffer into memory (if necessary) and managing a DMA transfer of data.

The programming interface supported by the generic SCSI driver is quite primitive. A library of higher-level functions makes it easier to use. This library is formally documented in the dslib(3) reference page, and is described under "Using dslib Functions" on page 90.

Generic SCSI Device Special Files

The creation and use of device special files is discussed under "Device Special Files" on page 32. A device special file represents a device, and is the mechanism for associating a device with a kernel-level device driver.

The device special files in the */dev/scsi* directory are all associated with the *dsreq* driver. A basic set of these names is created automatically by the */dev/MAKEDEV* script (see "The Script MAKEDEV" on page 36). You have to create additional device special files if you need to control logical units other than logical unit 0.

Form of Names in /dev/scsi

Each device special filename in the */dev/scsi* directory reflects the values of the device's adapter (bus) number, SCSI ID, and logical unit number (LUN).

Tip: The character between the SCSI ID and the LUN in these names is the letter "l." When reading or copying these device names, take care not to write a digit 1 instead. This is a frequent error.

Names of SCSI Devices on a SCSI Bus

Devices attached directly to a SCSI bus have names in this form:

sc	Prefix "sc" for SCSI attachment.
0 to 137	Number of the SCSI adapter, typically 0 or 1.
d	Constant letter "d" for device.
0 to 7 (to 15 for wide SCSI)	SCSI ID of the target device or control unit, as set by switches on the device itself.
l (letter ell)	Constant letter "l" for logical unit.
0 to 7	Logical unit number (LUN) of this device, typically 0.

A typical device name would be */dev/scsi/sc1d3l0* meaning a SCSI device configured as ID 3 on SCSI bus 1. Either this device has no logical units, or this is the first logical unit on device 3.

Names of SCSI Devices on the Jag (VME Bus) Controller

Machines in the Challenge and Onyx systems can optionally have SCSI devices attached to the VME bus through a bridge using the *jag* device driver. These devices are also represented in */dev/scsi* with names of the following form:

jag	Prefix “jag” for VME/SCSI attachment.
0 to 4	Number of the VME adapter, typically 0 or 1.
d	Constant letter “d” for device.
0 to 7 (to 15 for wide SCSI)	SCSI ID of the target device or control unit, as set by switches on the device itself.
l (letter ell)	Constant letter “l” for logical unit.
0 to 7	Logical unit number (LUN) of this device, typically 0.

A typical device name would be */dev/scsi/jag1d3l0* meaning a SCSI device configured as ID 3 on VME bus 1. Either the device has no logical units, or this is the first logical unit on device 3.

Major and Minor Device Numbers in /dev/scsi

Device special files in */dev/scsi* have one of the following major device numbers:

- 195 for devices on a SCSI bus (files */dev/scsi/sc**).
- 196 for devices on a *jag* (VME) SCSI bridge (files */dev/scsi/jag**).

The minor number of these files encodes the adapter number, the SCSI ID, and the LUN, using the bit assignments shown in Figure 5-1.

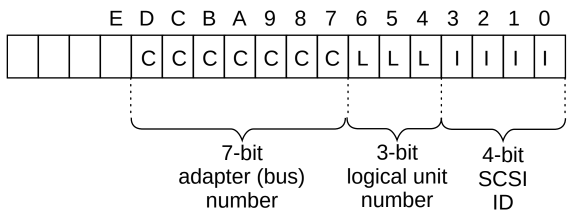
E D C B A 9 8 7 6 5 4 3 2 1 0
C C C C C C C L L L I I I I
7-bit adapter (bus) number
3-bit logical unit number
4-bit SCSI ID

Figure 5-1 Bit Assignments in SCSI Device Minor Numbers

Creating Additional Names in /dev/scsi

The script */dev/MAKEDEV*, which runs each time the system boots, creates 16 files for each existing SCSI or jag bus. These files represent the possible SCSI ID numbers 0-15 on each bus, with a logical number of 0. If you want to control a device with LUN 0, the device special file exists.

In order to control a device with a LUN of 1-7, you must create an additional device special file, using the *mknode* or *install* command (see the install(1) reference page). For example, before you can operate logical unit 2 of device 5 on SCSI bus 1, you must create */dev/scsi/sc1d5l2* using a command such as

```
install -F /dev/scsi -m 600 -u root -g sys \
-chr 195,165 sc1d5l2
```

Relationship to Other Device Special Files

The files in */dev/scsi* describe many of the same devices that are described by files in */dev/dsk*, */dev/tape*, and other directories. There is a security exposure in that a user-level program could use a */dev/scsi* file to do almost anything to a disk or tape, including total erasure.

The *dsreq* device driver forces exclusivity with itself; that is, a given */dev/scsi* file can be opened only by one process at a time. However, a device could be open through the *dsreq* driver at the same time it is open by another process, or by a filesystem, through a different device special file and device driver. For example, a disk volume could be simultaneously open through the name */dev/scsi/sc0d0l0* and through */dev/rdsk/dks0d1s0*.

The process that opens a generic SCSI device can request exclusivity using the O_EXCL option to **open()**. In that case, the open is rejected when the device is already open through another driver; and no other driver can open the device until the generic device file is closed.

The dsreq Structure

The primary input to most *dsreq* **ioctl()** calls, as well as the primary input to most dslib functions, is the *dsreq* structure. This structure is declared in */usr/include/sys/dsreq.h*, a header file that rewards careful study.

The important fields of the *dsreq* structure are shown in Table 5-1. Some of the field values are expanded in the following topics. The *sys/dsreq.h* file declares macros for access to many of the fields. Use these macros (listed in Table 5-1) in both expressions and assignments in order to insulate your code against future changes.

Table 5-1 Fields of the dsreq Structure

Field Name	Macro	Purpose
ds_flags	FLAGS(dp)	Bits used to determine device driver actions. See "Values for ds_flags" on page 83.
ds_time	TIME(dp)	Timeout value in milliseconds. If the command does not complete, it is ended with an error code. The driver sets a default of 5000 (5 seconds) when this is set to zero. **dsopen()** initializes it to 10000.
ds_private	PRIVATE(dp)	Field for use by the calling program. **dsopen()** uses this field to point to its "context" data (see "Using dsopen() and dsclose()" on page 92).
ds_cmdbuf	CMDBUF(dp)	Address of SCSI command string to be sent.
ds_cmdlen	CMDLEN(dp)	Length of the SCSI command string.
ds_databuf	DATABUF(dp)	Address of a single data buffer. See "Data Transfer Options" on page 85.
ds_datalen	DATALEN(dp)	Length of data buffer.
ds_sensebuf	SENSEBUF(dp)	Address to receive sense data after an error.
ds_senselen	SENSELEN(dp)	Length of sense buffer in bytes.
ds_iovbuf	IOVBUF(dp)	Address of an *iov_t* structure. See "Data Transfer Options" on page 85.
ds_iovlen	IOVLEN(dp)	Length of data described by *ds_iovbuf*.
ds_link		This field is not supported, and should be zero-filled.

Table 5-1 (continued) Fields of the dsreq Structure

Field Name	Macro	Purpose
ds_synch		This field is not supported, and should be zero-filled.
ds_revcode		Intended for the version code of the *dsreq* driver, not currently set to a useful value.
ds_ret	RET(dp)	Return code for the requested operation. See Table 5-3 on page 85.
ds_status	STATUS(dp)	SCSI status byte from the operation. See Table 5-4 on page 87.
ds_msg	MSG(dp)	The first byte of a message returned by the target. See Table 5-5 on page 87.
ds_cmdsent	CMDSENT(dp)	Length of command string actually sent (same as *ds_cmdlen*, unless an error occurs).
ds_datasent	DATASENT(dp)	Length of data transferred.
ds_sensesent	SENSESENT(dp)	Length of sense data received.

The dslib library contains functions to simplify the preparation and execution of a *dsreq* request; see "Using dslib Functions" on page 90.

Values for ds_flags

The possible flag values in the *ds_flags* field are listed in Table 5-2. The flag values are designed for the most flexible, capable type of bus, device, and device driver. Not all values are supported, and different host adapters can support different combinations.

Table 5-2 Flag Values for ds_flags

Constant Name	Supported by Any Driver?	Meaning When Set to 1
DSRQ_ASYNC	Yes	Return at once, do not sleep until the operation is complete.
DSRQ_SENSE	Yes	Get sense data following an error on the requested command.
DSRQ_TARGET	No	Act as the SCSI target, not the SCSI initiator.

Table 5-2 (continued) Flag Values for ds_flags

Constant Name	**Supported by Any Driver?**	**Meaning When Set to 1**
DSRQ_SELATN	Yes	Select with ATN.
DSRQ_DISC	Yes	Allow identify disconnect.
DSRQ_SYNXFR	Yes	Negotiate a synchronous transfer if possible. Needed only to switch into synchronous mode. Repeated negotiation is wasteful.
DSRQ_ASYNXFR	Yes	Negotiate an asynchronous transfer. Needed only to return to asynch after a synchronous transfer. Repeated negotiation is wasteful.
DSRQ_SELMSG	No	A specific select is coded in the message. This feature is not supported.
DSRQ_IOV	Yes	Use the *iov_t* from *ds_iovbuf*, not the single buffer from *ds_databuf* (see “Data Transfer Options” on page 85).
DSRQ_READ	Yes	This is a data input command (as opposed to an immediate command or an output).
DSRQ_WRITE	Yes	This is a data output command (as opposed to an immediate command or an input).
DSRQ_MIXRDWR	No	This command can both read and write.
DSRQ_BUF	No	Buffer the input and copy to the supplied buffer, instead of direct input to the buffer.
DSRQ_CALL	No	Notify completion (with DSRQ_ASYNC).
DSRQ_ACKH	No	Hold ACK asserted.
DSRQ_ATNH	No	Hold ATN asserted.
DSRQ_ABORT	No	Send ABORT messages until the bus is clear.Useful only with SCSI commands that have the immediate bit set.
DSRQ_TRACE	Yes	Trace this request (accepted but has no effect).
DSRQ_PRINT	Yes	Print this request (accepted but has no effect).
DSRQ_CTRL1	Yes	Request with host control bit 1.
DSRQ_CTRL2	Yes	Request with host control bit 2.

In order to find out which flags are supported by a particular driver, use the DS_CONF operation (see "Testing the Driver Configuration" on page 88).

Data Transfer Options

When reading or writing data, you have two design options:

- You can transfer a single segment of data directly between the device and a buffer you supply (set neither DSRQ_BUF nor DSRQ_IOV).
- You can transfer segments of data between the device and a series of one or more memory locations based on an *iov_t* object (set DSRQ_IOV).

All read/write requests are done using DMA. The "scatter/gather" support of DSRQ_IOV is presently restricted to only one memory segment, so it is not greatly different from single-buffer I/O. If you elect to use it, the *iov_t* structure is declared in *sys/iov.h* (see also the part of the read(2) reference page that deals with the **readv()** function).

During a direct transfer using either a single buffer or scatter/gather, the data buffer spaces are locked in memory.

The maximum amount of data you can transfer in one operation is set by the host adapter driver for the bus, and can be retrieved with an **ioctl()** (see "Testing the Driver Configuration" on page 88). The maximum length for a buffered transfer is returned by the same **ioctl()**. It can be less than the direct-transfer size because there may be a limit on the size of kernel memory that can be allocated.

Return Codes and Status Values

A zero return code in the *ds_ret* field signifies success. The possible nonzero return codes are summarized in Table 5-3 and are declared in *sys/dsreq.h*. Not all return codes are possible with every driver.

Table 5-3 Return Codes From SCSI Operations

Constant Name	Meaning
DSRT_DEVSCSI	General failure from SCSI driver.
DSRT_MULT	General software failure, typically a SCSI-bus request.

Table 5-3 (continued) Return Codes From SCSI Operations

Constant Name	Meaning
DSRT_CANCEL	Operation cancelled in host adapter driver.
DSRT_REVCODE	Software level mismatch, recompile application.
DSRT_AGAIN	Try again, recoverable SCSI-bus error.
DSRT_HOST	Failure reported by host adapter driver for the bus in use.
DSRT_NOSEL	No unit responded to select.
DSRT_SHORT	Incomplete transfer (not an error). See *ds_datasent*.
DSRT_OK	Not returned at this time.
DSRT_SENSE	Command returned with status; sense data successfully retrieved from SCSI host (see *ds_sensesent*).
DSRT_NOSENSE	Command with status, error occurred while trying to get sense data from SCSI host.
DSRT_TIMEOUT	Command did not complete in the time allowed by *ds_timeout*.
DSRT_LONG	Data transfer overran bounds (*ds_datalen*).
DSRT_PROTO	Miscellaneous protocol failure.
DSRT_EBSY	Busy dropped unexpectedly; protocol error.
DSRT_REJECT	Message rejected; protocol error.
DSRT_PARITY	Parity error on SCSI bus; protocol error.
DSRT_MEMORY	Memory error in system memory.
DSRT_CMDO	Protocol error during command phase.
DSRT_STAI	Protocol error during status phase.
DSRT_UNIMPL	Command not implemented; protocol error.

The possible SCSI status value in the *ds_status* field are summarized in Table 5-4.

Table 5-4 SCSI Status Codes

Constant Name	Meaning
STA_GOOD	The target has successfully completed the SCSI command.
STA_CHECK	An error or exception was detected. Sense was attempted if DSRQ_SENSE was specified.
STA_BUSY	Command not attempted; addressed unit is busy.
STA_IGOOD	Linked SCSI command completed.
STA_RESERV	Command aborted because it tried to access a logical unit or an extent within a logical unit that reserves that type of access to another SCSI device.

The possible SCSI message byte values in the *ds_msg* field are summarized in Table 5-5.

Table 5-5 SCSI Message Byte Values

Constant Name	Meaning
MSG_COMPL	Command complete.
MSG_XMSG	Extended message (only byte returned).
MSG_SAVEP	Initiator should save data pointers.
MSG_RESTP	Initiator restore data pointers.
MSG_DISC	Disconnect.
MSG_IERR	Initiator detected error.
MSG_ABORT	Abort.
MSG_REJECT	Optional message rejected, not supported.
MSG_NOOP	Empty message.
MSG_MPARITY	Parity error during Message In phase.
MSG_LINK	Linked command complete.
MSG_LINKF	Linked command complete with flag.

Table 5-5 (continued) SCSI Message Byte Values

Constant Name	Meaning
MSG_BRESET	Bus device reset.
MSG_IDENT	Value 0x80, first of the 0x80-0xFF identifier messages.

Testing the Driver Configuration

Different buses have different host adapter drivers that can have different features. The *dsreq* device driver supports an **ioctl()** call that retrieves the configuration of the driver for the bus where the device resides. This call fills in the fields of a structure of type *dsconf* (declared in *sys/dsreq.h*) listed in Table 5-6.

Table 5-6 Fields of the dsconf Structure

Field Name	Contents
dsc_flags	DSRQ flags honored by this driver (see Table 5-2 on page 83)
dsc_preset	DSRQ preset values (defaults) that are merged with the input *ds_flags* using logical OR in any request.
dsc_bus	Number of this SCSI bus, as encoded in the device minor number.
dsc_imax	Maximum target ID for this bus (7 for SCSI, 15 for wide SCSI).
dsc_lmax	Maximum number LUN values per ID on this bus.
dsc_iomax	Maximum length of a single I/O transfer.
dsc_biomax	Maximum length of a buffered I/O transfer.

The code in Example 5-1 shows a function that tests if a particular flag is supported by a particular bus. The input arguments are a file descriptor for an open device special file, and a flag value (or values) from *sys/dsreq.h*.

Example 5-1 Testing the Generic SCSI Configuration

```
uint
test_dsreq_flags(int dev_fd, uint flag)
{
   dsconf_t config;
   int ret;
   ret = ioctl(dev_fd, DS_CONF, &config);
   if (!ret) { /* no problem in ioctl */
      return (flag & config.dsc_flags);
   } else { /* ioctl failure */
      return 0; /* not supported, it seems */
   }
}
```

A program could use the function in Example 5-1 to find out if a particular feature is supported. For example, a test of support for the DSRQ_SYNXFER feature could be coded as follows:

```
if (test_dsreq_flags(the_dev, DSRQ_SYNXFER)) {
   /* synchronous negotiation is supported */...
```

Using the Special DS_RESET and DS_ABORT Calls

Two special functions of the generic SCSI driver are available only as **ioctl()** calls, not through dslib functions.

Using DS_ABORT

The DS_ABORT **ioctl()** sends a SCSI ABORT message to the bus, target, and LUN defined by the file descriptor. The resulting status is returned in the *dsreq* that is also specified. The host adapter driver waits until no commands are pending on that bus, so there is no point in using this function to cancel anything but an immediate command such as a rewind. And example of this call is as follows:

```
ioctl(dev_fd, DS_ABORT, &some_dsreq);
```

Using DS_RESET

The DS_RESET **ioctl()** function causes a reset of the SCSI bus specified by the file descriptor. The resulting status is returned in the *dsreq* that is also specified. This powerful operation should be used with great care, because it terminates all pending activity on the bus.

Using dslib Functions

The functions in the dslib library are built upon calls to the dsreq device driver, and simplify the process of allocating a dsreq structure, setting values in it, and executing commands. The formal documentation of the library is found in dslib(3). The source code is distributed with the system in the */usr/share/src/irix/examples/scsi* directory so that you can read and extend it. (This directory installs as part of the irix_dev software component, and the examples directory does not install by default.)

dslib Functions

In order to use the functions in the library, you include */usr/include/dslib.h* in your code, and link with the *-lds* option so as to link */usr/lib/libds.so*. Then the functions summarized in Table 5-7 are available.

Table 5-7 dslib Function Summary

Function Name	Purpose	Page
ds_ctostr	Look up a string in a table using an integer key.	page 95
ds_vtostr	Look up a string in a table using an integer key.	page 95
dsopen	Open a device special file and allocate a *dsreq* for use with it.	page 92
dsclose	Free the *dsreq* structure and close the device.	page 92
doscsireq	Perform an operation on a device as specified in a *dsreq*.	page 93
filldsreq	Set values in fields of a *dsreq* structure.	page 93
fillg0cmd	Set up the *dsreq* structure for a group 0 SCSI command.	page 94
fillg1cmd	Set up the *dsreq* structure for a group 1 SCSI command.	page 94

Table 5-7 (continued) dslib Function Summary

Function Name	Purpose	Page
inquiry12	Issue an Inquiry command and retrieve information from the device concerning such things as its type.	page 96
modeselect15	Issue a group 0 Mode Select command to a SCSI device.	page 96
modesense1a	Send a group 0 Mode Sense command to a device to retrieve a parameter page from the device.	page 97
read08	Issue a group 0 Read command in disk-drive form.	page 98
readextended28	Issue a group 1 Read command in disk-drive form.	page 98
readcapacity25	Issue a Read Capacity command.	page 99
requestsense03	Issue a Request Sense command and test or probe for the device.	page 100
reserveunit16	Issue a Reserve Unit command.	page 100
releaseunit17	Issue a Release Unit command.	page 100
senddiagnostic1d	Issue a Send Diagnostic command to test if the device or the SCSI bus is online, or run a self-test on the device.	page 101
testunitready00	Issue a Test Unit Ready command to the SCSI device.	page 102
write0a	Issue a group 0 Write command to the SCSI device.	page 102
writeextended2a	Issue an extended Write command to the SCSI device.	page 102

Using dsopen() and dsclose()

The **dsopen()** function opens a device special file for a generic SCSI device, and allocates a *dsreq* structure initialized for use with that device. The function prototype is

```
struct dsreq* dsopen(char *opath, int oflags);
```

The arguments are

opath	The name of the device special file as a character string, for example "/dev/scsi/jag0d7l0" (see "Form of Names in /dev/scsi" on page 79).
oflags	The *oflag* value expected by **open()** when opening this device special file. O_EXCL has special meaning; see "Relationship to Other Device Special Files" on page 81.

If the **open()** call fails or memory cannot be allocated, the function returns NULL. Otherwise it allocates a *dsreq* structure as well as generous buffers for command and sense strings. The following fields of the *dsreq* are initialized:

ds_time	Set to 10000 (10 second timeout).
ds_private	Set to the address of the context that contains the *dsreq* as well as the command and sense buffers.
ds_cmdbuf	Set to the address of the command buffer.
ds_cmdlen	Set to the length of the allocated command buffer.
ds_sensebuf	Set to the address of the allocated sense buffer.
ds_senselen	Set to the length of the sense buffer.

Other fields of the *dsreq* are cleared to zero.

Note: Other functions in dslib assume that a *dsreq* has been initialized by **dsopen()**. In particular they assume the *ds_private* value points to a context block. You should not attempt to use any *dsreq* structure with a dslib function except one returned by **dsopen()**; and you should not use a *dsreq* opened for one file with another file.

The **dsclose()** function releases the *dsreq* structure and close the device. Its prototype is

```
void dsclose(struct dsreq *dsp);
```

The only argument is the *dsreq* created by **dsopen()**.

Issuing a Request With doscsireq()

The **doscsireq()** function issues a SCSI request by passing a *dsreq* to the SCSI device driver using an **ioctl()** call. The *dsreq* must have been prepared completely beforehand. The function prototype is

```
int doscsireq(int fd, struct dsreq *dsp);
```

The arguments are as follows:

fd	The file descriptor for the open device file.
dsp	The address of the *dsreq* prepared by **dsopen()**.

Normally the returned value is the SCSI status byte. When the requested operation ends with Busy or Reserve Conflict status, the function sleeps 2 seconds and tries the operation up to four times. The returned value is -1 when the device driver rejects the ioctl() or the third retry ends in failure.

SCSI Utility Functions

The functions **filldsreq()**, **fillg0cmd()**, **fillg1cmd()**, **ds_vtostr()**, and **ds_ctostr()** are not oriented toward particular SCSI operations, but are used to construct your own task-oriented SCSI functions.

Using filldsreq()

The **filldsreq()** function is used to set the *ds_flags*, *ds_databuf*, and *ds_datalen* members of a *dsreq* structure. Its prototype is

```
void filldsreq(struct dsreq *dsp, uchar_t *data,long datalen, long flags)
```

The arguments are as follows:

dsp	The address of a *dsreq* prepared by dsopen().
data	The address of a buffer area.
datalen	The length of the buffer area.
flags	Flag values for *ds_flags* (see “Values for ds_flags” on page 83).

The bits in *flags* are added to *ds_flags* with an OR; they do not replace the contents of the field.

Note: Besides the specified values, the function also sets 10000 in *ds_timeout* and clears *ds_link*, *ds_synch*, and *ds_ret* to zero.

Using fillg0cmd() and fillg1cmd()

The **fillg0cmd()** function stores a group 0 (6-byte) SCSI command in a command buffer. The **fillg1cmd()** stores a group 1 (10-byte) SCSI command in the buffer. Both functions set the *ds_cmdbuf* and *ds_cmdlen* fields of a *dsreq*. The function prototypes are:

```
void fillg0cmd(struct dsreq *dsp, uchar_t *cmdbuf, b0, ..., b5)
void fillg1cmd(struct dsreq *dsp, uchar_t *cmdbuf, b0, ..., b9)
```

The arguments are as follows:

dsp	The address of any *dsreq*.
cmdbuf	The address of a buffer to receive the command string.
b0, *b1*,...	Expressions for the successive bytes of a SCSI command.

In typical use, the arguments are as follows:

dsp	The address of a *dsreq* initialized by **dsopen()**.
cmdbuf	The command buffer allocated by **dsopen()**, whose address is stored in the *ds_cmdbuf* field of the *dsreq*.
b0	A SCSI command verb expressed as one of the constants declared in dslib.h, for example G0_INQU.

A typical call resembles the following:

```
fillg0cmd(dsp, (uchar_t *)CMDBUF(dsp), G0_INQU, 1, inq_page, 0,
B1(datalen),0);
```

The macros B1(), B2(), and B4() defined in *sys/dsreq.h* are useful for expressing halfword and word values as byte sequences.

Using ds_vtostr() and ds_ctostr()

The dslib library module contains six static tables that can be used to convert between numeric values and character strings for message display. The tables are summarized in Table 5-8. The table definitions are in the source file *dstab.c*.

Table 5-8 Lookup Tables in dslib

External Name	Type	Table Contents
cmdnametab	vtab	Names for SCSI command bytes, for example “Test Unit.”
cmdstatustab	vtab	Names for SCSI status byte codes, for example “BUSY.”
dsrqnametab	vtab	Descriptions of flag values from *ds_flags*, for example “select with (without) atn” for DSRQ_SELATN.
dsrtnametab	vtab	Descriptions of return values in ds_ret, for example “parity error on SCSI bus” for DSRT_PARITY.
msgnametab	vtab	Descriptions of SCSI message bytes, for example “Save Pointers.”
sensekeytab	ctab	Descriptions of SCSI sense byte values, for example “Illegal Request.”

The **ds_vtostr()** function searches any of the five vtab tables for the string matching an integer key. The **ds_ctostr()** function searches a ctab (currently, only sensekeytab is a ctab) for the string matching a key. The function prototypes are

```
char * ds_vtostr(unsigned long v, struct vtab *table);
char * ds_ctostr(unsigned long v, struct ctab *table);
```

Each function searches the specified table for a row containing the numeric value *v*, and returns address of the corresponding string. If there is no such row, the functions return the address of a zero-length string.

Using Command-Building Functions

The remaining functions in dslib each construct and execute a specific type of common SCSI command. Each function follows this general pattern:

1. Use **fillg0cmd()** or **fillg1cmd()** to set up the command string, based on the function's arguments.
2. Use **filldsreq()** to set up the remaining fields of the *dsreq* structure.
3. Execute the command using **doscsireq()**.
4. Return the value returned by **doscsireq()**.

You can construct similar, additional functions using the utility functions in this same way. In particular you are likely to need to construct your own function to issue Read commands.

inquiry12()—Issue an Inquiry Command

The **inquiry12()** function prepares and issues an Inquiry command to retrieve device-specific information. The function prototype is

```
int inquiry12(struct dsreq *dsp, caddr_t data, long datalen, int vu);
```

The arguments are as follows:

dsp	The address of a *dsreq* structure prepared by **dsopen()**.
data	The address of a buffer to receive the inquiry response.
datalen	The length of the buffer, at least 36 and typically 64.
vu	The least-significant two bits are used to set the vendor-specific bits in the Control byte in the command.

modeselect15()—Issue a Group 0 Mode Select Command

The **modeselect15()** function prepares and issues a group 0 Mode Select command. This command is used to control a variety of standard and vendor-specific device parameters. Typically, **modesense1A()** is first used to retrieve the current parameters. The function prototype is

```
int modeselect15(struct dsreq *dsp, caddr_t data, long datalen,
                 int save, int vu);
```

The arguments are as follows:

dsp	The address of a *dsreq* structure prepared by **dsopen()**.
data	The address of a mode data page to send.
datalen	The length of the data.
save	The least significant bit sets the SP bit in the command.
vu	The least-significant two bits are used to set the vendor-specific bits in the Control byte in the command.

modesense1a()—Send a Group 0 Mode Sense Command

The **modesense1a()** function prepares and issues a group 0 Mode Sense command to a SCSI device to retrieve a page of device-dependent information. The function prototype:

```
int modesense1a(struct dsreq *dsp, caddr_t data, long datalen,
                int pagectrl, int pagecode, int vu);
```

The arguments are as follows:

dsp	The address of a *dsreq* structure prepared by **dsopen()**.
data	The address of a buffer to receive the page of data.
datalen	The length of the buffer.
pagectrl	The least significant 2 bits are set as the PCF bits in the command.
pagecode	The least significant 6 bits are set as the page number.
vu	The least-significant two bits are used to set the vendor-specific bits in the Control byte in the command.

For reference, the PCF codes are as follows:

0	Current values.
1	Changeable values.
2	Default values.
3	Saved values.

For reference, some page numbers are as follows:

0	Vendor unique.
1	Read/write error recovery.
2	Disconnect/reconnect.
3	Direct access device format; parallel interface; measurement units.
4	Rigid disk geometry; serial interface.
5	Flexible disk; printer options.
6	Optical memory.
7	Verification error.
8	Caching.
9	Peripheral device.
63 (0x3f)	Return all pages supported.

read08() and readextended28()—Issue a Read Command

The **read08()** and **readextended28()** functions prepare and issue particular forms of SCSI Read commands. The Read and extended Read commands have so many variations that it is unlikely that either of these functions will work with your device. However, you can use them as models to build additional variations on Read. Do not preempt the function names.

The function prototypes are

```
int
read08(struct dsreq *dsp, caddr_t data, long datalen,
              long lba, int vu);
int
readextended28(struct dsreq *dsp, caddr_t data, long datalen,
              long lba, int vu);
```

The arguments are as follows:

dsp	The address of a *dsreq* structure prepared by **dsopen()**.
data	The address of a buffer to receive the data.
datalen	The length of the buffer (not exceeding 255 for **read08**)
lba	The logical block address for the start of the read (not exceeding 16 bits for **read08**)
vu	The least-significant two bits are used to set the vendor-specific bits in the Control byte in the command.

The functions set the transfer length in the command to the number of bytes given by datalen. This is often incorrect; many devices want a number of blocks of some size. Function **read08()** sets only 16 bits from lba as the logical block number, although the SCSI command format permits another 5 bits to be encoded in the command. For these and other reasons you are likely to need to create customized Read functions of your own.

readcapacity25()—Issue a Read Capacity Command

The **readcapacity25()** function prepares and issues a Read Capacity command to a SCSI device. The function prototype is

```
int
readcapacity25(struct dsreq *dsp, caddr_t data, long datalen,
               long lba, int pmi, int vu);
```

The arguments are as follows:

dsp	The address of a *dsreq* structure prepared by **dsopen()**.
data	The address of a buffer to receive the capacity data.
datalen	The length of the buffer, typically 8.
lba	Last block address, 0 unless *pmi* is nonzero.
pmi	The least-significant bit is used to set the partial medium indicator (PMI) bit of the command.
vu	The least-significant two bits are used to set the vendor-specific bits in the Control byte in the command.

When *pmi* is 0, *lba* should be given as 0 and the command returns the device capacity. When *pmi* is 1, the command returns the last block following block *lba* before which a delay (seek) will occur.

requestsense03()—Issue a Request Sense Command

The **requestsense03()** function prepares and issues a Request Sense command. If you include DSRQ_SENSE in the *flag* argument to **doscsireq()**, a Request Sense is sent automatically after an error in a command. The function prototype is

```
int
requestsense03(struct dsreq *dsp, caddr_t data,
               long datalen, int vu);
```

The arguments are:

dsp	The address of a *dsreq* structure prepared by **dsopen()**.
data	The address of a buffer to receive the sense data.
datalen	The length of the buffer.
vu	The least-significant two bits are used to set the vendor-specific bits in the Control byte in the command.

reserveunit16() and releaseunit17()—Control Logical Units

The **reserveunit16()** function prepares and issues a Reserve Unit command to reserve a logical unit, causing it to return Reservation Conflict status to requests from other initiators. The **releaseunit17()** function prepares and issues a Release Unit command to release a reserved unit. The function prototypes are

```
int
reservunit16(struct dsreq *dsp, caddr_t data, long datalen,
            int tpr, int tpdid, int extent, int res_id, int vu);
int
releaseunit17(struct dsreq *dsp,
            int tpr, int tpdid, int extent, int res_id, int vu);
```

The arguments are as follows:

dsp	The address of a *dsreq* structure prepared by **dsopen()**.
data	The address of data to send with the Reserve Unit. (This may be NULL for **reservunit16()** which does not normally transfer data.)
datalen	The length of the data (typically 0).
tpr	The least-significant bit is used to set the Third-Party Reservation bit in the command: 1 means the reservation is on behalf of another initiator.
tpdid	The device ID for the device to hold the reservation: 0 unless *tpr* is 1.
extent	The least-significant bit sets the least-significant bit of byte 1 of the command string.
res_id	Passed as byte 2 of the command string.
vu	The least-significant two bits are used to set the vendor-specific bits in the Control byte in the command.

senddiagnostic1d()—Issue a Send Diagnostic Command

The **senddiagnostic1d()** function prepares and issues a Send Diagnostic command. The function prototype is

```
int
senddiagnostic1d(struct dsreq *dsp, caddr_t data, long datalen,
                 int self, int dofl, int uofl, int vu);
```

The arguments are as follows:

dsp	The address of a *dsreq* structure prepared by **dsopen()**.
data	The address of a page or pages of diagnostic parameter data to be sent.
datalen	The length of the data (0 if none).
self	The least-significant bit sets the Self Test (ST) bit in the command: 1 means return status from the self-test; 0 means hold the results.
dofl	The least-significant bit sets the Device Offline bit of the command: 1 authorizes tests that can change the status of other logical units.

uofl	The least-significant bit sets the Unit Offline bit of the command: 1 authorizes tests that can change the status of the logical unit.
vu	The least-significant two bits are used to set the vendor-specific bits in the Control byte in the command.

When *self* is 1, the status reflects the success of the self-test. You should either set the DSRQ_SENSE flag in the *dsreq* so that if the self-test fails, a Sense command will be issued, or be prepared to call **requestsense03()**. When *self* is 0, you can use a Read Diagnostic command to return detailed results of the test (however, dslib does not contain a predefined function for Read Diagnostic).

testunitready00—Issue a Test Unit Ready Command

The **testunitready00()** function prepares and issues a Test Unit Ready command to a SCSI device. The function prototype is

```
int
testunitready00(struct dsreq *dsp);
```

This function is reproduced here in Example 5-2 as an example of how other command-oriented functions can be created.

Example 5-2 Code of the testunitread00() Function

```
int
testunitready00(struct dsreq *dsp)
{
   fillg0cmd(dsp, CMDBUF(dsp), G0_TEST, 0, 0, 0, 0, 0);
   filldsreq(dsp, 0, 0, DSRQ_READ|DSRQ_SENSE);
   return(doscsireq(getfd(dsp), dsp));
}
```

write0a() and writeextended2a()—Issue a Write Command

The **write0a()** function prepares and issues a group 0 Write command. The **writeextended2a()** function prepares and issues an extended (10-byte) Write command. As with Read commands (see “read08() and readextended28()—Issue a Read Command” on page 98), Write commands have many device-specific features, and you will very likely have to create your own customized version of these functions.

The function prototypes are

```
int
write0a(struct dsreq *dsp, caddr_t data, long datalen,
        long lba, int vu);
int
writeextended2a(struct dsreq *dsp, caddr_t data, long datalen,
        long lba, int vu);
```

The arguments are as follows:

dsp	The address of a *dsreq* structure prepared by **dsopen()**.
data	The address of the data to be sent.
datalen	The length of the data (at most 255 for **write0a**)
lba	The logical block address (at most 16 bits for **write0a**)
vu	The least-significant two bits are used to set the vendor-specific bits in the Control byte in the command.

Example dslib Program

The program in Example 5-3 illustrates the use of the dslib functions. This is an edited version of a program that can be obtained in full from Dave Olson's home page, *http://reality.sgi.com/employees/olson/Olson/index.html*.

Example 5-3 Program That Uses dslib Functions

```
#ident "scsicontrol.c: $Revision $"

#include <sys/types.h>
#include <stddef.h>
#include <stdlib.h>
#include <unistd.h>
#include <ctype.h>
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <dslib.h>
```

```
typedef struct
{
    unchar  pqt:3;  /* peripheral qual type */
    unchar  pdt:5;  /* peripheral device type */
    unchar  rmb:1,  /* removable media bit */
        dtq:7;  /* device type qualifier */
    unchar  iso:2,  /* ISO version */
        ecma:3, /* ECMA version */
        ansi:3; /* ANSI version */
    unchar  aenc:1, /* async event notification supported */
        trmiop:1,   /* device supports 'terminate io process msg */
        res0:2, /* reserved */
        respfmt:3;  /* SCSI 1, CCS, SCSI 2 inq data format */
    unchar  ailen;  /* additional inquiry length */
    unchar  res1;   /* reserved */
    unchar  res2;   /* reserved */
    unchar  reladr:1,   /* supports relative addressing (linked cmds) */
        wide32:1,   /* supports 32 bit wide SCSI bus */
        wide16:1,   /* supports 16 bit wide SCSI bus */
        synch:1,    /* supports synch mode */
        link:1, /* supports linked commands */
        res3:1, /* reserved */
        cmdq:1, /* supports cmd queuing */
        softre:1;   /* supports soft reset */
    unchar  vid[8]; /* vendor ID */
    unchar  pid[16];    /* product ID */
    unchar  prl[4]; /* product revision level*/
    unchar  vendsp[20]; /* vendor specific; typically firmware info */
    unchar  res4[40];   /* reserved for scsi 3, etc. */
    /* more vendor specific information may follow */
} inqdata;

struct msel {
    unsigned char rsv, mtype, vendspec, blkdesclen; /* header */
    unsigned char dens, nblks[3], rsv1, bsize[3];   /* block desc */
    unsigned char pgnum, pglen; /* modesel page num and length */
    unsigned char data[240]; /* some drives get upset if no data requested
        on sense*/
};

#define hex(x) "0123456789ABCDEF" [ (x) & 0xF ]

/* only looks OK if nperline a multiple of 4, but that's OK.
 * value of space must be 0 <= space <= 3;
 */
```

```
void
hprint(unsigned char *s, int n, int nperline, int space)
{
    int   i, x, startl;

    for(startl=i=0;i<n;i++)  {
        x = s[i];
        printf("%c%c", hex(x>>4), hex(x));
        if(space)
            printf("%.*s", ((i%4)==3)+space, "    ");
        if ( i%nperline == (nperline - 1) ) {
            putchar('\t');
            while(startl < i) {
                if(isprint(s[startl]))
                    putchar(s[startl]);
                else
                    putchar('.');
                startl++;
            }
            putchar('\n');
        }
    }
    if(space && (i%nperline))
        putchar('\n');
}

/* aenc, trmiop, reladr, wbus*, synch, linkq, softre are only valid if
 * if respfmt has the value 2 (or possibly larger values for future
 * versions of the SCSI standard). */

static char pdt_types[][16] = {
    "Disk", "Tape", "Printer", "Processor", "WORM", "CD-ROM",
    "Scanner", "Optical", "Jukebox", "Comm", "Unknown"
};
#define NPDT (sizeof pdt_types / sizeof pdt_types[0])

void
printinq(struct dsreq *dsp, inqdata *inq, int allinq)
{

    if(DATASENT(dsp) < 1) {
        printf("No inquiry data returned\n");
        return;
    }
    printf("%-10s", pdt_types[(inq->pdt<NPDT) ? inq->pdt : NPDT-1]);
```

```
if (DATASENT(dsp) > 8)
    printf("%12.8s", inq->vid);
if (DATASENT(dsp) > 16)
    printf("%.16s", inq->pid);
if (DATASENT(dsp) > 32)
    printf("%.4s", inq->prl);
printf("\n");
if(DATASENT(dsp) > 1)
    printf("ANSI vers %d, ISO ver: %d, ECMA ver: %d; ",
        inq->ansi, inq->iso, inq->ecma);
if(DATASENT(dsp) > 2) {
    unchar special = *(inq->vid-1);
    if(inq->respfmt >= 2 || special) {
        if(inq->respfmt < 2)
            printf("\nResponse format type %d, but has "
              "SCSI-2 capability bits set\n", inq->respfmt);

        printf("supports: ");
        if(inq->aenc)
            printf(" AENC");
        if(inq->trmiop)
            printf(" termiop");
        if(inq->reladr)
            printf(" reladdr");
        if(inq->wide32)
            printf(" 32bit");
        if(inq->wide16)
            printf(" 16bit");
        if(inq->synch)
            printf(" synch");
        if(inq->synch)
            printf(" linkedcmds");
        if(inq->cmdq)
            printf(" cmdqueing");
        if(inq->softre)
            printf(" softreset");
    }
    if(inq->respfmt < 2) {
        if(special)
            printf(".  ");
        printf("inquiry format is %s",
            inq->respfmt ? "SCSI 1" : "CCS");
    }
}
putchar('\n');
```

```
    printf("Device is  ");
    /*  do test unit ready only if inquiry successful, since many
        devices, such as tapes, return inquiry info, even if
        not ready (i.e., no tape in a tape drive). */
    if(testunitready00(dsp) != 0)
        printf("%s\n",
            (RET(dsp)==DSRT_NOSEL) ? "not responding" : "not ready");
    else
        printf("ready");
    printf("\n");
}

/* inquiry cmd that does vital product data as spec'ed in SCSI2 */
int
vpinquiry12( struct dsreq *dsp, caddr_t data, long datalen, char vu,
  int page)
{
  fillg0cmd(dsp, (uchar_t *)CMDBUF(dsp), G0_INQU, 1, page, 0, B1(datalen),
    B1(vu<<6));
  filldsreq(dsp, (uchar_t *)data, datalen, DSRQ_READ|DSRQ_SENSE);
  return(doscsireq(getfd(dsp), dsp));
}

int
startunit1b(struct dsreq *dsp, int startstop, int vu)
{
  fillg0cmd(dsp,(uchar_t *)CMDBUF(dsp),0x1b,0,0,0,(uchar_t)startstop,B1(vu<<6));
  filldsreq(dsp, NULL, 0, DSRQ_READ|DSRQ_SENSE);
  dsp->ds_time = 1000 * 90; /* 90 seconds */
  return(doscsireq(getfd(dsp), dsp));
}


int
myinquiry12(struct dsreq *dsp, uchar_t *data, long datalen, int vu, int neg)
{
  fillg0cmd(dsp, (uchar_t *)CMDBUF(dsp), G0_INQU, 0,0,0, B1(datalen), B1(vu<<6));
  filldsreq(dsp, data, datalen, DSRQ_READ|DSRQ_SENSE|neg);
  dsp->ds_time = 1000 * 30; /* 90 seconds */
  return(doscsireq(getfd(dsp), dsp));
}

int
dsreset(struct dsreq *dsp)
{
  return ioctl(getfd(dsp), DS_RESET, dsp);
}

void
usage(char *prog)
{
    fprintf(stderr,
    "Usage: %s [-i (inquiry)] [-e (exclusive)] [-s (sync) | -a (async)]\n"
    "\t[-l (long inq)] [-v (vital proddata)] [-r (reset)] [-D (diagselftest)]\n"
    "\t[-H (halt/stop)] [-b blksize]\n"
    "\t[-g (get host flags)] [-d (debug)] [-q (quiet)] scsidevice [...]\n",
        prog);
    exit(1);
}

main(int argc, char **argv)
{
    struct dsreq *dsp;
    char *fn;
    /* int because they must be word aligned. */
    int errs = 0, c;
    int vital=0, doreset=0, exclusive=0, dosync=0;
    int dostart = 0, dostop = 0, dosenddiag = 0;
    int doinq = 0, printname = 1;
    unsigned bsize = 0;
    extern char *optarg;
    extern int optind, opterr;

    opterr = 0; /* handle errors ourselves. */
    while ((c = getopt(argc, argv, "b:HDSaserdvlgCiq")) != -1)
    switch(c) {
    case ‘i’:
        doinq = 1;  /* do inquiry */
        break;
    case ‘D’:
        dosenddiag = 1;
        break;
    case ‘r’:
        doreset = 1;    /* do a scsi bus reset */
        break;
    case ‘e’:
        exclusive = O_EXCL;
        break;
    case ‘d’:
        dsdebug++;  /* enable debug info */
        break;
    case ‘q’:
        printname = 0;  /* print devicename only if error */
        break;
    case ‘v’:
        vital = 1;  /* set evpd bit for scsi 2 vital product data */
        break;
    case ‘H’:
        dostop = 1; /* send a stop (Halt) command */
        break;
    case ‘S’:
        dostart = 1;    /* send a startunit/spinup command */
        break;
    case ‘s’:
        dosync = DSRQ_SYNXFR;   /* attempt to negotiate sync scsi */
        break;
    case ‘a’:
        dosync = DSRQ_ASYNXFR;  /* attempt to negotiate async scsi */
        break;
    default:
        usage(argv[0]);
    }

    if(optind >= argc || optind == 1)   /* need at 1 arg and one option */
        usage(argv[0]);

    while (optind < argc) { /* loop over each filename */
        fn = argv[optind++];
        if(printname) printf("%s:  ", fn);
        if((dsp = dsopen(fn, O_RDONLY|exclusive)) == NULL) {
            /* if open fails, try pre-pending /dev/scsi */
            char buf[256];
            strcpy(buf, "/dev/scsi/");
            if((strlen(buf) + strlen(fn)) < sizeof(buf)) {
                strcat(buf, fn);
                dsp = dsopen(buf, O_RDONLY|exclusive);
            }
            if(!dsp) {
                if(!printname) printf("%s:  ", fn);
                fflush(stdout);
                perror("cannot open");
                errs++;
                continue;
            }
        }
```

```
/* try to order for reasonableness; reset first in case
 * hung, then inquiry, etc. */

if(doreset) {
    if(dsreset(dsp) != 0) {
        if(!printname) printf("%s:  ", fn);
        printf("reset failed: %s\n", strerror(errno));
        errs++;
    }
}

if(doinq) {
    int inqbuf[sizeof(inqdata)/sizeof(int)];
    if(myinquiry12(dsp, (uchar_t *)inqbuf, sizeof inqbuf, 0, dosync)) {
        if(!printname) printf("%s:  ", fn);
        printf("inquiry failure\n");
        errs++;
    }
    else
        printinq(dsp, (inqdata *)inqbuf, 0);
}

if(vital) {
    unsigned char *vpinq;
    int vpinqbuf[sizeof(inqdata)/sizeof(int)];
    int vpinqbuf0[sizeof(inqdata)/sizeof(int)];
    int i, serial = 0, asciidef = 0;
    if(vpinquiry12(dsp, (char *)vpinqbuf0,
        sizeof(vpinqbuf)-1, 0, 0)) {
        if(!printname) printf("%s:  ", fn);
        printf("inquiry (vital data) failure\n");
        errs++;
        continue;
    }
    if(DATASENT(dsp) <4) {
        printf("vital data inquiry OK, but says no"
            "pages supported (page 0)\n");
        continue;
    }
    vpinq = (unsigned char *)vpinqbuf0;
    printf("Supported vital product pages: ");
    for(i = vpinq[3]+3; i>3; i--) {
        if(vpinq[i] == 0x80)
            serial = 1;
        if(vpinq[i] == 0x82)
```

```
                asciidef = 1;
            printf("%2x  ", vpinq[i]);
        }
        printf("\n");
        vpinq = (unsigned char *)vpinqbuf;
        if(serial) {
            if(vpinquiry12(dsp, (char *)vpinqbuf,
                sizeof(vpinqbuf)-1, 0, 0x80) != 0) {
                if(!printname) printf("%s:  ", fn);
                printf("inquiry (serial #) failure\n");
                errs++;
            }
            else if(DATASENT(dsp)>3) {
                printf("Serial #: ");
                fflush(stdout);
                /* use write, because there may well be
                 *nulls; don't bother to strip them out */
                write(1, vpinq+4, vpinq[3]);
                printf("\n");
            }
        }
        if(asciidef) {
            if(vpinquiry12(dsp, (char *)vpinqbuf,
            sizeof(vpinqbuf)-1, 0, 0x82) != 0) {
            if(!printname) printf("%s:  ", fn);
            printf("inquiry (ascii definition) failure\n");
            errs++;
            }
            else if(DATASENT(dsp)>3) {
            printf("Ascii definition: ");
            fflush(stdout);
            /* use write, because there may well be
             *nulls; don't bother to strip them out */
            write(1, vpinq+4, vpinq[3]);
            printf("\n");
            }
        }
    }

    if(dostop && startunit1b(dsp, 0, 0)) {
        if(!printname) printf("%s:  ", fn);
        printf("stopunit fails\n");
        errs++;
    }
```

```
        if(dostart && startunit1b(dsp, 1, 0)) {
            if(!printname) printf("%s:  ", fn);
            printf("startunit fails\n");
            errs++;
        }

        if(dosenddiag && senddiagnostic1d(dsp, NULL, 0, 1, 0, 0, 0)) {
            if(!printname) printf("%s:  ", fn);
            printf("self test fails\n");
            errs++;
        }
dsclose(dsp);
    }
    return(errs);
}
```

Chapter 6

Control of External Interrupts

Some Silicon Graphics computer systems can generate and receive external interrupt signals. These are simple, two-state signal lines that cause an interrupt in the receiving system.

The external interrupt hardware is managed by a kernel-level device driver that is distributed with IRIX and automatically configured when the system supports external interrupts. The driver provides two abilities to user-level processes:

- the ability to change the state of an outgoing interrupt line, so as to interrupt the system to which the line is connected)
- the ability to capture an incoming interrupt signal with low latency

Note: Some software for external interrupt support is closely tied to the hardware of the system. The features described in this chapter are hardware-dependent and in many cases cannot be ported from one system type to another without making software changes.

Beginning in IRIX 6.2, there is a hardware-independent way to capture incoming external interrupts: the user-level interrupt facility (ULI). To learn about ULI, see Chapter 7, “User-Level Interrupts,” especially “Registering an External Interrupt Handler” on page 127.

External Interrupts in Challenge and Onyx Systems

The hardware architecture of the Challenge/Onyx series supports external interrupt signals as follows:

- Four jacks for outgoing signals are available on the master IO4 board. A user-level program can change the level of these lines individually.
- Two jacks for incoming interrupt signals are also provided. The input lines are combined with logical OR and presented as a single interrupt; a program cannot distinguish one input line from another.

The electrical interface to the external interrupt lines is documented at the end of the ei(7) reference page.

A program controls the outgoing signals by interacting with the external interrupt device driver. This driver is associated with the device special file */dev/ei*, and is documented in the ei(7) reference page.

Generating Outgoing Signals

A program can generate an outgoing signal on any one of the four external interrupt lines. To do so, it first opens */dev/ei*. Then it can apply **ioctl()** on the file descriptor to switch the outgoing lines. The principal functions are summarized in Table 6-1.

Table 6-1 Functions for Outgoing External Signals

Operation	**Typical ioctl() Call**
Set pulse width to *N* microseconds.	ioctl(*eifd*, EIIOCSETOPW, *N*)
Return current output pulse width.	ioctl(*eifd*,EIIOCGETOPW,&*var*)
Send a pulse on some lines *M*.[a]	ioctl(*eifd*, EIIOCSTROBE, *M*)
Set a high (active, asserted) level on lines *M*.	ioctl(*eifd*, EIIOCSETHI, *M*)
Set a low (inactive, deasserted) level on lines *M*.	ioctl(*eifd*, EIIOCSETLO, *M*)

a. *M* is an unsigned integer whose bits 0, 1, 2, and 3 correspond to the external interrupt lines 0, 1, 2, and 3. At least one bit must be set.

In the Challenge and Onyx series, the level on an outgoing external interrupt line is set directly from a CPU. The device driver generates a pulse (function EIIOCSTROBE) by asserting the line, then spinning in a disabled loop until the specified pulse time has elapsed, and finally deasserting the line. Clearly, if the pulse width is set to much more than the default of 5 microseconds, pulse generation could interfere with the handling of other interrupts.

The calls to assert and deassert the outgoing lines (functions EIIOCSETHI and EIIOCSETLO) do not tie up the kernel. However, direct assertion of the outgoing signal should be used only when the desired signal frequency and pulse duration are measured in milliseconds or seconds. No user-level program can hope to generate pulse durations measured in microseconds by calling these functions. For one thing, the minimum guaranteed interrupt service time is 200 microseconds. An interrupt occurring between the call to assert the signal and the call to deassert it will stretch the intended pulse width by at least 200 microseconds.

Receiving Incoming External Interrupts

An important feature of the Challenge and Onyx external input line is that interrupts are triggered by the level of the signal, not by the transition from deasserted to asserted. This means that, whenever external interrupts are enabled and any of the input lines are in the asserted state, an external interrupt occurs. The interface between your program and the external interrupt device driver is affected by this hardware design. The functions for incoming signals are summarized in Table 6-2.

Table 6-2 Functions for Incoming External Interrupts

Operation	Typical ioctl() Call

Enable receipt of external interrupts.	ioctl(*eifd*, EIIOCENABLE)
	eicinit();

Disable receipt of external interrupts.	ioctl(*eifd*, EIIOCDISABLE)
Block in the driver until an interrupt occurs.	ioctl(*eifd*,EIIOCRECV,&*eiargs*)
Request a signal when an interrupt occurs.	ioctl(*eifd*, EIIOCSTSIG, *signumber*)
Wait in an enabled loop for an interrupt.	eicbusywait(1)
Set expected pulse width of incoming signal.	ioctl(*eifd*, EIIOCSETIPW,*microsec*)

Table 6-2 (continued) Functions for Incoming External Interrupts

Operation	**Typical ioctl() Call**
Set expected time between incoming signals.	ioctl(*eifd*, EIIOCSETSPW,*microsec*)

Return current expected time values.	ioctl(*eifd*, EIIOCGETIPW,&*var*)
	ioctl(*eifd*, EIIOCGETSPW,&*var*)

Detecting Invalid External Interrupts

The external interrupt handler maintains two important numbers:

- the expected input pulse duration in microseconds
- the minimum pulse-to-pulse interval, called the "stuck" pulse width because it is used to detect when an input line is "stuck" in the asserted state

When the external interrupt device driver is entered to handle an interrupt, it waits with interrupts disabled until time equal to the expected input pulse duration has passed since the interrupt occurred. The default pulse duration is 5 microseconds, and it typically takes longer than this to recognize and process the interrupt, so no time is wasted in the usual case. However, if a long expected pulse duration is set, the interrupt handler might have to waste some cycles waiting for the end of the pulse.

At the end of the expected pulse duration, the interrupt handler counts one external interrupt and returns to the kernel, which enables interrupts and returns to the interrupted process.

Normally the input line is deasserted within the expected duration. However, if the input line is still asserted when the time expires, another external interrupt occurs immediately. The external interrupt handler notes that it has been reentered within the "stuck" pulse time since the last interrupt. It assumes that this is still the same input pulse as before. In order to prevent the stuck pulse from saturating the CPU with interrupts, the interrupt handler disables the external interrupt signal.

External interrupts remain disabled for one timer tick (10 milliseconds). Then the device driver reenables external interrupts. If an interrupt occurs immediately, the input line is still asserted. The handler disables external interrupts for another, longer delay. It continues to delay and to test the input signal in this manner until it finds the signal deasserted.

Setting the Expected Pulse Width

You can set the expected input pulse width and the minimum pulse-to-pulse time using **ioctl()**. For example, you could set the expected pulse width using a function like the one shown in Example 6-1.

Example 6-1 Function to Test and Set External Interrupt Pulse Width

```
int setEIPulseWidth(int eifd, int newWidth)
{
   int oldWidth;
   if ( (0==ioctl(eifd, EIIOCGETIPW, &oldWidth))
   &&   (0==ioctl(eifd, EIIOCSETIPW, newWidth)) )
      return oldWidth;
   perror("setEIPulseWidth");
   return 0;
}
```

The function retrieves the original pulse width and returns it. If either **ioctl()** call fails, it returns 0.

The default pulse width is 5 microseconds. Pulse widths shorter than 4 microseconds are not recommended.

Since the interrupt handler keeps interrupts disabled for the duration of the expected width, you want to specify as short an expected width as possible. However, it is also important that all legitimate input pulses terminate within the expected time. When a pulse persists past the expected time, the interrupt handler is likely to detect a “stuck” pulse, and disable external interrupts for several milliseconds.

Set the expected pulse width to the duration of the longest valid pulse. It is not necessary to set the expected width longer than the longest valid pulse. A few microseconds are spent just reaching the external interrupt handler, which provides a small margin for error.

Setting the Stuck Pulse Width

You can set the minimum pulse-to-pulse width using code like that in Example 6-1, using constants EIIOCGETSPW and EIIOCSETSPW.

The default stuck-pulse time is 500 microseconds. Set this time to the nominal pulse-to-pulse interval, minus the largest amount of “jitter” that you anticipate in the signal. In the event that external signals are not produced by a regular oscillator, set this value to the expected pulse width plus the duration of the shortest expected “off” time, with a minimum of twice the expected pulse width.

For example, suppose you expect the input signal to be a 10 microsecond pulse at 1000 Hz, both numbers plus or minus 10%. Set the expected pulse width to 10 microseconds to ensure that all pulses are seen to complete. Set the stuck pulse width to 900 microseconds, so as to permit a legitimate pulse to arrive 10% early.

Receiving Interrupts

The external interrupt device driver offers you four different methods of receiving notification of an interrupt. You can

- have a signal of your choice delivered to your process
- test for interrupt-received using either an **ioctl()** call or a library function
- sleep until an interrupt arrives or a specified time expires
- spin-loop until an interrupt arrives

You would use a signal when interrupts are infrequent and irregular, and when it is not important to know the precise arrival time. Signal latency can be milliseconds long in some extreme cases. For this reason, it would not be wise to use signals to handle a high rate of interrupts, nor to expect to time interrupts closely. Use a signal when, for example, the external interrupt represents a human-operated switch or some kind of out-of-range alarm condition.

The **ioctl(**EIIOCRECV**)** call tests for an interrupt, or suspends the caller until an interrupt arrives or a timeout expires (see the ei(7) reference page for details). Use this method when interrupts arrive frequently enough that it is worthwhile devoting a process to handling them.

The **ioctl()** call is a fairly costly method of polling, since it entails entry to and exit from the kernel. This is not significant if the polling is infrequent—for example, if one poll call is made every 60th of a second.

When the **ioctl()** call is used to wait for an interrupt, an unknown amount of time can pass between the moment when the interrupt handler unblocks the process and the moment when the kernel dispatches the process. This makes it impossible to timestamp the interrupt at the microsecond level.

In order to detect an incoming interrupt with minimum latency, use the library function **eicbusywait()** (see the ei(7) reference page). This function does not switch into kernel mode, so it is a very fast method of polling for an interrupt. However, if you ask it to wait until an interrupt occurs, it waits by spinning on a repeated test for an interrupt. This monopolizes the CPU, so this form of waiting can be used reliably only by a process running in an isolated CPU. (If there are other processes to run, or interrupts to handle, the polling loop in **eicbusywait()** shares the CPU and can be preempted for long periods.)

The benefit of **eicbusywait()** is that, in an isolated, nonpreemptive CPU, control returns to the calling process in negligible time after the interrupt handler detects the interrupt, so the interrupt can be handled quickly and timed precisely.

Chapter 7

User-Level Interrupts

The user-level interrupt (ULI) facility complements the ability to perform programmed I/O (PIO) from user space. You can use PIO to initiate a device action that leads to a device interrupt, and you can intercept and handle the interrupt in your program. For function prototypes and other details, see the uli(3) reference page.

Overview of ULI

In the past, PIO could only be synchronous: the program wrote to a device register, then polled the device until the operation was complete. With ULI, the program can manage a device that causes interrupts on the VME bus. You set up a handler function within your program. The handler is called whenever the device causes an interrupt.

In IRIX 6.2, user-level interrupts are supported for VME-bus devices, and for external interrupts on the Challenge and Onyx systems.

When using ULI with a VME device, you use VME PIO to initiate device actions and to transfer data to and from device registers (see “VME Programmed I/O” on page 62). When using ULI to trap external interrupts, you enable the interrupts with **ioctl()** calls to the external interrupt handler (see Chapter 6, “Control of External Interrupts”).

The User Level Interrupt Handler

The ULI handler is a function within your program. It is entered asynchronously from the IRIX kernel's interrupt-handling code. The kernel transfers from the kernel address space into the user process address space, and makes the call in user (not privileged kernel) execution mode. Despite this more complicated linkage, you can think of the ULI handler as a subroutine of the kernel's interrupt handler. As such, the performance of the ULI handler has a direct bearing on the system's interrupt response time.

Like the kernel's interrupt handler, the ULI handler can be entered at almost any time, regardless of what code is being executed by the CPU—a process of your program or a process of another program, executing in user space or in a system function. In fact, the ULI handler can be entered from one CPU while the your program executes concurrently in another CPU. Your normal code and your ULI function can execute in true concurrency, accessing the same global variables.

Restrictions on the ULI Handler

Because the ULI handler is called in a special context of the kernel's interrupt handler, it is severely restricted in the system facilities it can use. The list of features the ULI handler may not use includes the following:

- Any use of floating-point calculations. The kernel does not take time to save floating-point registers during an interrupt trap. The floating-point coprocessor is turned off and an attempt to use it in the ULI handler causes a SIGILL (illegal instruction) exception.
- Any use of IRIX system functions. Because most of the IRIX kernel runs with interrupts enabled, the ULI handler could be entered while a system function was already in progress. System functions do not support reentrant calls. In addition, many system functions can sleep, which an interrupt handler may not do.
- Any storage reference that causes a page fault. The kernel cannot suspend the ULI handler for page I/O. Reference to an unmapped page causes a SIGSEGV (memory fault) exception.
- Any calls to C library functions that might violate the preceding restrictions.

There are very few library functions that you can be sure will use no floating point and make no system calls. Unfortunately, library functions such as **sprintf()**, often used in debugging, must be avoided.

In essence, the ULI handler should only do such things as

- store data in program variables to record the interrupt event

  A ring buffer is a data structure that is suitable for concurrent access.

- program the device as required to clear the interrupt or acknowledge it

  The ULI handler has access to the whole program address space, including any mapped-in devices, so it can perform PIO loads and stores.

- post a semaphore to wake up the main process

  This must be done using a ULI function.

Planning for Concurrency

Since the ULI handler can interrupt the program at any point, or run concurrently with it, the program must be prepared for concurrent execution. There are two areas to consider: global variables, and library routines.

Debugging With Interrupts

The asynchronous, possibly concurrent entry to the ULI handler can confuse a debugging monitor such as *dbx*. Some strategies for dealing with this are covered in the uli(3) reference page.

Declaring Global Variables

When variables can be modified by both the main process and the ULI handler, you must take special care to avoid race conditions.

An important step is to specify -D_SGI_REENTRANT_FUNCTIONS to the compiler, so as to get the reentrant versions of the C library functions. This ensures that, if the main process and the ULI handler both enter the C library, there will be no collision over global variables.

You can declare the global variables that are shared with the ULI handler with the keyword “volatile,” so that the compiler will generate code to load the variables from memory on each reference. However, the compiler never holds global values in registers over a function call, and you almost always have a function call (such as **ULI_block_intr()**) preceding a test of a shared global variable.

Using Multiple Devices

The ULI feature allows a program to open more than one interrupting device. You register a handler for each device. However, the program can only wait for a specific interrupt to occur; that is, the **ULI_sleep()** function specifies the handle of one particular ULI handler. This does not mean that the main program must sleep until that particular interrupt handler is entered, however. Any ULI handler can waken the main program, as discussed under “Interacting With the Handler” on page 128.

Setting Up

A program initializes for ULI in the following major steps:

1. Open the device special file for the device.
2. For a VME device, map the device addresses into process memory (see “Mapping a VME Device Into Process Address Space” on page 62.
3. Lock the program address space in memory.
4. Initialize any data structures used by the interrupt handler.
5. Register the interrupt handler.
6. Interact with the device and the interrupt handler.

Any time after the handler has been registered, an interrupt can occur, causing entry to the ULI handler.

Opening the Device Special File

Devices are represented by device special files (see “Device Special Files” on page 32). In order to gain access to a device, you open the device special file that represents it.

The file that represents the external interrupt lines on a Challenge or Onyx system is */dev/ei*. It can be opened by more than one process at a time, under rules that are spelled out in the ei(7) reference page. The methods of opening */dev/ei*, configuring pulse widths, and generating output pulses are covered in Chapter 6, “Control of External Interrupts,” and remain the same.

The files that represent VME control units are */dev/vme/vme**. The rules for opening one of these files and mapping a device into memory are covered under "VME Programmed I/O" on page 62, and remain the same. The difference is that, with ULI, you can map in a device that can cause interrupts.

The program should open the device and (for a VME device) verify that the device exists and is active before proceeding.

Locking the Program Address Space

The ULI handler must not reference a page of program text or data that is not present in memory. You prevent this by locking the pages of the program address space in memory. The simplest way to do this is to call the **plock()** system function:

```
if (plock(PROCLOCK))
{ perror("plock"); exit();}
```

The **plock()** function has two possible difficulties. One is that the calling process must have superuser privilege (see the plock(2) reference page). This may not pose a problem if the program needs superuser privilege in any case, for example in order to open a device special file. The second is that it locks all text and data pages. In a very large program this may be so much memory that system performance is harmed.

The **mpin()** function can be used by unprivileged programs to lock a limited number of pages. The limit is set by the tunable system parameter *maxlkmem*. (Check its value—typically 2000—in */var/sysgen/mtune/kernel*. See the systune(1) reference page for instructions on changing a tunable parameter.)

In order to use **mpin()**, you must specify the exact address ranges to be locked. Provided that the ULI handler refers only to global data and its own code, it is relatively simple to derive address ranges that encompass the needed pages. If the ULI handler calls any library functions, the library DSO needs to be locked as well. The smaller the scope of the ULI handler, the easier it is to use **mpin()**.

Registering the Interrupt Handler

When the program is ready to start operations, it registers its ULI handler. The ULI handler is a function that matches the prototype

```
void function_name(void *arg);
```

The registration function takes arguments with the following purposes:

- the file descriptor of the device special file
- four arguments related to the ULI handler:
  - the address of the handler function
  - an argument value to be passed to the handler on each interrupt—typically a pointer to a work area that is unique to the interrupting device, supposing the program is using more than one device
  - the size, and an optional address, of memory to be used as stack space when calling the handler
- a count of semaphores to be allocated for use with this interrupt

You can ask the ULI support to allocate a stack space by passing a null pointer for the stack argument. When the ULI handler is as simple a function as it normally is, the default stack size of 1024 bytes is ample.

The semaphores are allocated and maintained by the ULI support. They are used to coordinate between the program process and the interrupt handler, as discussed under "Interacting With the Handler" on page 128. You should specify one semaphore for each independent process that can wait for interrupts from this handler. Normally one semaphore is sufficient.

The returned value is a handle that is used to identify this interrupt in other functions. Once registered, the ULI handler remains registered until the program terminates (there is no function for un-registration).

Registering an External Interrupt Handler

The **ULI_register_ei()** function takes the arguments described in the preceding topic. Once it has successfully registered your handler, all external interrupts are directed to that handler.

It is important to realize that, so long as a ULI handler is registered, none of the other interrupt-reporting features supported by the external interrupt device driver (see Chapter 6, "Control of External Interrupts" and the ei(7) reference page) operate any more. These restrictions include the facts that:

- The per-process external interrupt queues are not updated.
- Signals requested by **ioctl(**EIIOCSETSIG**)** are not sent.
- Calls to **ioctl(**EIIOCRECV**)** sleep until they are interrupted by a timeout, a signal, or because the program using ULI terminated and an interrupt arrived.
- Calls to the library function **eicbusywait()** do not terminate.

Clearly you should not use ULI for external interrupts when there are other programs running that also use them.

Registering a VME Interrupt Handler

The **ULI_register_vme()** function takes two additional arguments:

- the interrupt level that the device uses.
- a word that contains, or receives, an interrupt vector number

The interrupt level used by a device is normally set by hardware and documented in the VECTOR line that defines the device (see "Learning VME Device Addresses" on page 62).

Some VME devices have a fixed interrupt vector number; others are programmable. You pass a fixed vector number to the function. If the number is programmable, you pass 0, and the function allocates a number. You must then use PIO to program the vector number into the device.

Interacting With the Handler

The program process and the ULI handler synchronize their actions using two functions.

When the program cannot proceed without an interrupt, it calls **ULI_sleep()**, specifying

- the handle of the interrupt for which to wait
- the number of the semaphore to use for waiting

Typically only one process ever calls **ULI_sleep()** and it specifies waiting on semaphore 0. However, it is possible to have two or more processes that wait. For example, if the device can produce two distinct kinds of interrupts—normal and high-priority, perhaps—you could set up an independent process for each interrupt type. One would sleep on semaphore 0, the other on semaphore 1.

When an ULI handler is entered, it wakes up a program process by calling **ULI_wakeup()**, specifying the semaphore number to be posted. The handler must know which semaphore to post, based on the values it can read from the device or from program variables.

The **ULI_sleep()** call can terminate early, for example if a signal is sent to the process. The process that calls **ULI_sleep()** must test to find the reason the call returned—it is not necessarily because of an interrupt.

The **ULI_wakeup()** function can be called from normal code as well as from a ULI handler function. It could be used within any type of asynchronous callback function to wake up the program process.

The ULI_wakeup() call also specifies the handle of the interrupt. When you have multiple interrupting devices, you have the following design choices:

- You can have one child process waiting on the handler for each device. In this case, each ULI handler specifies its own handle to **ULI_wakeup()**.
- You can have a single process that waits on any interrupt. In this case, the main program specifies the handle of one particular interrupt to **ULI_sleep()**, and every ULI handler specifies that same handle to **ULI_wakeup()**.

Achieving Mutual Exclusion

The program can gain exclusive use of global variables with a call to **ULI_block_intr()**. This function does not block receipt of the hardware interrupt, but does block the call to the ULI handler. Until the program process calls **ULI_unblock_intr()**, it can test and update global variables without danger of a race condition. This period of time should be as short as possible, because it extends the interrupt latency time. If more than one hardware interrupt occurs while the ULI handler is blocked, it will be called for only the last-received interrupt.

There are other techniques for safe handling of shared global variables besides blocking interrupts. One important, and little-known, set of tools is the **test_and_set()** group of functions documented in the test_and_set(3) reference page. These instructions use the Load Linked and Store Conditional instructions of the MIPS instruction set to safely update global variables in various ways.

Sample Program

The program listed in Example 7-1 is a hypothetical example of how user-level interrupts can be used to handle external interrupts in a Challenge and Onyx system.

Example 7-1 Hypothetical ULI Program

```
/* This program demonstrates use of the External Interrupt source
 * to drive a User Level Interrupt.
 *
 * The program requires the presence of an external interrupt cable looped
 * back between output number 0 and one of the inputs on the machine on
 * which the program is run.
 */
#include <sys/ei.h>
#include <sys/uli.h>
#include <sys/lock.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
/* The external interrupt device file is used to access the EI hardware */
#define EIDEV "/dev/ei"
static int eifd;
/* The user level interrupt id. This is returned by the ULI registration
 * routine and is used thereafter to refer to that instance of ULI
 */
static void *ULIid;
/* Variables which are shared between the main process thread and the ULI
 * thread may have to be declared as volatile in some situations. For
 * example, if this program were modified to wait for an interrupt with
 * an empty while() statement, e.g.
 *      while(!intr);
 * the value of intr would be loaded on the first pass and if intr is
 * false, the while loop will continue forever since only the register
 * value, which never changes, is being examined. Declaring the variable
 * intr as volatile causes it to be reloaded from memory on each iteration.
 * In this code however, the volatile declaration is not necessary since
 * the while() loop contains a function call, e.g.
 *   while(!intr)
 *      ULI_sleep(ULIid, 0);
 * The function call forces the variable intr to be reloaded from memory
 * since the compiler cannot determine if the function modified the value
 * of intr. Thus the volatile declaration is not necessary in this case.
 * When in doubt, declare your globals as volatile.
 */
static int intr;
/* This is the actual interrupt service routine. It runs
 * asynchronously with respect to the remainder of this program, possibly
 * simultaneously, on an MP machine. This function must obey the ULI mode
 * restrictions, meaning that it may not use floating point or make
 * any system calls. (Try doing so and see what happens.) Also, this
 * function should be written to execute as quickly as possible, since it
 * runs at interrupt level with lower priority interrupts masked.
 * The system imposes a 1-second time limit on this function to prevent
 * the cpu from freezing if an infinite loop is inadvertently programmed
 * in. Try inserting an infinite loop to see what happens.
 */
static void
intrfunc(void *arg)
{
   /* Set the global flag indicating to the main thread that an
    * interrupt has occurred, and wake it up
    */
   intr = 1;
   ULI_wakeup(ULIid, 0);
}
/* This function creates a new process and from it, generates a
 * periodic external interrupt.
 */
static void
```

```
signaler(void)
{
   int pid;
   if ((pid = fork()) < 0) {
   perror("fork");
   exit(1);
   }
   if (pid == 0) {
      while(1) {
         if (ioctl(eifd, EIIOCSTROBE, 1) < 0) {
            perror("EIIOCSTROBE");
            exit(1);
         }
         sleep(1);
      }
   }
}
/* The main routine sets everything up, then sleeps waiting for the
 * interrupt to wake it up.
 */
int
main()
{
   /* open the external interrupt device */
   if ((eifd = open(EIDEV, O_RDONLY)) < 0) {
      perror(EIDEV);
      exit(1);
   }
   /* Set the target cpu to which the external interrupt will be
    * directed. This is the cpu on which the ULI handler function above
    * will be called. Note that this is entirely optional, but if
    * you do set the interrupt cpu, it must be done before the
    * registration call below. Once a ULI is registered, it is illegal
    * to modify the target cpu for the external interrupt.
    */
   if (ioctl(eifd, EIIOCSETINTRCPU, 1) < 0) {
      perror("EIIOCSETINTRCPU");
      exit(1);
   }
   /* Lock the process image into memory. Any text or data accessed
    * by the ULI handler function must be pinned into memory since
    * the ULI handler cannot sleep waiting for paging from secondary
    * storage. This must be done before the first time the ULI handler
    * is called. In the case of this program, that means before the
    * first EIIOCSTROBE is done to generate the interrupt, but in
```

```
     * general it is a good idea to do this before ULI registration
     * since with some devices an interrupt may occur at any time
     * once registration is complete
     */
    if (plock(PROCLOCK) < 0) {
       perror("plock");
       exit(1);
    }
    /* Register the external interrupt as a ULI source. */
    ULIid = ULI_register_ei( eifd,   /* the external interrupt device */
                            intrfunc,   /* the handler function pointer */
                            0,      /* the argument to the handler */
                            1,      /* the number of semaphores needed */
                            NULL,   /* the stack to use (supply one) */
                            0);     /* the stack size to use (default) */
    if (ULIid == 0) {
       perror("register ei");
       exit(1);
    }
    /* Enable the external interrupt. */
    if (ioctl(eifd, EIIOCENABLE) < 0) {
       perror("EIIOCENABLE");
       exit(1);
    }
    /* Start creating incoming interrupts. */
    signaler();
    /* Wait for the incoming interrupts and report them. Continue
     * until the program is terminated by ^C or kill.
     */
    while (1) {
    intr = 0;
    while(!intr) {
       if (ULI_sleep(ULIid, 0) < 0) {
          perror("ULI_sleep");
          exit(1);
       }
       printf("sleeper woke up\n");
    }
}
```

PART THREE

Kernel-Level Drivers

Chapter 8: Structure of a Kernel-Level Driver
The software structure of a block or character device driver: the entry points it provides for kernel use, and how it communicates with user-level processes

Chapter 9: Device Driver/Kernel Interface
A topical survey of the facilities the IRIX kernel provides to device drivers.

Chapter 10: Building and Installing a Driver
How a kernel-level driver is compiled, loaded, and linked with the IRIX kernel.

Chapter 11: Testing and Debugging a Driver
How a kernel-level driver is tested and debugged using *symmon* and other facilities.

Chapter 12: Driver Example
Annotated code of a simple memory-mapping device driver.

Chapter 8

Structure of a Kernel-Level Driver

A kernel-level device driver supplies services to the kernel through a set of entry points in the driver. When events occur, the kernel calls these entry points. The driver takes action and returns a result code.

This chapter discusses when the driver entry points are called, what parameters they receive, and what actions they are expected to take. For a conceptual overview of the kernel and drivers, see “Kernel-Level Device Control” on page 45. For details on how a driver is compiled, linked, and added to IRIX, see Chapter 10, “Building and Installing a Driver.”

Note: This chapter discusses device drivers. The entry point conventions for STREAMS drivers are covered in Chapter 19, “STREAMS Drivers.”

The primary topics covered in this chapter are:

- “Summary of Driver Structure” on page 136 summarizes the entry points and how they are made known to the kernel.
- “Driver Flag Constant” on page 140 documents a public constant the driver must supply.
- “Initialization Entry Points” on page 143 documents the entry points that are called at boot time and when a loadable driver is loaded.
- “Open and Close Entry Points” on page 146 documents the entry points called by the **open()** and **close()** kernel functions.
- “Control Entry Point” on page 150 documents the entry point called by the **ioctl()** kernel function.
- “Data Transfer Entry Points” on page 151 documents the entry points called by the **read()** and **write()** kernel functions.
- “Poll Entry Point” on page 155 documents the entry point called by the **poll()** kernel function.

- "Memory Map Entry Points" on page 158 documents the entry points called by the **mmap()** kernel function.
- "Interrupt Entry Point" on page 163 documents the entry point called to handle a device interrupt.
- "Support Entry Points" on page 167 documents the entry points that support kernel operations and system administration.
- "Planning for Multiprocessor Use" on page 171 points out methods for writing a multiprocessor-aware driver.

Summary of Driver Structure

A driver consists of a binary object module in ELF format stored in the */var/sysgen/boot* directory. As a program, the driver consists of a set of functional entry points that supply services to the IRIX kernel.

The entry points that a driver supports must be named according to a specified convention. The *lboot* command uses entry point names to build tables used by the kernel. No single driver supports all possible entry points.

Entry Point Naming and lboot

The device driver makes known which entry points it supports by giving them public names in its object module. The *lboot* command links together the object modules of drivers and other kernel modules to make a bootable kernel. *lboot* recognizes the entry points by the form of their names.

Driver Name Prefix

A device driver must be described by a file in the */var/sysgen/master.d* directory (see "Master Configuration Database" on page 38). One of the items in that configuration file specifies the driver *prefix*, a string of 1 to 14 characters that is unique to that driver. For example, the prefix of the SCSI driver is *scsi_*.

The prefix string is defined in the */var/sysgen/master.d* file only. The string does not have to appear as a constant in the driver, and the name of the driver object file does not have to correspond to the prefix (although it typically has a related name).

The *lboot* command recognizes driver entry points by searching the driver object module for public names that begin with the prefix string. For example, the entry point for the **open()** operation must have a name that consists of the prefix string followed by the letters "open."

In this book, entry point names are written as follows: *pfx***open**, where *pfx* stands for the driver's prefix string.

Kernel Switch Tables

The IRIX kernel maintains tables that allow it to dispatch calls to device drivers quickly. These tables are built by *lboot* based on the device major numbers and the names of the driver entry points. The tables are named as follows:

bdevsw	Table of block device drivers
cdevsw	Table of character device drivers
fmodsw	Table of STREAMS drivers
vfssw	Table of filesystem modules (not related to device drivers)

The tables for block and character drivers have one row for each major device number, and one column for each possible driver entry point. As *lboot* loads a driver, it fills in that driver's row of a switch table with the addresses of the driver's entry points. Where an entry point is not defined, *lboot* leaves the address of a null routine that returns the ENODEV error code.

The sizes of the switch tables are fixed at boot time in order to minimize kernel data space. The table sizes are tunable parameters that can be set with *systune* (see the systune(1) reference page).

When a driver is loaded dynamically (see "Configuring a Loadable Driver" on page 237), the associated row of the switch table is not filled at link time but rather is filled when the driver is loaded. When you add new, loadable drivers, you might need to specify a larger switch table. The *IRIX Administration: System Configuration and Operation* book documents these tunable parameters.

Entry Point Summary

The names of all possible driver entry points and their purposes are summarized in Table 8-1. STREAMS drivers are covered in Chapter 19.

Table 8-1 Entry Points in Alphabetic Order

Entry Point	Purpose	Discussion
*pfx***close**	Note the device is not in use.	page 149
*pfx***devflag**	Constant flag bits for driver features.	page 140
*pfx***edtinit**	Initialize driver from VECTOR statement.	page 144
*pfx***halt**	Prepare for system shutdown.	page 168
*pfx***init**	Initialize driver at load or boot time.	page 144
*pfx***intr**	Handle device interrupt.	page 164
*pfx***ioctl**	Implement control operations.	page 150
*pfx***map**	Implement memory-mapping (IRIX).	page 159
*pfx***mmap**	Implement memory-mapping (SVR4).	page 161
*pfx***open**	Connect a process to a device. Connect a stream module.	page 146 page 543
*pfx***poll**	Implement device event test.	page 156
*pfx***print**	Display diagnostic about block device.	page 169
*pfx***read**	Implement device input.	page 152
*pfx***rput**	STREAMS message on read queue.	page 544
*pfx***size**	Return logical size of block device.	page 169
*pfx***srv**	STREAMS service queued messages.	page 545
*pfx***start**	Initialize driver with interrupts enabled.	page 145
*pfx***strategy**	Input/output for a block device.	page 154
*pfx***unload**	Prepare loadable module for unloading.	page 167
*pfx***unmap**	Undo *pfx***mmap** memory mapping.	page 162

Table 8-1 (continued) Entry Points in Alphabetic Order

Entry Point	Purpose	Discussion
*pfx***wput**	STREAMS message on write queue.	page 544
*pfx***write**	Implement device output.	page 152

The use of entry points in different types of drivers is summarized in Table 8-2. The columns of Table 8-2 show the different types of drivers. The table cells show whether a given entry point is optional (O), required (R), or not allowed (N).

Table 8-2 Use of Driver Entry Points

Entry Point	Character	Block	Pseudo	STREAMS
*pfx***close**	R	R	R	R
*pfx***devflag**	O	O	O	O
*pfx***edtinit**	O	O	N	N
*pfx***halt**	O	O	O	O
*pfx***init**	O	O	O	O
*pfx***intr**	O	O	N	N
*pfx***ioctl**	O	O	O	N
*pfx***map**	O	O	O	N
*pfx***mmap**	O	O	O	N
*pfx***open**	R	R	R	R
*pfx***poll**	O	O	N	O
*pfx***print**	N	O	N	N
*pfx***read**	O	N	O	N
*pfx***rput**	N	N	N	R
*pfx***size**	N	R	N	N
*pfx***srv**	N	N	N	R
*pfx***start**	O	O	O	O

Table 8-2 (continued) Use of Driver Entry Points

Entry Point	Character	Block	Pseudo	STREAMS
*pfx***strategy**	N	R	N	N
*pfx***unload**	O	O	O	O
*pfx***unmap**	O	O	O	N
*pfx***wput**	N	N	N	R
*pfx***write**	O	N	O	N

As can be seen from Table 8-2, no driver supports all entry points.

- A minimal driver for a character device supports *pfx***init()**, *pfx***open()**, *pfx***read()**, *pfx***write()**, and *pfx***close()**. The *pfx***ioctl()** and *pfx***poll()** entry points are optional.
- A minimal pseudo-device driver supports *pfx***start()**, *pfx***open()**, *pfx***map()**, *pfx***unmap()**, and *pfx***close()** (the latter two as stubs).
- A minimal block device driver supports *pfx***edtinit()**, *pfx***open()**, *pfx***size()**, *pfx***strategy()**, and *pfx***close()**.

Driver Flag Constant

Any device driver or STREAMS module should define a public name *pfx***devflag** as a static integer. This integer contains a bitmask with zero or more of the following flags, which are declared in *sys/conf.h*:

D_MP	The driver is prepared for multiprocessor systems.
D_WBACK	The driver handles its own cache-writeback operations.
D_MT	The driver is prepared for a multithreaded kernel.
D_OLD	The driver implements IRIX 4.x semantics.

The flag names are declared in the header file *sys/ddi.h*. A typical definition would resemble the following:

```
int testdrive_devflag = D_MP;
```

A STREAMS module should also provide this flag, but the only relevant bit value for a STREAMS driver is D_MP (see “Driver Flag Constant” on page 542).

The flag value is saved in the kernel switch table with the driver’s entry points (see “Kernel Switch Tables” on page 137).

When a driver does not define a *pfx***devflag**, *lboot* saves a word containing D_OLD by default. See the note regarding D_OLD on page 142.

Flag D_MP

You specify D_MP in *pfx***devflag** to tell *lboot* that your driver is designed to operate in a multiprocessor system. The top half of the driver is designed to cope with multiple concurrent entries in multiple CPUs. The top and bottom halves synchronize through the use of semaphores or locks and do not rely on interrupt masking for critical sections. These issues are discussed further under “Planning for Multiprocessor Use” on page 171.

When D_MP is not present in *pfx***devflag**, IRIX ensures that the driver code, including the upper-half entry points and the interrupt handler, executes only on CPU 0 of a multiprocessor. This ensures behavior similar to a uniprocessor, but can cause a performance bottleneck when either the device or CPU 0 is heavily used.

Note: This flag is only tested when loading a character driver or STREAMS driver. There is no special handling for block drivers and network drivers in multiprocessors. Block and network drivers must be multiprocessor-aware.

Flag D_WBACK

You specify D_WBACK in *pfx***devflag** to tell *lboot* that a block driver performs any necessary cache write-back operations through explicit calls to **dki_dcache_wb()** and related functions (see the dki_dcache_wb(D3) reference page).

When D_WBACK is not present in *pfx***devflag**, the **physiock()** function ensures that all cached data related to *buf_t* structures is written back to main memory before it enters the driver’s strategy routine. (See the physiock(D3) reference page and “Entry Point strategy()” on page 154.)

Flag D_MT

This flag is defined in IRIX 6.2 but has no effect in that release. Future releases of IRIX will run driver interrupt routines as lightweight threads of control within the kernel address space. D_MT indicates that this driver understands that it can be run as one or more cooperating threads, and uses kernel synchronization primitives to serialize access to driver common data structures.

Flag D_OLD

The D_OLD flag exists only to retain compatibility with certain drivers written originally for IRIX 4.x. It changes two features of the kernel-to-driver interface:

- The first argument to the *pfx***open()** entry is a *dev_t* value instead of the pointer-to-*dev_t* that is now standard.
- The driver sets its return code by storing it into a global, *u.u_error*, instead of returning it as the result of the function call.

D_OLD is incompatible with D_MP.

When a driver has no *pfx***devflag** constant, *lboot* assumes it is a D_OLD driver.

Note: The D_OLD flag value, and the incompatible interface that it implies, is supported for compatiblity only. Support for this flag, and support for drivers that use it (or that have no *pfx***devflag** constant), will be withdrawn in the release of IRIX after 6.2. It may be removed from certain platform-specific releases of IRIX 6.2. Silicon Graphics urges you to revise any driver that depends on D_OLD to use current semantics for parameters and return codes.

Initialization Entry Points

The kernel calls a driver to initialize itself at any of three different entry points, as follows:

*pfx***init**	Initialize self-defining hardware or a pseudo-device.
*pfx***edtinit**	Initialize a hardware device based on VECTOR data.
*pfx***start**	General initialization.

Each call has different abilities. A driver may define any combination of the three entry points. It is not uncommon to define both a *pfx***start()** and one of *pfx***edtinit()** or *pfx***init()**.

When Initialization Is Performed

The initialization entry points of ordinary (nonloadable) drivers are called during system startup, after interrupts have been enabled and before the message "The system is coming up" is displayed. In all cases, interrupts are enabled and basic kernel services are available at this time. However, other loadable or optional kernel modules might not have been initialized, depending on the sequence of statements in the files in */var/sysgen/system*.

Whenever a driver is initialized, the entry points are called in the following sequence:

1. *pfx***init()** is called first.
2. *pfx***edtinit()** is called once for each VECTOR statement in reverse order of the VECTOR statements.
3. *pfx***start()** is called last.

Initialization of Loadable Drivers

When a loadable driver (see "Loadable Drivers" on page 57) is configured for autoregister, it is loaded with other drivers during system startup. (For more information on autoregister, see "Configuring a Loadable Driver" on page 237.) Such a driver is initialized at system startup time along with the nonloadable drivers.

When a loadable driver is dynamically loaded at some time after system startup, its initialization entry points are called in sequence at the time it is loaded.

Entry Point init()

The *pfx***init()** entry point is called once during system startup or when a loadable driver is loaded. It receives no input arguments; its prototype is simply:

```
void pfxinit(void);
```

You can use this entry point to initialize a hardware device that is self-defining; that is, all the information the driver needs is either coded into the driver, or can be gotten by probing the device itself. You can also use *pfx***init()** to initialize a pseudo-device driver; that is, a driver that does not have real hardware attached.

A driver that is brought into the system by a USE or INCLUDE line in a system configuration file (see "Configuring a Kernel" on page 236) typically initializes in the *pfx***init()** entry point.

Entry Point edtinit()

The *pfx***edtinit()** entry is designed to initialize devices that are configured using the VECTOR statement in the system configuration file (see "System Configuration Files" on page 39). The entry point name is a contraction of "*e*arly *d*evice *t*able *init*ialization."

The VECTOR statement specifies hardware details about a device on the VME, GIO, or EISA bus, including iospace addresses, interrupt level, and an integer parameter. The VECTOR statement can specify a "probe" parameter that lets the kernel test for the existence of the specified hardware.

When the kernel processes a VECTOR statement during bootstrap and the probe is successful (or no probe is specified), the kernel stores the VECTOR parameters in a structure of type *edt_t*. (This structure is declared in *sys/edt.h*.)

Each time the kernel loads a driver that is named in a VECTOR statement, the kernel calls the driver's *pfx***edtinit()** entry one time for each VECTOR statement that named that driver and had a successful probe (or that had no probe). VECTOR statements are processed in reverse sequence to the order in which they are coded in */var/sysgen/system* files.

The prototype of the *pfx***edtinit()** entry is

```
void pfxedtinit(edt_t *e);
```

The *edt_t* contains at least the following fields (see the system(4) reference page for the corresponding VECTOR parameters):

e_bus_type	Integer specifying the bus type; constant values are declared in *sys/edt.h*, for example ADAP_VME, ADAP_GIO, or ADAP_EISA.
e_adap	Integer specifying the adapter (bus) number.
e_ctlr	Value from the VECTOR *ctlr*= parameter; typically the device minor number.
e_space	Array of up to three I/O space structures of type *iospace_t*.

The difference between *pfx***init()** and *pfx***edtinit()** is that *pfx***edtinit()** is parameterized with information from the VECTOR line, and is called once for each VECTOR line that is associated with real hardware.

A driver that uses *pfx***edtinit()** needs to save the *edt_t* information in a data structure. If the driver supports multiple devices—that is, if it can be called for multiple VECTOR statements—it needs to allocate an array or chain of structures, and save new data on each entry.

Entry Point start()

The *pfx***start()** entry point is called at system startup, and whenever a loadable driver is loaded. It is called after *pfx***edtinit()** and *pfx***init()**, but before any other entry point such as *pfx***open()**. The *pfx***start()** entry point receives no arguments; its prototype is simply

```
void pfxstart(void);
```

The *pfx***start()** entry point is a suitable place to allocate a poll-head structure using **phalloc()**, as discussed in "Use and Operation of poll(2)" on page 155.

Open and Close Entry Points

The *pfx***open()** and *pfx***close()** entries for block and character devices are called when a device comes into use and when use of it is finished. For a conceptual overview of the **open()** process, see "Overview of Device Open" on page 46.

Entry Point open()

The kernel calls a device driver's *pfx***open()** entry when a process executes the **open()** system call on any device special file (see the open(2) reference page). It is also called when a process executes the **mount()** system call on a block device (see the mount(2) reference page). (For the *pfx***open()** entry point of a STREAMS driver, see "Entry Point open()" on page 543.)

The prototype of *pfx***open()** is as follows:

```
int pfxopen(dev_t *devp, int oflag, int otyp, cred_t *crp);
```

The argument values are

**devp*	Pointer to a *dev_t* value from which you can extract both the major and minor device numbers.
otyp	An integer flag specifying the source of the call: a user process opening a character device or block device, or another driver.
oflag	Flag bits specifying user mode options on the **open()** call.
crp	A *cred_t* object—an opaque structure for use in authentication. Standard access privileges to the special device file have already been verified.

Note: When the driver's *pfx***devflag** entry contains D_OLD or when *pfx***devflag** is not defined, the first argument to *pfx***open()** is a *dev_t* value, not a pointer to a *dev_t* value. See "Flag D_OLD" on page 142.

In general, the driver is expected to verify that this user process is permitted access in the way specified in *otyp* (reading, writing, or both) for the device specified in **devp*. If access is not allowable, the driver returns a nonzero error code from *sys/errno.h*, for example ENOMEM or EBUSY.

Use of the Device Number

When the driver supports a single device with no logical unit divisions, the device number is of little interest except for diagnostic displays. When the driver supports multiple devices, or a device with multiple logical units, the minor device number is the key to locating the device information. The device number can also encode device options, as discussed under "Minor Device Number" on page 35.

When the driver supports the *pfx***edtinit()** entry, the driver needs a way to associate the different *edt_t* structures passed to *pfx***edtinit()** with the device numbers passed to *pfx***open()** and other routines. One solution is to require that the *ctlr*= value from the VECTOR statement—which is passed in the *e_ctlr* field of *edt_t*—must be the same as the device minor number.

Use of the Open Type

The *otyp* flag distinguishes between the following possible sources of this call to *pfx***open()** (the constants are defined in *sys/open.h*).

- a call to open a character device (OTYP_CHR)
- a call to open a block device (OTYP_BLK)
- a call to a mount a block device as a filesystem (OTYP_MNT)
- a call to open a block device as swapping device (OTYP_SWP)
- a call direct from a device driver at a higher level (OTYP_LYR)

Typically a driver is written only to be a character driver or a block driver, and can be called only through the switch table for that type of device. When this is the case, the *otyp* value has little use.

It is possible to have the same driver treated as both block and character, in which case the driver needs to know whether the **open()** call addressed a block or character special device. It is possible for a block device to support different partitions with different uses, in which case the driver might need to record the fact that a device has been mounted, or opened as a swap device.

With all open types except OTYP_LYR, *pfx***open()** is called for every open or mount operation, but *pfx***close()** is called only when the last close or unmount occurs. The OTYP_LYR feature is used almost exclusively by drivers distributed with IRIX, like the host adapter SCSI driver (see "Host Adapter Concepts" on page 355). For each open of this type, there is one call to *pfx***close()**.

Use of the Open Flag

The interpretation of the open mode flags is up to the designer of the driver. Four modes can be requested (declared in *sys/file.h*):

FREAD	Input access wanted.
FWRITE	Output access wanted (both FREAD and FWRITE may be set, corresponding to O_RDWR mode).
FNDELAY or FNONBLOCK	Return at once, do not sleep if the open cannot be done immediately.
FEXCL	Request exclusive use of the device.

You decide which of the flags have meaning with respect to the abilities of this device. You can return an EINVAL error when an unsupported mode is requested.

A key decision is whether the device can be opened only by one process at a time, or by multiple processes. If multiple opens are supported, a process can still request exclusive access with the FEXCL mode.

When the device can be used by only one process, or when FEXCL access is supported, the driver must keep track of the fact that the device is open. When the device is busy, the driver can test the FNDELAY and FNONBLOCK flags; if either is set, it can return EBUSY. Otherwise, the driver should sleep until the device is free; this requires coordination with the *pfx***close()** entry point.

Use of the cred_t Object

The *cred_t* object passed to *pfx***open()**, *pfx***close()**, and *pfx***ioctl()** can be used with the **drv_priv()** function to find out if the effective calling user ID is privileged or not (see the drv_priv(D3) reference page). Do not examine the object in detail, since its contents are subject to change from release to release.

Saving the Size of a Block Device

In a block device driver, the *pfx***size()** entry point will be called soon after *pfx***open()** (see "Entry Point size()" on page 169). It is typically best to calculate or read the device capacity at open time, and save it to be reported from *pfx***size()**.

Saving the User ABI

If your driver is, or might be, compiled to the 64-bit model for use with a 64-bit IRIX kernel, and if it supports the *pfx***ioctl()** or *pfx***poll()** entry points, the driver should test and save the user process's programming model during an open. For details, see "Handling 32-Bit and 64-Bit Execution Models" on page 170.

Entry Point close()

The kernel calls the *pfx***close()** entry when the last process calls **close()** or **umount()** for the device special file. It is important to know that when the device can be opened by multiple processes, *pfx***close()** is not called for every **close()** function, but only when the last remaining process closes the device and no other processes have it open.

The function prototype and arguments of *pfx***close()** are

```
int pfxclose(dev_t dev, int flag, int otyp, cred_t *crp);
```

The arguments are the same as were passed to *pfx***open()**. However, the flag argument is not necessarily the same as at any particular call to **open()**.

It is up to you to design the meaning of "close" for this type of device:

- If the device is opened and closed frequently, you may decide to retain dynamic data structures.
- If the device can perform an action such as "rewind" or "eject," you decide whether that action should be done upon close. Possibly the choice of acting or not acting can be set by an **ioctl()** call; or possibly the choice can be encoded into the device minor number—for example, the no-rewind-on-close option is encoded in certain tape minor device numbers.

- If the *pfx***open()** entry point supports exclusive access, and it can be waiting for the device to be free, *pfx***close()** must release the wait.

The *pfx***close()** entry can detect an error and report it with a return code. However, the file is closed or unmounted regardless.

Control Entry Point

The *pfx***ioctl()** entry point is called by the kernel when a user process executes the **ioctl()** system call (see the ioctl(2) reference page). This entry point is allowed in character drivers only. Block device drivers do not support it, and STREAMS drivers pass control information as messages.

For an overview of the relationship between the user process, kernel, and the control entry point, see "Overview of Device Control" on page 48.

The prototype of the entry point is

```
int pfxioctl(dev_t dev, int cmd, void *arg,
            int mode, cred_t *crp, int *rvalp);
```

The argument values are

dev	A *dev_t* value from which you can extract the major and minor device numbers.
cmd	The request value specified in the **ioctl()** call.
arg	The optional argument value specified in the **ioctl()** call, or NULL if none was specified.
mode	Flag bits specifying the **open()** mode, as associated with the file descriptor passed to the **ioctl()** system function.
crp	A *cred_t* object—an opaque structure for use in authentication, describing the process that is in-context. Standard access privileges to the special device file have already been verified.
**rvalp*	The integer result to be returned to the user process.

It is up to the device driver to interpret the *cmd* and *arg* values in the light of the *mode* and other arguments. When the *arg* value is a pointer to data in the process address space, the driver uses the **copyin()** kernel function to copy the data into kernel space, and the **copyout()** function to return updated values. (See the copyin(D3) and copyout(D3) reference pages, and also "Transferring Data" on page 192.)

Choosing the Command Numbers

The command numbers supported by *pfx***ioctl()** are arbitrary; but the recommended practice is to make sure that they are different from those of any other driver. One method to achieve this is suggested in the ioctl(D2) reference page.

Supporting 32-Bit and 64-Bit Callers

The **ioctl()** entry point may need to interpret a structure prepared in the user process. In a 64-bit system, the user process can be either a 32-bit or a 64-bit program. For discussion of this issue, see "Handling 32-Bit and 64-Bit Execution Models" on page 170

User Return Value

The kernel returns 0 to the **ioctl()** system function unless the *pfx***ioctl()** function returns an error code. In the event of an error, the kernel returns the code the driver places in **rvalp*, if any, or -1. To ensure that the user process sees a specific error code, set the code in **rvalp*, and return that value.

Data Transfer Entry Points

The *pfx***read()** and *pfx***write()** entry points are supported by character device drivers and pseudo-device drivers that allow reading and writing. They are called by the kernel when the user process calls the **read()**, **readv()**, **write()**, or **writev()** system function.

The *pfx***strategy()** entry point is required of block device drivers. It is called by the kernel when either a filesystem or the paging subsystem needs to transfer a block of data.

Entry Points read() and write()

The *pfx***read()** and *pfx***write()** entry points are similar to each other—only the direction of data transfer differs. The prototypes of the functions are

```
int pfxread (dev_t dev, uio_t *uiop, cred_t *crp);
int pfxwrite(dev_t dev, uio_t *uiop, cred_t *crp);
```

The arguments are

dev	A *dev_t* value from which you can extract both the major and minor device numbers.
**uiop*	A *uio_t* object—a structure that defines the user's buffer memory areas.
crp	A *cred_t* object—an opaque structure for use in authentication. Standard access privileges to the special device file have already been verified.

Data Transfer for a PIO Device

A character device driver using PIO transfers data in the following steps:

1. If there is a possibility of a timeout, start a timeout delay (see "Waiting for Time to Pass" on page 214).
2. Initiate the device operation as required.
3. Transfer data between the device and the buffer represented by the *uio_t* (see "Transferring Data Through a uio_t Object" on page 194).
4. If it is necessary to wait for an interrupt, put the process to sleep (see "Waiting and Mutual Exclusion" on page 204).
5. When data transfer is complete, or when an error occurs, clear any pending timeout and return the final status of the operation. If the return code is 0, the final state of the *uio_t* determines the byte count returned by the **read()** or **write()** call.

Calling Entry Point strategy() From Entry Point read() or write()

A device driver that supports both character and block interfaces must have a *pfx***strategy()** routine in which it performs the actual I/O. For example, the Silicon Graphics disk drivers support both character and block driver interfaces, and perform all I/O operations in the *pfx***strategy()** function. However, the *pfx***read()**, *pfx***write()** and *pfx***ioctl()** entries supported for character-type access also need to perform I/O operations. They do this by calling the *pfx***strategy()** routine indirectly, using the kernel function **physiock()** or **uiophysio()** (see the physiock(D3) and uiophysio(D3) reference pages, and see “Waiting for Block I/O to Complete” on page 217).

Both the **physiock()** and **uiophysio()** functions takes care of the housekeeping needed to interface to the *pfx***strategy()** entry, including the work of allocating a buffer and a *buf_t* structure, locking buffer pages in memory and waiting for I/O completion. The **physiock()** function is preferable because it is compatible with SVR4. Example 8-1 shows the skeleton of a hypothetical driver in which the *pfx***read()** entry does its work through the *pfx***strategy()** entry.

Example 8-1 Hypothetical pfxread() entry in a Character/Block Driver

```
hypo_read (dev_t dev, uio_t *uiop, cred_t *crp)
{
   // ...validate the operation... //
   return physiock(hypo_strategy, /* our strategy entry */
                  0,   /* allocate temp buffer & buf_t */
                  dev, /* dev_t arg for strategy */
                  B_READ, /* direction flag for buf_t */
                  uiop);
}
```

The *pfx***write()** entry would be identical except for passing B_WRITE instead of B_READ.

This dual-entry strategy is required only in a driver that supports both character and block access. **physiock()** returns an error if the length requested in the *uio_t* object is not a multiple of the standard sector length, so a different approach is required for a device that permits odd-length I/O.

Entry Point strategy()

A block device driver does not directly support user process calls. Instead, it provides services to a filesystem such as EFS, or to the memory paging subsystem of IRIX. These subsystems call the *pfx***strategy()** entry point to read data in whole blocks.

Calls to *pfx***strategy()** are not directly related in time to system functions called by a user process. For example, a filesystem may buffer many blocks of data in memory, so that the user process may execute dozens or hundreds of **write()** calls without causing an entry to the device driver. When the user function closes the file or calls **fsync()**—or when for unrelated reasons the filesystem needs to free some buffers—the filesystem calls *pfx***strategy()** to write numerous blocks of data.

In a driver that supports the character interface as well, the *pfx***strategy()** entry can be called indirectly from the *pfx***read()**, *pfx***write()** and *pfx***ioctl()** entries, as described under "Calling Entry Point strategy() From Entry Point read() or write()" on page 153.

The prototype of the *pfx***strategy()** entry point is

```
int pfxstrategy(struct buf *bp);
```

The argument is the address of a *buf_t* structure, which gives the strategy routine the information it needs to perform the I/O:

- The major and minor device numbers
- The direction of the transfer (read or write)
- The location of the buffer in kernel memory
- The amount of data to transfer
- The starting block number on the device

For more on the contents of the *buf_t* structure, see "Structure buf_t" on page 183 and the buf(D4) reference page.

The driver uses the information in the *buf_t* to validate the data transfer and programs the device to start the transfer. Then it stores the address of the *buf_t* where the interrupt handler can find it (see "Entry Point intr()" on page 164) and calls **biowait()** to wait for the operation to complete. For the next step, see "Completing Block I/O" on page 166 (see also the biowait(D3) reference page).

Poll Entry Point

The *pfx***poll()** entry point is called by the kernel when a user process calls the **poll()** or **select()** system function asking for status on a character special device. To implement it, you need to understand the IRIX implementation of **poll()**.

Use and Operation of poll(2)

The IRIX version of **poll()** allows a process to wait for events of different types to occur on any combination of devices, files, and STREAMS (see the poll(2) and select(2) reference pages). It is possible for multiple processes to be waiting for events on the same device.

It is up to you as the designer of a driver to decide which of the events that are documented in poll(2) are meaningful for your device. Other requested events simply never happen to the device.

Much of the complexity of **poll()** is handled by the IRIX kernel, but the kernel requires the assistance of any device driver that supports **poll()**. The driver is expected to allocate and hold a *pollhead* structure (declared in *sys/poll.h*) for each minor device that it supports. Allocation is simple; the driver merely calls the **phalloc()** kernel function. (The *pfx***start()** entry point is a suitable place for this call; see "Entry Point start()" on page 145.)

There are two phases to the operation of **poll()**. When the system function is called, the kernel calls the *pfx***poll()** entry point to find out if any requested events are pending at this time. If the kernel finds any event s pending (on this or any other polled object), the **poll()** function returns to the user process. Nothing further is required.

However, when no requested event has happened, the user process expects the **poll()** function to block until an event has occured. The kernel cannot implement this delay by repeatedly testing for events; that would be too inefficient. The kernel must rely on device drivers to notify it when an event has occurred.

Use of pollwakeup()

A device driver that supports *pfx***poll()** is required to notify the kernel whenever an event that the driver supports has occurred. The driver does this by calling a kernel function, **pollwakeup()**, passing the *pollhead* structure for the affected device, and bit flags for the events that have taken place. In the event that one or more user processes are blocked in a **poll()**, waiting for an event from this device, the call to **pollwakeup()** will release the sleeping processes. For an example, see "Calling pollwakeup()" on page 166.

Use of pollwakeup() Without Interrupts

If the device in question does not support interrupts, the driver cannot support **poll()** unless it can somehow get control to discover an event and report it to **pollwakeup()**. One possibility is that the driver could simulate interrupts by setting a succession of **itimeout()** delays. On each timeout the driver would test its device for a change of status, call **pollwakeup()** when an event has occurred; and schedule a new delay. (See "Waiting for Time to Pass" on page 214.)

Entry Point poll()

The prototype for *pfx***poll()** is as follows:

```
int pfxpoll(dev_t dev, short events, int anyyet,
            short *reventsp, struct pollhead **phpp);
```

The argument values are

dev	A *dev_t* value from which you can extract the major and minor device numbers.
events	Bit-flags for the events the user process is testing, as passed to **poll()** and declared in *sys/poll.h*..
reventsp	A field to receive the bit-flags of events that have occurred, or to receive 0x0000 if no requested events have occurred..
anyyet and *phpp*	When *anyyet* is nonzero and no events have occurred, the kernel requires the address of the pollhead structure for this minor device to be returned in **phpp*.

Example 8-2 shows the *pfx***poll()** code of a hypothetical device driver. Only three event tests are supported: POLLIN and POLLRDNORM (treated as equivalent) and POLLOUT. The device driver maintains an array of *pollhead* structures, one for each supported minor device. These are presumably allocated during initialization.

Example 8-2 *pfx*poll() Code for Hypothetical Driver

```
struct pollhead phds[MAXMINORS];
#define OUR_EVENTS (POLLIN|POLLOUT|POLLRDNORM)
hypo_poll(dev_t dev, short events, int anyyet,
          short *reventsp, struct pollhead **phpp)
{
   minor_t dminor = geteminor(dev);
   short happened = 0;
   short wanted = events & OUR_EVENTS;
   if (wanted & (POLLIN|POLLRDNORM))
   {
      if (device_has_data_ready(dminor))
         happened |= (POLLIN|POLLRDNORM);
   }
   if (wanted & POLLOUT)
   {
      if (device_ready_for_output(dminor))
         happened |= POLLOUT;
   }
   if (device_pending_error(dminor))
      happened |= POLLERR;
   if (0 == (*reventsp = happened))
   {
      if (anyyet) *phpp = phds[dminor]
   }
   return 0;
}
```

The code in Example 8-2 begins by discarding any unsupported event flags that might have been requested. Then it tests the remaining flags against the device status. If the device has an uncleared error, the code inserts the POLLERR event. If no events were detected, and if the kernel requested it, the address of the *pollhead* structure for this minor device is returned.

Memory Map Entry Points

A user process requests memory mapping by calling the system function **mmap()**. When the mapped object is a character device special file, the kernel calls the *pfx***mmap()** or *pfx***map()** entry to validate and complete the mapping. To understand these entry points, you must understand the **mmap()** system function.

Concepts and Use of mmap()

The purpose of the **mmap()** system function (see the mmap(2) reference page) is to make the contents of a file directly accessible as part of the virtual address space of the user process. The results depend on the kind of file that is mapped:

- When the mapped object is a normal file, the process can load and store data from the file as if it were an array in memory.
- When the mapped object is a character device special file, the process can load and store data from device registers as if they were memory variables.
- When the mapped object is a block of memory owned and prepared by a pseudo-device driver, the process gains access to some special piece of memory data that it would not normally be able to access.

In all cases, access is gained through normal load and store instructions, without the overhead of calling system functions such as **read()**. Furthermore, the same mapping can be executed by other processes, in which case the same memory, or file, or device is shared by multiple, concurrent processes. This is how shared memory segments are achieved.

Use of mmap()

The **mmap()** system function takes four key parameters:

- the file descriptor for an open file, which can be either a normal disk file or a device special file
- an offset within that file at which the mapped data is to start. For a normal file, this is a file offset; for a device file, it represents an address in the address space of the device or the bus
- the length of data to be mapped
- protection flags, showing whether the mapped data is read-only or read-write

When the mapped object is a normal file, the filesystem implements the mapping. The filesystem does not call the block device driver for assistance in mapping a file. It does call the block device driver *pfx***strategy()** entry to read and write blocks of file data as necessary, but the mapping of pages of data into pages of memory is controlled in the filesystem code.

When the mapped object is a device special file, the **mmap()** parameters are passed to the device driver at either its *pfx***mmap()** or *pfx***map()** entry point. The device driver interprets the parameters in the context of the device, and uses a kernel function to create the mapping.

Persistent Mappings

Once a device or kernel memory has been mapped into some user address space, the mapping persists until the user process terminates or calls **unmap()** (see the unmap(2) reference page). In particular, the mapping does not end simply because the device special file is closed. You cannot assume, in the *pfx***close()** or *pfx***unload()** entry points, that all mappings to devices have ended.

Entry Point map()

The *pfx***map()** entry point can be defined in either a character or a block driver (it is the only mapping entry point that a block driver can supply). The function prototype is

```
int pfxmap(dev_t dev, vhandl_t *vt,
          off_t off, int len, int prot);
```

The argument values are

dev	A *dev_t* value from which you can extract both the major and minor device numbers.
vt	The address of an opaque structure that describes the assigned address in the user process address space. The structure contents are subject to change.
off, *len*	The offset and length arguments passed to **mmap()** by the user process.
prot	Flags showing the access intentions of the user process.

The first task of the driver is to verify that the access specified in *prot* is allowed. The next task is to validate the *off* and *len* values: do they fall in the valid address space of the device?

When the device driver approves of a mapping, it uses a kernel function, **v_mapphys()**, to establish the mapping. This function (documented in the v_mapphys(D3) reference page) takes the *vhandle_t*, an address in kernel cached or uncached memory, and a length. It makes the specified region of kernel space a part of the address space of the user process.

For example, a pseudo-device driver that intends to share kernel virtual memory with user processes would first allocate the memory:

```
caddr_t *kaddr = kmem_alloc (len , KM_CACHEALIGN);
```

It would then use the address of the allocated memory with the *vhandle_t* value it had received to map the allocated memory into the user space:

```
v_mapphys (vt, kaddr, len)
```

Note: There are no special precautions to take when mapping cached memory into user space, or when mapping device registers or bus addresses. However, you should almost never map *uncached memory* into user space. The effects of uncached memory access are hardware dependent and differ between multiprocessors and uniprocessors. Among uniprocessors, the IP26 CPU module has highly restrictive rules for the use of uncached memory (see "Uncached Memory Access in the IP26 CPU" on page 27). In general, mapping uncached memory makes a driver nonportable and is likely to lead to subtle failures that are hard to resolve.

Example 8-3 contains an edited fragment of code from a Silicon Graphics device driver. This pseudo-device driver, whose prefix is *flash_*, provides access to "flash" PROM in certain computer models. It allows a user process to map the PROM into user space.

Example 8-3 Edited Fragment of flash_map()

```
int flash_map(dev_t dev, vhandl_t *vt, off_t off, long len)
{
   long offset = (long) off; /*Actual offset in flash prom*/
/* Don’t allow requests which exceed the flash prom size */
   if ((offset + len) > FLASHPROM_SIZE)
      return ENOSPC;
/* Don’t allow non page-aligned offsets */
   if ((offset % NBPC) != 0)
      return EIO;
/* Only allow mapping of entire pages */
   if ((len % NBPC) != 0)
      return EIO;
   return v_mapphys(vt, FLASHMAP_ADDR + offset, len);
}
```

When the driver allocates some memory resource associated with the mapping, and when more than one mapping can be active at a time, the driver needs to tag each memory resource so it can be located when the *pfx***unmap()** entry point is called. One answer is to use the **vt_gethandle()** macro defined in *sys/region.h*. This macro takes a pointer to a *vhandle_t* and returns a unique pointer-sized integer that can be used to tag allocations. No other information in *sys/region.h* is supported for driver use.

Entry Point mmap()

The *pfx***mmap()** (note: *two* letters “m”) entry can be used only in a character device driver. The prototype is

```
int pfxmmap(dev_t dev, off_t off, int prot);
```

The argument values are

dev	A *dev_t* value from which you can extract both the major and minor device numbers.
off	The offset argument passed to **mmap()** by the user process.
prot	Flags showing the access intentions of the user process.

The function is expected to return the page frame number (PFN) that corresponds to the offset *off* in the device address space. A PFN is an address divided by the page size. (See "Working With Page and Sector Units" on page 197 for page unit conversion functions.)

This entry point is supported only for compatibility with SVR4. When the kernel needs to map a character device, it looks first for *pfx***map()**. It calls *pfx***mmap()** only when *pfx***map()** is not available. The differences between the two entry points are as follows:

- This entry point receives no *vhandl_t* argument, so it cannot use **v_mapphys()**. It has to calculate a page frame number, which means that it has to be aware of the current page size (obtainable from the **ptob()** kernel function, see the ptob(D3) reference page).
- This entry point does not receive a length argument, so it has to assume a default length for every map (typically the page size).
- When a mapping is created using this entry point, the *pfx***unmap()** entry is not called.

Entry Point unmap()

The kernel calls the *pfx***unmap()** entry point when a mapping is created using the *pfx***map()** entry point. This entry should be supplied, even if it is an empty function, when the *pfx***map()** entry point is supplied. If it is not supplied, the **munmap()** system function returns the ENODEV error.

The *pfx***unmap()** entry point is only called when the mapped region has been completely unmapped by all processes. For example, suppose a parent process calls **mmap()** to map a device. Then the parent creates one or more child processes using **sproc()**. Each child shares the address space, including the mapped segment. A process in the share group can terminate, or can explicitly **unmap()** the segment or part of the segment; these actions do not result in a call to *pfx***unmap()**. Only when the last process with access to the segment has fully unmapped the segment is *pfx***unmap()** called.

On entry, the kernel has completed unmapping the object from the user process address space. This entry point does not need to do anything to affect the user address space; it only needs to release any resources that were allocated to support the mapping.

The prototype is

```
int pfxunmap(dev_t dev, vhandl_t *vt);
```

The argument values are

dev	A *dev_t* value from which you can extract both the major and minor device numbers.
vt	The address of an opaque structure that describes the assigned address in the user process address space.

If the driver allocated no resources to support a mapping, no action is needed here; the entry point can consist of a "return 0" statement.

When the driver does allocate memory to support a mapping, and supports multiple mappings, the driver needs to identify the resource associated with this particular mapping in order to release it. The **vt_gethandle()** function returns a unique number based on the *vt* argument; this can be used to identify resources.

Interrupt Entry Point

When a hardware device presents an interrupt, the IRIX kernel locates a device driver for the device and calls its *pfx***intr()** entry point. The driver processes the interrupt as quickly as possible.

In principle an interrupt can happen at any time. Normally an interrupt occurs because at some previous time, the driver initiated a device operation. Some devices can interrupt without a preceding command.

Associating Interrupt to Driver

The association between an interrupt and the driver is established in different ways depending on the hardware.

- The VECTOR statement establishes the interrupt level and the associated driver for devices on the EISA and VME busses.
- For some VME devices, the interrupt level is established dynamically using **vme_ivec_set()** (see Chapter 14, "Services for VME Drivers").

- The interrupt vector for a device on the GIO bus is set dynamically, by calling **setgiovector()** from the *pfx***init()** entry point (see Chapter 18, “GIO Device Drivers”).
- For devices on the SCSI bus, all interrupts are handled by a single, low-level driver which notifies a callback function (see Chapter 15, “SCSI Device Drivers”).

Entry Point intr()

The prototype of the *pfx***intr()** entry point is

```
void pfxintr(int ivec);
```

The *ivec* argument is a number that represents the interrupt, depending on the type of device, type of bus, and the way the interrupt was associated to the driver.

When an interrupt occurs, the system is in an unknown state. As a result, the interrupt handler can use only a restricted set of kernel services, and no services that can sleep. In general, the interrupt handler implements the following tasks.

- When the driver supports multiple logical units, use *ivec* to locate the data structure for the interrupting unit.
- Determine the reason for the interrupt by interrogating the device.
- When the interrupt is a response to a device operation, note the success or failure of the command.
- If the driver top half is waiting for the interrupt, waken it.
- If the driver supports polling, and the interrupt represents a pollable event, call **pollwakeup()**.
- If the device is not in an error state and another operation is waiting to be started, start it.

The details of each of these tasks depends on the hardware and on the design of the data structures used by the driver top half.

Mutual Exclusion

In a uniprocessor system, there is only one CPU and when it is executing the interrupt handler, nothing else is executing. An interrupt handler can only be preempted by an interrupt of higher priority—which would be an interrupt for a different driver, and so would have no conflicts with this driver over the use of data.

In a multiprocessor, an interrupt can be taken on any CPU, while other CPUs continue to execute kernel or user code. (In the Challenge and Onyx systems, interrupts are "sprayed" to CPUs in rotation to equalize the workload. Also, VME interrupts can be directed to specific CPUs for handling; see "Using the IPL Statement" on page 312.)

In a multiprocessor, when an interrupt must be handled by a driver that is not marked as multiprocessor-aware (see "Flag D_MP" on page 141), the interrupt may be received on some other CPU, but the driver interrupt entry point is always executed on CPU 0.

In a multiprocessor, when the driver is multiprocessor-aware, one or more other CPUs can execute in the driver's top-half entry points while another CPU executes the driver's interrupt entry point. An interrupt handler written for a multiprocessor must not assume that it has exclusive use of the driver's data (see "Planning for Multiprocessor Use" on page 171).

It is theoretically possible in a multiprocessor for a device to interrupt; for one CPU to enter the interrupt handler; and for the device to interrupt again, resulting in multiple concurrent entries to the same interrupt handler. However, IRIX prevents this. You can assume that your interrupt handler code is entered serially, and not used concurrently by multiple CPUs.

Performance and Latency

Speed in exiting the interrupt handler is critical to system performance. In a uniprocessor, the system is doing nothing else while it executes the handler, and it cannot respond to interrupts of a lower priority. In a multiprocessor, interrupts can be taken by different CPUs. While a CPU executes a handler, that CPU cannot respond to lower-priority interrupts, but other CPUs can be processing user-level code or responding to other interrupts.

Completing Block I/O

In a block device driver, an I/O operation is represented by the *buf_t* structure. The *pfx***strategy()** routine starts operations and waits for them to complete (see “Entry Point strategy()” on page 154).

The interrupt entry point sets the residual count in *b_resid*. It can post an error using **bioerror()**. It posts the operation complete and wakens the *pfx***strategy()** routine by calling **biodone()**. If the *pfx***strategy()** entry has set the address of a completion callback function in the *b_iodone* field of the *buf_t*, **biodone()** invokes it. (For more discussion, see “Waiting for Block I/O to Complete” on page 217.)

Completing Character I/O

In a character device driver, the driver top half typically awaits an interrupt by sleeping on a semaphore or synchronizing variable, and the interrupt routine posts the semaphore (see “Waiting for a General Event” on page 218). Error information must be passed in driver variables according to some local convention.

Calling pollwakeup()

When the interrupt represents an event that can be reported by the driver’s *pfx***poll()** entry point (see “Entry Point poll()” on page 156), the interrupt handler must report the event to the kernel, in case some user process is waiting in a **poll()** call. Hypothetical code to do this is shown in Example 8-4.

Example 8-4 Hypothetical Call to pollwakeup()

```
hypo_intr(int ivec)
{
   struct hypo_dev_info *pinfo;
   if (! pinfo = find_dev_info(ivec))
      return; /* not our device */
   ...
   if (pinfo->have_data_flag)
      pollwakeup (pinfo->phead, POLLIN, POLLRDNORM);
   if (pinfo->output_ok_flag)
      pollwakeup (pinfo->phead, POLLOUT);
   ...
}
```

Support Entry Points

Certain driver entry points are used to support the operations of the kernel or the administration of the system.

Entry Point unload()

The *pfx***unload()** entry point is called when the kernel is about to dynamically remove a loadable driver from the running system. The prototype is

```
int pfxunload(void);
```

A driver can be unloaded either because all its devices are closed and a timeout has elapsed, or because the operator has used the *ml* command (see the ml(1) reference page). The kernel does not unload a driver unless the driver provides a *pfx***unload()** entry point. Without this entry point, the driver can be dynamically loaded, but then remains in memory.

It is not easy to retain state information about the device over the time when the driver is not in memory. The entire text and data of a loadable driver, including static variables, are removed and reloaded. Only global variables defined in the descriptive file (see "Describing the Driver in /var/sysgen/master.d" on page 233) remain in memory after the driver is unloaded. Be sure not to store any addresses of driver code or driver static variables in global variables, since these addresses will be different when the driver is reloaded.

The driver may have allocated dynamic memory. This should be released, because the addresses of allocated memory will be lost when the driver is unloaded, and more will be allocated if the driver is reloaded. For example, the driver should use **phfree()** to release a pollhead structure allocated by **phalloc()** (see "Use and Operation of poll(2)" on page 155, and the phalloc(D3) and phfree(D3) reference pages). It is also the time to release any PIO maps using **pio_mapfree()** (see "Mapping PIO Addresses" on page 325), and to release any process handles (see "Sending a Process Signal" on page 203).

The driver is not required to unload. If the driver should not be unloaded at this time, it returns a nonzero return code to the call, and the kernel does not unload it. There are several reasons why a driver should not be unloaded.

The kernel calls *pfx***unload()** only when no device special files managed by the driver are open. If any device had been opened, the *pfx***close()** entry has been called. However, if any device was mapped through the *pfx***map()** entry, the mapping could still exist. If the driver has any resources tied up in association with a memory mapping, it should return a nonzero value to the *pfx***unload** call.

A driver should never permit unloading when there is any kind of pointer to the driver held in any kernel data structure. It is a frequent design error to unload when there is a live pointer to the driver. Unpredictable kernel panics often result.

One example of a live pointer to a driver is a pending callback function. Any pending **itimeout()** or **bufcall()** timers should be cancelled before returning 0 from *pfx***unload()**. A driver for the EISA or GIO bus can register an interrupt handler. There is no way to retract the registration of an EISA interrupt handler (see "Allocating and Programming an IRQ" on page 442), so an EISA driver should never permit unloading once an IRQ has been programmed. A GIO driver is able to unregister an interrupt handler (see "GIO-Specific Kernel Functions" on page 515), and must do so before it permits unloading.

Entry Point halt()

The kernel calls the *pfx***halt()** entry point, if one exists, while performing an orderly system shutdown (see the halt(1) reference page). No other driver entry points are called after this one. The prototype is simply

```
void pfxhalt(void);
```

Since the system is shutting down, there is no point in returning allocated memory. The only purpose this entry point can serve is to leave the device in a safe and stable condition. For example, this is the place at which a disk driver could command the heads of the drive to move to a safe zone for power off.

The driver cannot assume that interrupts are disabled or enabled. The driver cannot block waiting for device actions, so whatever commands it issues to the device must take effect immediately.

Entry Point size()

The *pfx***size()** entry point is required of block device drivers. It reports the size of the device in “sector” units, where a “sector” size is declared as NBPSCTR in *sys/param.h* (currently 512). The prototype is

```
int pfxsize(dev_t dev);
```

The device major and minor numbers can be extracted from the dev argument. The entry point is not called until *pfx***open()** has been called. Typically the driver will calculate the size of the medium during *pfx***open()**.

Since the *int* return value is 32 bits in all systems, the largest possible block device is 1,024 gigabytes ((2^{31}*512)/$1{,}024^3$).

Entry Point print()

The *pfx***print()** entry point is called from the kernel to display a diagnostic message when an error is detected on a block device. The prototype and the complete logic of the entry point is shown in Example 8-5.

Example 8-5 Entry Point pfxprint()

```
#include <sys/cmn_err.h>
#include <sys/ddi.h>
int hypo_print(dev_t dev, char *str)
{
   cmn_err(CE_NOTE,"Error on dev %d: %s\n",geteminor(dev),str);
   return 0;
}
```

Handling 32-Bit and 64-Bit Execution Models

The *pfx***ioctl()** entry point can be passed a data structure from the user process address space; that is, the *arg* value can be a pointer to a structure or an array of data. In order to interpret such a structure, the driver has to know the execution model for which the user process was compiled.

The execution model is specified when code is compiled. The 32-bit model (compiler option -32 or -n32) uses 32-bit address values and a *long int* contains 32 bits. The 64-bit model (compiler option -64) uses 64-bit address values and a *long int* contains 64 bits. (The size of an unqualified *int* is 32 bits in both models.) The execution model is sometimes casually called the "ABI" (Authorized Binary Interface), but this is an improper use of that term—an ABI comprises calling conventions, public names, and structure definitions, as well as the execution model.

An IRIX kernel compiled to the 32-bit model contains 32-bit drivers and supports only 32-bit user processes. A kernel compiled to the 64-bit model contains 64-bit drivers, but it supports user processes compiled to *either* 32-bit or 64-bit models. Therefore, in a 64-bit kernel, a driver can be asked to interpret data produced by a 32-bit program.

This is true only of the *pfx***ioctl()** and *pfx***poll()** entry points. Other driver entry points move data to and from user space as streams or blocks of bytes—not as a structure with fields to be interpreted.

Since in other respects it is easy to make your driver portable between 64-bit and 32-bit systems, you should design your driver so that it can handle the case of operating in a 64-bit kernel, receiving **ioctl()** requests alternately from 32-bit and 64-bit programs.

The simplest way to do this is to define the arguments passed to the entry points in such a way that they have the same precision in either system. However, this is not always possible. To handle the general case, the driver must know to which model the user process was compiled.

You find this out by calling the **userabi()** kernel function (for which, unfortunately, there is no reference page available).

The prototype of **userabi()** (declared in *sys/ddi.h*) is

```
int userabi(__userabi_t *);
```

If there is no user process context, **userabi()** returns ESRCH. Otherwise it fills out a *__userabi_t* structure and returns 0. The structure of type *__userabi_t* (declared in *sys/types.h*) contains the fields listed below:

uabi_szint	Size of a user int (4).
uabi_szlong	Size of a user long (4 or 8).
uabi_szptr	Size of a user address (4 or 8).
uabi_szlonglong	Size of a user long long (8).

Store the value of *uabi_szptr* when opening a device. Then you can use it to choose between 32-bit and 64-bit declarations of a structure passed to *pfx***ioctl()** or an address passed to *pfx***poll()**.

Planning for Multiprocessor Use

Multiprocessor computers are a central part of the Silicon Graphics product line and will become increasingly common in the future. A device driver that is not multiprocessor-ready can be used in a multiprocessor, but it is likely to cause a performance bottleneck. A multiprocessor-ready driver, on the other hand, works well in a uniprocessor with little if any loss of speed.

The Multiprocessor Environment

A multiprocessor has two or more CPU modules, all of the same type. The CPUs execute independently, but all share the same main memory. Any CPU can execute the code of the IRIX kernel, and it is common for two or more CPUs to be executing kernel code, including driver code, simultaneously.

Uniprocessor Assumptions

The original UNIX architecture assumed a uniprocessor hardware environment with a hierarchy of interrupt levels. Ordinary code could be preempted by an interrupt, but an interrupt handler could only be preempted by an interrupt at a higher level.

This assumed hardware environment was reflected in the design of device drivers and kernel support functions.

- In a uniprocessor, an upper-half driver entry point such as *pfx***open()** cannot be preempted except by an interrupt. It has exclusive access to driver variables except for those changed by the interrupt handler.
- Once in an interrupt handler, no other code can possibly execute except an interrupt of a higher hardware level. The interrupt handler has exclusive access to driver variables.
- The interrupt handler can use kernel functions such as **splhi()** to set the hardware interrupt mask, blocking interrupts of all kinds, and thus getting exclusive access to all memory including kernel data structures.

All of these assumptions fail in a multiprocessor.

- Upper-half entry points can be entered concurrently on multiple CPUs. For example, one CPU can be executing *pfx***open()** while another CPU is in *pfx***strategy()**. Exclusive use of driver variables cannot be assumed.
- An interrupt can be taken on one CPU while upper-half routines or a timeout function execute concurrently on other CPUs. The interrupt routine cannot assume exclusive use of driver variables.
- Interrupt-level functions such as **splhi()** are meaningless, since at best they set the interrupt mask on the current CPU only. Other CPUs can accept interrupts at all levels. The interrupt handler can never gain exclusive access to kernel data.

The process of making a driver multiprocessor-ready consists of changing all code whose correctness depends on uniprocessor assumptions.

Protecting Common Data

Whenever a common resource can be updated by two processes concurrently, the resource must be protected by a *lock* that represents the exclusive right to update the resource. Before changing the resource, the software acquires the lock, claiming exclusive access. After changing the resource, the software releases the lock.

The IRIX kernel provides a set of functions for creating and using locks. It provides another set of functions for creating and using *semaphore* objects, which are like locks but sometimes more flexible. Both sets of functions are discussed under "Waiting and Mutual Exclusion" on page 204.

Sleeping and Waking

Sometimes the lock is not available—some other process executing in another CPU has acquired the lock. When this happens, the requesting process is delayed in the lock function until the lock is free. To delay, or *sleep*, is allowed for upper-half entry points, because they execute (in effect) as subroutines of user processes.

Interrupt handlers and timeout functions are not permitted to sleep. They have no process identity and so there is no mechanism for saving and restoring their state. An interrupt handler can test a lock, and can claim the lock conditionally, but if a lock is already held, the handler must have some alternate way of storing data.

Synchronizing Within Upper-Half Functions

When designing an upper-half entry point, keep in mind that it could be executed concurrently with any other upper-half entry point, and that the one entry point could even be executed concurrently by multiple CPUs. Only a few entry points are immune:

- The *pfx***init()**, *pfx***edtinit()**, and *pfx***start()** entry points cannot be entered concurrently with each other or any other entry point (*pfx***start()** could be entered concurrently with the interrupt handler).
- The *pfx***unload()** and *pfx***halt()** entry points cannot be entered concurrently with any other entry point except for stray interrupts.
- Certain entry points have no cause to use shared data; for example, *pfx***size()** and *pfx***print()** normally do not need to take any precautions.
- Other upper-half entry points, and all STREAMS entry points, can be entered concurrently by multiple CPUs, when the driver is multiprocessor-aware.

You can deal with concurrency at different levels of sophistication.

Running on CPU 0

If you do not set the D_MP flag in a character driver or STREAMS driver (see "Flag D_MP" on page 141), the driver is executed only on CPU 0. As a result, upper-half entry points cannot execute concurrently, and the interrupt handler cannot run in true concurrency with an upper-half routine (although it can preempt an upper-half routine as it does in a uniprocessor).

The result is that user processes are serialized for the use of the driver for any purpose. Since CPU 0 is often busy with other housekeeping activities, access to the driver can have a latency that is long and variable.

Serializing on a Single Lock

You can create a single lock for upper-half serialization. Each upper-half function begins with read-only operations such as extracting the device minor number and testing and validating arguments. You allow these to execute concurrently on any CPU (the D_MP flag is set.)

In each enry point, when the preliminaries are complete, you acquire the single lock, and release it just before returning. The result is that processes are serialized for I/O through the driver. If the driver supports only a single device, processes would be serialized in any case, waiting for the device to operate. Since the upper half can execute on any CPU, latency is more predictable.

Serializing on a Lock Per Device

When the driver supports multiple minor devices, you will normally have a data structure per device, indexed by the device minor number. Typically an upper-half routine is concerned only with one minor device. You can define a lock in the data structure for the minor device, and acquire that lock as soon as the device number is known.

This permits concurrent execution of upper-half requests for different minor devices, while serializing access to any one device.

Coordinating Upper-Half and Interrupt Entry Points

Upper-half entry points prepare work for the device to do, and the interrupt routine reports the completion of the device action. In a block device driver, this communication is relatively simple. In a character driver, you have more design options. The kernel functions mentioned in the following topics are covered under "Waiting and Mutual Exclusion" on page 204.

Coordinating Through the buf_t

In a block device driver, the *pfx***strategy()** routine initiates a read or a write based on a *buf_t* structure (see "Entry Point strategy()" on page 154), and leaves the address of the *buf_t* where the interrupt routine can find it. Then *pfx***strategy()** calls the **biowait()** kernel function to wait for completion.

The *pfx***intr()** entry point updates the *buf_t* (using *pfx***bioerror()** if necessary) and then uses **biodone()** to mark the *buf_t* as complete. This ends the wait for *pfx***strategy()**. These kernel functions are multiprocessor-aware.

Coordination in a Character Driver

In a character driver that supports interrupts, you design your own coordination mechanism. The simplest (and not recommended) would be based on using the kernel function **sleep()** in the upper half, and **wakeup()** in the interrupt routine. You can also use a semaphore and use **psema()** in the upper half and **vsema()** in the interrupt handler.

If you need to allow for timeouts, you have to deal with the complication that the timeout function can be called concurrently with an interrupt. When you use a semaphore, the interrupt routine can use **vsema()** to post completion, and the timeout function can use **cvsema()** to post it only if it has not already been posted.

Converting a Uniprocessor Driver

As a general approach, you can convert a uniprocessor driver to make it multiprocessor-safe in the following steps:

1. If it currently uses the D_OLD flag (or has no *pfx***devflag** constant), convert it to use the current interface, with a *pfx***devflag** of 0x00.
2. Make sure it works in the original uniprocessor at the current release of IRIX.
3. Test it in a multiprocessor running in CPU 0.
4. Begin adding semaphores, locks, and other exclusion and synchronization tools. Since the driver still runs serially on CPU 0, it will never wait for a lock, but the coordination between upper half and interrupt handler should work.
5. Add the D_MP flag and test on a multiprocessor.

In performing the conversion, you can use calls to **spl**..**()** functions as signs that work is needed. These functions are used for mutual exclusion in a uniprocessor, and they are all ineffective or unnecessary in a multiprocessor-safe driver.

Example Conversion Problem

The code in Example 8-6 shows typical logic in a uniprocessor character driver.

Example 8-6 Uniprocessor Upper-Half Wait Logic

```
s = splvme();
flag |= WAITING;
while (flag & WAITING) {
   sleep(&flag, PZERO);
}
splx(s);
```

The upper half calls the **splvme()** function with the intention of blocking interrupts, and thus preventing execution of this driver's interrupt handler while the *flag* variable is updated. In a multiprocessor this is ineffective because at best it sets the interrupt level on the current CPU. The interrupt handler can execute on another CPU and change the variable.

The corresponding interrupt handler is sketched in Example 8-7.

Example 8-7 Uniprocessor Interrupt Logic

```
if (flag & WAITING) {
   wakeup(&flag);
   flag &= ~WAITING;
}
```

The interrupt handler could execute on another CPU, and test the flag after the upper half has called **splvme()** and before it has set WAITING in *flag*. The interrupt is effectively lost. This would happen rarely and would be hard to repeat, but it would happen and would be hard to trace.

A more reliable, and simpler, technique is to use a semaphore. The driver defines a global semaphore:

```
static sema_t sleeper;
```

A driver with multiple devices would have a semaphore per device, perhaps as an array of *sema_t* items indexed by device minor number.

The semaphore (or array) would be initialized to a starting value of 1 in the *pfx***init()** or *pfx***start()** entry:

```
void hypo_start()
{
...
   initnsema(&sleeper,1,"sleeper");
}
```

After the upper half started a device operation, it would await the interrupt using **psema()**:

```
psema(sleeper,PZERO);
```

The PZERO argument makes the wait immune to signals. If the driver should wake up when a signal is sent to the calling process (such as SIGINT or SIGTERM), the second argument can be PCATCH. A return value of -1 indicates the semaphore was posted by a signal, not by a **vsema()** call.

The interrupt handler would use **vsema()** or **cvsema()** to post the semaphore. The use of **cvsema()** ensures that the semaphore is not incremented past 1, in the event that it is posted from more than one location (as from a timeout or a signal handler).

Chapter 9

Device Driver/Kernel Interface

The programming interface between a device driver and the IRIX kernel is completely documented in the reference pages in volume "D." This chapter provides a survey and a summary of the API under the following headings:

- "Important Data Types" on page 180 describes the data types that are exchanged between the kernel and a driver.
- "Important Header Files" on page 185 summarizes the C header files that are frequently included in a driver source file.
- "Memory Allocation" on page 187 describes the kernel functions for general memory allocation, for allocation of objects of specific types, and for resource suballocation.
- "Transferring Data" on page 192 describes the functions for transferring data between a driver and a buffer in the address space of either the kernel or a user process.
- "Managing Virtual and Physical Addresses" on page 195 discusses the translation from virtual to physical storage locations for DMA.
- "User Process Administration" on page 202 discusses process signalling and authentication.
- "Waiting and Mutual Exclusion" on page 204 describes and summarizes a wide array of functions you can use for those purposes.

In addition to these topics, data types and functions specific to the following areas are in the chapters shown:

Debugging and logging	Chapter 11, "Testing and Debugging a Driver"
EISA Bus	Chapter 17, "EISA Device Drivers"
GIO Bus	Chapter 18, "GIO Device Drivers"
SCSI bus	Chapter 15, "SCSI Device Drivers"
STREAMS drivers	Chapter 19, "STREAMS Drivers"
VME bus	Chapter 14, "Services for VME Drivers"

Important Data Types

The Device Number Types

Two numbers are carried in the inode of a *device special file*: a *major device number* of up to 9 bits, and a *minor device number* of up to 18 bits. The numbers are assigned when the device special file is created, either by the */dev/MAKEDEV* script or by the system administrator. The contents and meaning of device numbers is discussed under "Device Representation" on page 33.

At almost every upper-half entry point, the first argument to a driver is a *dev_t* object, an unsigned integer containing the values of the major and minor numbers for the device that is to be used. The *dev_t* type is declared in *sys/types.h* along with types *major_t* and *minor_t*, which represent major and minor numbers as variables.

Use of the Device Numbers

You typically use the major device number to learn which device driver has been called. This is important only when a device driver supports multiple interfaces, for example when one driver represents both character and block access to the same hardware.

You use the minor device number to learn which hardware unit is being accessed. This is of interest only when a driver supports multiple units. In addition, device management options can be encoded into the minor number, as described under "Minor Device Number" on page 35.

Device Number Functions

The kernel provides several functions for manipulating device numbers, and these are summarized in Table 9-1.

Table 9-1 Functions to Manipulate Device Numbers

Function	Header Files	Can Sleep	Purpose
etoimajor(D3)	ddi.h	N	Convert external to internal major device number.
getemajor(D3)	ddi.h	N	Get external major device number.
geteminor(D3)	ddi.h	N	Get external minor device number.
getmajor(D3)	ddi.h	N	Get internal major device number.
getminor(D3)	ddi.h	N	Get internal minor device number.
itoemajor(D3)	ddi.h	N	Convert internal to external major device number.
makedevice(D3)	ddi.h	N	Make device number from major and minor numbers.

The most important of these are

- **getemajor()**, which extracts the major number from a *dev_t* and returns it as a *major_t*
- **geteminor()**, which extracts the minor number from a *dev_t* and returns it as a *minor_t*
- **makedevice()**, which combines a *major_t* and a *minor_t* to form a *dev_t*

External and Internal Numbers

The kernel uses the major device number as a subscript to index various tables. Some variants of UNIX, in order to avoid wasting space on sparse tables, translate the major device number to an internal code. Sometimes the minor number is translated too.

This internal encoding of the device number is of no interest in IRIX. If it is done, it is done only for the purpose of subscripting tables within the kernel that are not accessible to device drivers. Internal device numbers have no utility in IRIX. However, functions related to internal device numbers are included for compatibility with SVR4.

If you are writing a new device driver specifically for IRIX, use only external device numbers. If you are porting a device driver that uses the **getmajor()**, **getminor()**, **etoimajor()** and **etoiminor()** functions, you can leave these function calls unchanged. (But if the driver attempts to access the kernel switch tables, it is nonportable and should be changed.)

Structure uio_t

The *uio_t* structure describes data transfer for a character device:

- The *pfx***read()** and *pfx***write()** entry points receive a *uio_t* that describes the buffer of data.
- Within an *pfx***ioctl()** entry point, you might construct a *uio_t* to represent data transfer for control purposes.
- In a hybrid character/block driver, the **physiock()** function translates a *uio_t* into a *buf_t* for use by the *pfx***strategy()** entry point.

The fields and values in a *uio_t* are declared in *sys/uio.h*, which is included by *sys/ddi.h*. For a detailed discussion, see the uio(D4) reference page. Typically the contents of the *uio_t* reflect the buffer areas that were passed to a **read()**, **readv()**, **write()**, or **writev()** call (see the read(2) and write(2) reference pages).

Data Location and the iovec_t

One *uio_t* describes data transfer to or from a single address space, either the address space of a user process or the kernel address space. The address space is indicated by a flag value, either UIO_USERSPACE or UIO_SYSSPACE, in the *uio_segflg* field.

The total number of bytes remaining to be transferred is given in field *uio_resid*. Initially this is the total requested transfer size.

Although the transfer is to a single address space, it can be directed to multiple segments of data within the address space. Each segment of data is described by a structure of type *iovec_t*. An *iovec_t* contains the virtual address and length of one segment of memory.

The number of segments is given in field *uio_iovcnt*. The field *uio_iov* points to the first *iovec_t* in an array of *iovec_t* structures, each describing one segment. of data. The total size in *uio_resid* is the sum of the segment sizes.

For a simple data transfer, *uio_iovcnt* contains 1, and *uio_iov* points to a single *iovec_t* describing a buffer of 1 or more bytes. For a complicated transfer, the *uio_t* might describe a number of scattered segments of data. Such transfers can arise in a network driver where multiple layers of message header data are added to a message at different levels of the software.

Use of the uio_t

In the *pfx***read()** and *pfx***write()** entry points, you can test *uio_segflag* to see if the data is destined for user space or kernel space, and you can save the initial value of *uio_resid* as the requested length of the transfer.

In a character driver, you fetch or store data using functions that both use and modify the *uio_t*. These functions are listed under "Transferring Data Through a uio_t Object" on page 194. When data is not immediately available, you should test for the FNDELAY or FNONBLOCK flags in *uio_fmode*, and return when either is set rather than sleeping.

Structure buf_t

The *buf_t* structure describes a block data transfer. It is designed to represent the transfer (in or out) of a sequence of adjacent, fixed-size blocks from a random-access device to a block of contiguous memory. The size of one device block is NBPSCTR, declared in *sys/param.h*. For a detailed discussion of the *buf_t*, see the buf(D4) reference page.

The *buf_t* is used internally in IRIX by the paging I/O system to manage queues of physical pages, and by filesystems to manage queues of pages of file data. The paging system and filesystems are the primary clients of the *pfx***strategy()** entry point to a block device driver, so it is only natural that a *buf_t* pointer is the input argument to *pfx***strategy()**.

Tip: The idbg kernel debugging tool has several functions related to displaying the contents of *buf_t* objects. See "Commands to Display buf_t Objects" on page 268.

Fields of buf_t

The fields of the *buf_t* are declared in *sys/buf.h*, which is included by *sys/ddi.h*. This header file also declares the names of many kernel functions that operate on *buf_t* objects. (Many of those functions are not supported as part of the DDABI. You should only use kernel functions that have reference pages.)

Because *buf_t* is used by so many software components, it has many fields that are not relevant to device driver needs, as well as some fields that have multiple uses. The relevant fields include the following:

b_edev	*dev_t* giving device major and minor numbers.
b_flags	Operational flags; for a detailed list see buf(D4).
b_forw, *b_back*, *av_forw*, *av_back*	Queuing pointers, available for driver use within the *pfx***strategy()** routine.
b_un.b_addr	Sometimes the kernel virtual address of the buffer, depending on the *b_flags* setting BP_ISMAPPED.
b_bcount	Number of bytes to transfer.
b_blkno	Starting logical block number on device.
b_iodone	Address of a driver internal function to be called on I/O completion.
b_resid	Number of bytes not transferred, set at completion to 0 unless an error occurs.
b_error	Error code, set at completion of I/O.

Using the Logical Block Number

The logical block number is the number of the 512-byte block in the device. The “device” is encoded by the minor device number that you can extract from *b_edev*. It might be a complete device surface, or it might be a partition within a larger device (for example, the IRIX disk device drivers support different minor device numbers for different disk partitions).

The *pfx***strategy()** routine may have to translate the logical block number based on the driver’s information about device partitioning and device geometry (sector size, sectors per track, tracks per cylinder).

Buffer Location and b_flags

The data buffer represented by a *buf_t* can be in one of two places, depending on bits in *b_flags*.

When the macro BP_ISMAPPED(*buf_t-address*) returns true, the buffer is in kernel virtual memory and its virtual address is in *b_un.b_addr*.

When BP_ISMAPPED(*buf_t-address*) returns false, the buffer is described by a chain of *pfdat* structures (declared in *sys/pfdat.h*, but containing no fields of any use to a device driver). In this case, *b_un.b_addr* contains only an offset into the first page frame of the chain. See “Managing Buffer Virtual Addresses” on page 199 for a method of mapping an unmapped buffer.

Lock and Semaphore Types

The header files *sys/sema.h* and *sys/types.h* declare the data types of locks of different types, including the following:

lock_t	Basic lock, or spin-lock, used with LOCK() and related functions
mutex_t	Sleeping lock, used for mutual exclusion between upper-half instances.
sema_t	Semaphore object, used for general locking.
mrlock_t	Reader-writer locks, used with RW_RDLOCK() and related functions.
sv_t	Synchronization variable, used with SV_WAIT and related functions

These lock types should be treated as opaque objects because their contents can change from release to release (and in fact their contents are different in IRIX 6.2 from previous releases).

The families of locking and synchronization functions contain functions for allocating, initializing, and freeing each type of lock. See “Waiting and Mutual Exclusion” on page 204.

Important Header Files

The header files that are frequently needed in device driver source modules are summarized in Table 9-2.

Table 9-2 Header Files Often Used in Device Drivers

Header File	Reason for Including
sys/buf.h	The *buf_t* structure and related constants and functions (included by *sys/ddi.h*).
sys/cmn_err.h	The **cmn_err()** function.

Table 9-2 (continued) Header Files Often Used in Device Drivers

Header File	Reason for Including
sys/conf.h	The constants used in the *pfx***devflags** global.
sys/ddi.h	Many kernel functions declared. Also includes *sys/types.h*, *sys/uio.h*, and *sys/buf.h*.
sys/debug.h	Defines the ASSERT macro and others.
sys/dmamap.h	Data types and kernel functions related to DMA mapping.
sys/edt.h	Declares the *edt_t* type passed to *pfx***edtinit()**.
sys/eisa.h	EISA-bus hardware constants and EISA kernel functions.
sys/errno.h	Names for all system error codes.
sys/file.h	Names for file mode flags passed to driver entry points.
sys/immu.h	Types and macros used to manage virtual memory and some kernel functions.
sys/kmem.h	Constants like KM_SLEEP used with some kernel functions.
sys/ksynch.h	Functions used for sleep-locks.
sys/log.h	Types and functions for using the system log.
sys/major.h	Names for assigned major device numbers.
sys/map.h	Types and functions used for suballocation using **rmalloc()**.
sys/mman.h	Constants and flags used with **mmap()** and the *pfx***mmap()** entry point.
sys/param.h	Constants like PZERO used with some kernel functions.
sys/pio.h	VME PIO functions.
sys/poll.h	Types and functions for pollhead allocation and poll callback.
sys/scsi.h	Types and functions used to call the inner SCSI driver.
sys/sema.h	Types and functions related to semaphores, mutex locks, and basic locks.
sys/stream.h	STREAMS standard functions and data types.
sys/strmp.h	STREAMS multiprocessor functions.
sys/sysmacros.h	Macros for conversion between bytes and pages, and similar values.

Table 9-2 (continued) Header Files Often Used in Device Drivers

Header File	**Reason for Including**
sys/systm.h	Kernel functions related to system operations.
sys/types.h	Common data types and types of system objects (included by *sys/ddi.h*).
sys/uio.h	The *uio_t* structure and related functions (included by *sys/ddi.h*).
sys/vmereg.h	VME bus hardware constants and VME-related functions.

Memory Allocation

A device or STREAMS driver can allocate memory statically, as global variables in the driver module, and this is a good way to allocate any object that is always needed and has a fixed size.

When the number or size of an object can vary, but can be determined at initialization time, the driver can allocate memory in the *pfx***init()**, *pfx***edtinit()**, or *pfx***start()** entry point.

You can allocate memory dynamically in an upper-half entry point. When this is necessary, it should be done in an entry point that is called infrequently, such as *pfx***open()**. The reason is that memory allocation is subject to unpredictable delays.

Memory allocation should never be attempted in an interrupt routine. The resources that might be needed at interrupt time should be obtained and set aside by an upper-half entry point before the interrupt is made possible.

General-Purpose Allocation

There are two groups of general-purpose functions used to allocate and release memory.

- **kmem_alloc()** and two associated functions supply a complete set of services for allocating kernel virtual memory.
- **kern_malloc()** and two associated functions are an obsolete mechanism for allocating kernel virtual memory.

The functions you can use to dynamically allocate kernel virtual memory are summarized in Table 9-3.

Table 9-3 Functions for Kernel Virtual Memory

Function Name	Header Files	Can Sleep?	Purpose
kmem_alloc(D3)	kmem.h & types.h	Y	Allocate space from kernel free memory.
kmem_free(D3)	kmem.h & types.h	N	Free previously allocated kernel memory.
kmem_zalloc(D3)	kmem.h & types.h	Y	Allocate and clear space from kernel free memory.
kern_calloc(D3)	systm.h & types.h	Y	Allocate space from kernel memory and clear it.
kern_free(D3)	systm.h & types.h	N	Free kernel memory space.
kern_malloc(D3)	systm.h & types.h	Y	Allocate kernel virtual memory.

The most important of these functions is **kmem_alloc()**. You use it to allocate blocks of virtual memory at any time. It offers these important options, controlled by a flag argument:

- Sleeping or not sleeping when space is not available. You specify not-sleeping when in a lower-half routine or when holding a basic lock, but then you must be prepared to deal with a return value of NULL.
- Physically-contiguous memory. The memory allocated is virtual, and when it spans multiple pages, the pages are not necessarily adjacent in physical memory. You need physically contiguous pages when doing DMA with a device that cannot do scatter/gather. However, contiguous memory is harder to get as the system runs, so it is best to obtain it in an initialization routine.
- Cache-aligned memory. By requesting memory that is a multiple of a cache line in size, and aligned on a cache-line boundary, you ensure that DMA operations will affect the fewest cache lines (see “Setting Up a DMA Transfer” on page 198).

The **kmem_zalloc()** function takes the same options, but offers the additional service of zero-filling the allocated memory.

Calls to the “kern” group of functions should be replaced as follows:

kern_malloc(*n*)	Change to **kmem_alloc**(*n*,KM_SLEEP).
kern_calloc(*n*,*s*)	Change to **kmem_zalloc**(*n***s*,KM_SLEEP)
kern_free(*p*)	Change to **kmem_free**(*p*)

Allocating Objects of Specific Kinds

The kernel provides a number of functions with the purpose of allocating and freeing objects of specific kinds. Many of these are variants of **kmem_alloc()** and **kmem_free()**, but others use special techniques suited to the type of object.

Allocating pollhead Objects

Table 9-4 summarizes the functions you use to allocate and free the *pollhead* structure that is used within the *pfx***poll()** entry point (see “Entry Point poll()” on page 156). Typically you would call **phalloc()** while initializing each minor device, and call **phfree()** in the *pfx***unload()** entry point.

Table 9-4 Functions for Allocating pollhead Structures

Function Name	Header Files	Can Sleep?	Purpose
phalloc(D3)	ddi.h & kmem.h & poll.h	Y	Allocate and initialize a pollhead structure.
phfree(D3)	ddi.h & poll.h	N	Free a pollhead structure.

Allocating Semaphores and Locks

There are symmetrical pairs of functions to allocate and free all types of lock and synchronization objects. These functions are summarized together with the other locking functions under “Waiting and Mutual Exclusion” on page 204.

Allocating buf_t Objects and Buffers

The argument to the *pfx***strategy()** entry point is a *buf_t* structure that describes a buffer (see "Entry Point strategy()" on page 154 and "Structure buf_t" on page 183).

Ordinarily, both the *buf_t* and the buffer are allocated and initialized by the kernel or the filesystem that calls *pfx***strategy()**. However, some drivers need to create a *buf_t* and associated buffer for special uses. The functions summarized in Table 9-5 are used for this.

Table 9-5 Functions for Allocating buf_t Objects and Buffers

Function Name	Header Files	Can Sleep?	Purpose
geteblk(D3)	ddi.h	Y	Allocate a *buf_t* and a buffer of 1024 bytes.
ngeteblk(D3)	ddi.h	Y	Allocate a *buf_t* and a buffer of specified size.
brelse(D3)	ddi.h	N	Return a buffer header and buffer to the system.
getrbuf(D3)	ddi.h	Y	Allocate a *buf_t* with no buffer.
freerbuf(D3)	ddi.h	N	Free a *buf_t* with no buffer.

To allocate a *buf_t* and its associated buffer in kernel virtual memory, use either **geteblk()** or **ngeteblk()**. Free this pair of objects using **brelse()**, or by calling **biodone()**.

You can allocate a *buf_t* to describe an existing buffer—one in user space, statically allocated in the driver, or allocated with **kmem_alloc()—**using **getrbuf()**. Free such a *buf_t* using **freerbuf()**.

Suballocation Functions

The functions summarized in Table 9-6 are used to manage suballocation of any resource.

Table 9-6 Functions for Suballocation

Function Name	**Header Files**	**Can Sleep?**	**Purpose**
rmalloc(D3)	map.h & types.h	N	Allocate space from a private space management map.
rmalloc_wait(D3)	map.h & types.h	Y	Allocate resources from a space management map.
rmallocmap(D3)	map.h & types.h	N	Allocate and initialize a private space management map.
rmfree(D3)	map.h & types.h	N	Release resources into a space management map.
rmfreemap(D3)	map.h & types.h	N	Free a private space management map.

You use these functions as a convenient, efficient set of subroutines for allocating some resource—for example, disk sectors—that you obtain by other means. The expected sequence of use is as follows.

1. During driver initialization, or possibly in *pfx***open()**, use **rmallocmap()** to allocate a map. A map is a data structure large enough to keep track of as many objects as you will create. Initially the map reflects no available resources.
2. Use **rmfree()** to release existing resources into the map. For example, while opening a disk drive, you could use **rmfree()** to release all unused sectors into a sector map.
3. When a resource is needed in an upper-half routine, use **rmalloc()** or **rmalloc_wait()** to acquire it. The index number of the first allocated item is returned.
4. When a resource is released in any entry point, use **rmfree()** to note the available items and to wake up any upper-half process waiting in **rmalloc_wait()**.
5. On device close or when the driver is unloaded, use **rmfreemap()** to release the map itself.

Transferring Data

The device driver executes in the kernel virtual address space, but it must transfer data to and from the address space of a user process. The kernel supplies two kinds of functions for this purpose:

- functions that transfer data between driver variables and the address space of the current process
- functions that transfer data between driver variables and the buffer described by a *uio_t* object

Warning: **The use of an invalid address in kernel space with any of these functions causes a kernel panic.**

All functions that reference an address in user process space can sleep, because the page of process space might not be resident in memory. As a result, such functions cannot be used in an interrupt handler, or while holding a basic lock.

General Data Transfer

The kernel supplies functions for clearing and copying memory within the kernel virtual address space, and between the kernel address space and the address space of the user process that is the current context. These general-purpose functions are summarized in Table 9-7.

Table 9-7 Functions for General Data Transfer

Function Name	Header Files	Can Sleep?	Purpose
bcopy(D3)	ddi.h	N	Copy data between address locations in the kernel.
bzero(D3)	ddi.h	N	Clear memory for a given number of bytes.
copyin(D3)	ddi.h	Y	Copy data from a user buffer to a driver buffer.
copyout(D3)	ddi.h	Y	Copy data from a driver buffer to a user buffer.
fubyte(D3)	systm.h & types.h	Y	Load a byte from user space.

Table 9-7 (continued) Functions for General Data Transfer

Function Name	Header Files	Can Sleep?	Purpose
fuword(D3)	systm.h & types.h	Y	Load a word from user space.
hwcpin(D3)	systm.h & types.h	N	Copy data from device registers to kernel memory.
hwcpout(D3)	systm.h & types.h	N	Copy data from kernel memory to device registers.
subyte(D3)	systm.h & types.h	Y	Store a byte to user space.
suword(D3)	systm.h & types.h	Y	Store a word to user space.

Block Copy Functions

The **bcopy()** and **bzero()** functions are used to copy and clear data areas within the kernel address space, for example driver buffers or work areas. These are optimized routines that take advantage of available hardware features.

The **bcopy()** function is not appropriate for copying data between a buffer and a device; that is, for copying between virtual memory and the physical memory addresses that represent a range of device registers (or indeed any uncached memory). The reason is that **bcopy()** uses doubleword moves and any other special hardware features available, and devices many not be able to accept data in these units. The **hwcpin()** and **hwcpout()** functions copy data in 16-bit units; use them to transfer bulk data between device space and memory. (Use simple assignment to move single words or bytes.)

The **copyin()** and **copyout()** functions take a kernel virtual address, a process virtual address, and a length. They copy the specified number of bytes between the kernel space and the user space. They select the best algorithm for copying, and take advantage of memory alignment and other hardware features.

If there is no current context, or if the address in user space is invalid, or if the address plus length is not contained in the user space, the functions return -1. This indicates an error in the request passed to the driver entry point, and the driver normally returns an EFAULT error.

Byte and Word Functions

The functions **fubyte()**, **subyte()**, **fuword()**, and **suword()** are used to move single items to or from user space. When only a single byte or word is needed, these functions have less overhead than the corresponding **copyin()** or **copyout()** call. For example you could use **fuword()** to pick up a parameter using an address passed to the *pfx***ioctl()** entry point. When transferring more than a few bytes, a block move is more efficient.

Transferring Data Through a uio_t Object

A *uio_t* object defines a list of one or more segments in the address space of the kernel or a user process (see "Structure uio_t" on page 182). The kernel supplies three functions for transferring data based on a *uio_t*, and these are summarized in Table 9-8.

Table 9-8 Functions Moving Data Using uio_t

Function	Header Files	Can Sleep?	Purpose
uiomove(D3)	ddi.h	Y	Copy data using *uio_t*.
ureadc(D3)	ddi.h	Y	Copy a character to space described by *uio_t*.
uwritec(D3)	ddi.h	Y	Return a character from space described by *uio_t*.

The **uiomove()** function moves multiple bytes between a buffer in kernel virtual space—typically, a buffer owned by the driver—and the space or spaces described by a *uio_t*. The function takes a byte count and a direction flag as arguments, and uses the most efficient mechanism for copying.

The **ureadc()** and **uwritec()** functions transfer only a single byte. You would use them when transferring data a byte at a time by PIO. When moving more than a few bytes, **uiomove()** is faster.

All of these functions modify the *uio_t* to reflect the transfer of data:

- *uio_resid* is decremented by the amount moved
- In the *iovec_t* for the current segment, *iov_base* is incremented and *iov_len* is decremented
- As segments are used up, *uio_iov* is incremented and *uio_iovcnt* is decremented

The result is that the state of the *uio_t* always reflects the number of bytes remaining to transfer. When the *pfx***read()** or *pfx***write()** entry point returns, the kernel uses the finsl value of *ui_resid* to compute the count returned to the **read()** or **write()** function call.

Managing Virtual and Physical Addresses

The kernel supplies functions for querying the address of hardware registers and for performing memory mapping.

Testing Device Physical Addresses

The functions **badaddr()** and **wbadaddr()**, summarized in Table 9-9, are used to test a physical address to find out if it represents a usable device register.

Table 9-9 Functions to Test Physical Addresses

Function Name	Header Files	Can Sleep?	Purpose
badaddr(D3)	systm.h	N	Test physical address for input.
badaddr_val(D3)	systm.h	N	Test physical address for input and return the input value received.
wbadaddr(D3)	systm.h	N	Test physical address for output.
wbadaddr_val(D3)	systm.h	N	Test physical address for output of specific value.

The functions return a nonzero value when the address is bad, that is, unusable. Related functions for the VME bus are covered in Chapter 14.

You normally use these functions in the *pfx***init()** entry point to verify that an expected device is in fact present. The functions can also be useful in the *pfx***edtinit()** entry point. However, that entry point is only called from a VECTOR statement, and the VECTOR statement can contain a PROBE argument that tests for valid hardware.

Note: These functions must not be called in an interrupt handler. Verify device addresses in the upper-half code, during initialization.

Managing Mapped Memory

The *pfx***map()** and *pfx***unmap()** entry points receive a *vhandl_t* object that describes the region of user process space to be mapped. The functions summarized in Table 9-10 are used to manipulate that object.

Table 9-10 Functions to Manipulate a vhandl_t Object

Function Name	Header Files	Can Sleep?	Purpose
v_getaddr(D3)	region.h & types.h	N	Get the user virtual address associated with a *vhandl_t*.
v_gethandle(D3)	region.h & types.h	N	Get a unique identifier associated with a *vhandl_t*.
v_getlen(D3)	region.h & types.h	N	Get the length of user address space associated with a *vhandl_t*.
v_mapphys(D3)	region.h & types.h	N	Map kernel address space into user address space.

The **v_mapphys()** function actually performs a mapping between a kernel address and a segment described by a *vhandl_t* (see "Entry Point map()" on page 159).

The **v_getaddr()** function has hardly any use except for logging and debugging. The address in user space is normally undefined and unusable when the *pfx***map()** entry point is called, and mapped to kernel space when *pfx***unmap()** is called. The driver has no practical use for this value.

The **v_getlen()** function is useful only in the *pfx***unmap()** entry point—the *pfx***map()** entry point receives a length argument specifying the desired region size.

The **v_gethandle()** function returns a number that is unique to this mapping (actually, the address of a page table entry). You use this as a key to identify multiple mappings, so that the *pfx***unmap()** entry point can properly clean up.

Caution: Be careful when mapping device registers to a user process. Memory protection is available only on page boundaries, so configure the addresses of I/O cards so that each device is on a separate page or pages. When multiple devices are on the same page, a user process that maps one device can access all on that page. This can cause system security problems or other problems that are hard to diagnose.

Working With Page and Sector Units

In a 32-bit kernel, the page size for memory and I/O is 4 KB. In a 64-bit kernel, the memory page size is 16 KB, but because of hardware constraints such as the 4 KB span of DMA mapping registers in the Challenge and Onyx systems, a 4 KB page is used for I/O operations.

The header files *sys/immu.h* and *sys/sysmacros.h* contain constants and macros for working with page units. Some of the most useful are listed below:

NBPP	Number of bytes in a virtual memory page.
NBPSCTR	Number of bytes (512) in a standard disk "sector."
IO_NBPP	Number of bytes in an I/O page.
IO_PNUMSHFT	Number of bits to right-shift an address to get the I/O page number.
IO_POFFMASK	Mask to extract the I/O-page-offset value from an address.
btod**()**	Return number of 512-byte "sectors" in a byte count (rounded up)
btop(*x*)	Return number of I/O pages in a byte count (truncated)
io_pnum(*x*)	Return the I/O page number from an address *x*.
io_poff(*x*)	Return the I/O page offset from an address *x*.
io_numpages(*addr*, *len*)	Return the number of I/O pages that span a given address for a length.
io_ctob(*x*)	Return number of bytes in *x* I/O pages (rounded up).
io_ctobt(*x*)	Return number of bytes in *x* I/O pages (truncated).

The functions summarized in Table 9-11 are also provided as functions.

Table 9-11 Functions to Convert Bytes to Sectors or Pages

Function Name	Header Files	Can Sleep?	Purpose
btop(D3)	ddi.h	N	Return number of I/O pages in a byte count (truncate).
btopr(D3)	ddi.h	N	Return number of I/O pages in a byte count (round up).
ptob(D3)	ddi.h	N	Convert size in I/O pages to size in bytes.

Using these functions and macros, you can make your driver independent of the size of pages. When examining an existing driver, be alert for any assumption that a virtual memory page has a particular size, or that an I/O page is the same size as a memory page, and convert the code to use portable functions and macros.

Setting Up a DMA Transfer

There are two issues in preparing a DMA transfer:

- calculating physical addresses of the memory targets to be programmed into the device registers
- ensuring cache coherency in a uniprocessor

The functions you use to derive target addresses are different for different bus adapters and are discussed in the following chapters:

- The functions to set up DMA from a VME device are covered in Chapter 14, “Services for VME Drivers.”
- The functions to set up DMA from a SCSI device are covered in Chapter 15, “SCSI Device Drivers.”
- The functions to set up DMA from an EISA device are covered in Chapter 17, “EISA Device Drivers.”
- The functions to set up DMA from a GIO device are covered in Chapter 18, “GIO Device Drivers.”

Note: In addition, when designing a device driver for use in a Challenge or Onyx system, be sure to read the caution in Appendix B, “Challenge DMA with Multiple IO4 Boards.”

Converting Physical Addresses

General functions for calculating physical addresses are summarized in Table 9-12.

Table 9-12 Functions Related to Physical Memory

Function Name	Header Files	Can Sleep?	Purpose
kvtophys(D3)	ddi.h	N	Get physical address of kernel data
sgset(D3)	ddi.h & sg.h	N	Get physical addresses of a series of pages for simulated scatter/gather.

Managing Buffer Virtual Addresses

Functions to manipulate buffer page mappings are summarized in Table 9-13.

Table 9-13 Functions to Map Buffer Pages

Function Name	Header Files	Can Sleep?	Purpose
bp_mapin(D3)	buf.h	Y	Map buffer pages into kernel virtual address space.
bp_mapout(D3)	buf.h	N	Release mapping of buffer pages.
bptophys(D3)	ddi.h	N	Get physical address of buffer data.
clrbuf(D3)	buf.h	N	Clear the memory described by a mapped-in *buf_t*.
getnextpg(D3)	buf.h	N	Return *pfdat* structure for next page.
pptophys(D3)	buf.h	N	Return the physical address of a page described by a *pfdat* structure.

When a *pfx***strategy()** routine receives a *buf_t* that is not mapped into memory (see "Buffer Location and b_flags" on page 184), it must make sure that the pages of the buffer space are in memory, and it must obtain valid kernel virtual addresses to describe the pages. The simplest way is to apply the **bp_mapin()** function to the *buf_t*. This function allocates a contiguous range of page table entries in the kernel address space to describe the buffer, creating a mapping of the buffer pages to a contiguous range of kernel virtual addresses. It sets the virtual address of the first data byte in *b_un.b_addr*, and sets the flags so that BP_ISMAPPED**()** returns true—thus converting an unmapped buffer to a mapped case.

When the device does not handle scatter/gather DMA there is a disadvantage to using **bp_mapin()**. Without scatter/gather, each page's worth of data must be set up as a unique I/O operation. There is no need to have all of a possibly large buffer locked into memory for the whole time. Using **getnextpg()**, a driver can step through the *pfdat* structures that describe the successive pages of the buffer, calling **pptophys()** to get the physical address of the page frame.

Managing Memory for Cache Coherency

Some kernel functions used for ensuring cache coherency are summarized in Table 9-14.

Table 9-14 Functions Related to Cache Coherency

Function Name	Header Files	Can Sleep?	Purpose
dki_dcache_inval(D3)	systm.h & types.h	N	Invalidate the data cache for a given range of virtual addresses.
dki_dcache_wb(D3)	systm.h & types.h	N	Write back the data cache for a given range of virtual addresses.
dki_dcache_wbinval(D3)	systm.h & types.h	N	Write back and invalidate the data cache for a given range of virtual addresses.
flushbus(D3)	systm.h & types.h	?	Make sure contents of the write buffer are flushed to the system bus

The functions for cache invalidation are essential when doing DMA on a uniprocessor. They cost very little to use in a multiprocessor, so it does no harm to call them in every system. You call them as follows:

- Call **dki_dcache_inval()** prior to doing DMA input. This ensures that when you refer to the received data, it will be loaded from real memory.
- Call **dki_dcache_wb()** prior to doing DMA output. This ensures that the latest contents of cache memory are in system memory for the device to load.
- Call **dki_dcache_wbinval()** prior to a device operation that samples memory and then stores new data.

The **flushbus()** function is needed because in some systems the hardware collects output data and writes it to the bus in blocks. When you write a small amount of data to a device through PIO, delay, then write again, the writes could be batched and sent to the device in quick succession. Use **flushbus()** after PIO output when it is followed by PIO input from the same device. Use it also between any two PIO outputs when the device is supposed to see a delay between outputs.

DMA Buffer Alignment

In some systems, the buffers used for DMA must be aligned on a boundary the size of a *cache line* in the current CPU. Although not all system architectures require cache alignment, it does no harm to use cache-aligned buffers in all cases. The size of a cache line varies among CPU models, but if you obtain a DMA buffer using the KMEM_CACHEALIGN flag of **kmem_alloc()**, the buffer is properly aligned. The buffer returned by **geteblk()** (see “Allocating buf_t Objects and Buffers” on page 190) is cache-aligned.

Why is cache alignment necessary? Suppose you have a variable, *X*, adjacent to a buffer you are going to use for DMA write. If you invalidate the buffer prior to the DMA write, but then reference the variable *X*, the resulting cache miss brings part of the buffer back into the cache. When the DMA write completes, the cache is stale with respect to memory. If, however, you invalidate the cache after the DMA write completes, you destroy the value of the variable *X*.

Maximum DMA Transfer Size

The maximum size for a single DMA transfer is set by the system tuning variable *maxdmasz*, settable with the *systune* command (see the systune(1) reference page). A single I/O operation larger than this produces the error ENOMEM.

The unit of measure for *maxdmasz* is the page, which varies with the kernel. Under IRIX 6.2, a 32-bit kernel uses 4 KB pages while a 64-bit kernel uses 16 KB pages. In both systems, *maxdmasz* is shipped with the value 1024 decimal, equivalent to 4 MB in a 32-bit kernel and 16 MB in a 64-bit kernel.

In Challenge and Onyx systems, *maxdmasz* can be set as high as 64 MB. However, it is not usually possible to allocate a DMA map for a single transfer that large—see “Mapping DMA Addresses” on page 330.

User Process Administration

The kernel supplies a small group of functions, summarized in Table 9-15, that help a driver upper-half routine learn about the current user process.

Table 9-15 Functions for User Process Management

Function Name	Header Files	Can Sleep?	Purpose
drv_getparm(D3)	ddi.h	N	Retrieve kernel state information.
drv_priv(D3)	ddi.h	N	Test for privileged user.
drv_setparm(D3)	ddi.h	N	Set kernel state information.
proc_ref(D3)	ddi.h	N	Obtain a reference to a process for signaling.
proc_signal(D3)	ddi.h & signal.h	N	Send a signal to a process.
proc_unref(D3)	ddi.h	N	Release a reference to a process.

Use **drv_getparm()** to retrieve certain miscellaneous bits of information including the process ID of the current process. In a character device driver, the current process is the user process that caused entry to the driver, for example by calling the **open()**, **ioctl()**, or **read()** system functions. In a block device driver, the current process has no direct relationship to any particular user; it is usually a daemon process of some kind.

The **drv_setparm()** function is primarily of use to terminal drivers.

The **drv_priv()** function tests a *cred_t* object to see if it represents a privileged user. A *cred_t* object is passed in to several driver entry points, and the address of the current one can be retrieved **drv_getparm()**.

Sending a Process Signal

In traditional UNIX kernels, a device driver identified the current user process by the address of the *proc_t* structure that the kernel uses to represent a process. Direct use of the *proc_t* is no longer supported by IRIX. The reason is that the contents of the *proc_t* change from release to release, and also differ between 64-bit and 32-bit kernels.

The most common use of the *proc_t* by a driver was to send a signal to the process. This capability is still supported. To do it, take three steps:

1. Call **proc_ref()** to get a process handle, a number unique to the current process. The returned value must be treated as an arbitrary number (in some releases of IRIX it was the *proc_t* address, but this is not the defined behavior of the function.)
2. Use the process handle as an argument to **proc_signal()**, sending the signal to the process.
3. Release the process handle by calling **proc_unref()**.

The third step is important. In order to keep the process handle valid, IRIX retains information about the process to which it is related. However, that process could terminate (possibly as a result of the signal the driver sends) but until the driver announces that it is done with the handle, the kernel must try to retain process information.

It is especially important to release a process handles before unloading a loadable driver (see "Entry Point unload()" on page 167).

Waiting and Mutual Exclusion

The kernel supplies a rich variety of functions for waiting and for mutual exclusion. In order to use these features well, you must understand the different purposes for which they are designed. In particular, you must clearly understand the distinction between *waiting* and *mutual exclusion* (or locking).

Note: These waiting and mutual exclusion functions have been expanded significantly in IRIX release 6.2.

Mutual Exclusion Compared to Waiting

Mutual exclusion allows one entity to have exclusive use of a global resource, temporarily denying use of the resource to other entities. When software is well-designed, mutual exclusion normally does not require waiting—the resource is normally free when it is requested. A driver that calls a mutual exclusion function *expects to proceed* without delay—although there is a chance that the resource is in use, and the driver will have to wait.

The kernel offers an array of functions for mutual exclusion, and the choice among them can be critical to performance. The functions are reviewed in the following topics:

- "Basic Locks" on page 205 covers basic locks and mutex locks, the best locks for multiprocessor use.
- "Long-Term Locks" on page 207 covers sleep locks, which can be held for longer periods.
- "Reader/Writer Locks" on page 211 covers a class of locks that allow multiple, concurrent, read-only access to resources that are infrequently changed.
- "Priority Level Functions" on page 213 covers the traditional UNIX method of mutual exclusion, the **splhi()** and **splx()** functions, which have many disadvantages.

Waiting allows a driver to coordinate its actions with a specific event or action that occurs asynchronously. A driver can wait for a specified amount of time to pass, wait for an I/O action to complete, and so on. Therefore, a driver that calls a waiting function *expects to wait* for something to happen—although there is a chance that the expected event has already happened, and the driver will be able to continue at once.

The kernel offers several functions that allow you to wait for specific events; and also offers functions for general synchronization. These are covered in the following topics:

- "Waiting for Time to Pass" on page 214 covers timer-related functions.
- "Waiting for Memory to Become Available" on page 216 covers memory allocation waits.
- "Waiting for Block I/O to Complete" on page 217 covers waits used in the *pfx***strategy()** entry point.
- "Waiting for a General Event" on page 218 covers the general-purpose functions that you can adapt to any synchronization problem.

The most general facility, the semaphore, can be used for synchronization and for locking. This topic is covered under "Semaphores" on page 222.

Basic Locks

IRIX supports basic locks using functions compatible with SVR4. These functions are summarized in Table 9-16.

Table 9-16 Functions for Basic Locks

Function Name	**Header Files**	**Can Sleep?**	**Purpose**
LOCK(D3)	ksynch.h & types.h	Y	Acquire a basic lock, waiting if necessary.
LOCK_ALLOC(D3)	ksynch.h,kmem.h & types.h	Y	Allocate and initialize a basic lock.
LOCK_DEALLOC(D3)	ksynch.h & types.h	N	Deallocate an instance of a basic lock.
LOCK_INIT(D3)	ksynch.h & types.h	N	Initialize a basic lock that was allocated statically, or reinitialize an allocated lock.
LOCK_DESTROY(D3)	ksynch.h & types.h	N	Uninitialize a basic lock that was allocated statically.

Table 9-16 (continued) Functions for Basic Locks

Function Name	Header Files	Can Sleep?	Purpose
TRYLOCK(D3)	types.h & ksynch.h	N	Try to acquire a basic lock, returning a code if the lock is not currently free.
UNLOCK(D3)	types.h & ksynch.h	N	Release a basic lock.

Basic locks are objects of type *lock_t*. Although functions are provided for allocating and freeing them, a basic lock is a very small object. Locks are typically allocated as global variables or as fields of structures.

Call LOCK**()** to seize a lock and gain possession of the resource for which it stands. Release the lock with UNLOCK**()**. These functions are optimized for mutual exclusion in the available hardware, and may be implemented differently in uniprocessors and multiprocessors. However, the programming and binary interface is the same in all systems.

The code in Example 9-1 illustrates the use of LOCK and UNLOCK in implementing a simple last-in-first-out (LIFO) queueing package. In these functions, the time between locking a queue head and releasing it is only a few microseconds.

Example 9-1 LIFO Queue Using Basic Locks

```
typedef struct qitem {
   qitem *next; ...other fields...
} qitem_t;
typedef struct lifo {
   qitem *latest;
   lock_t grab;
} lifo_t;
void putlifo(lifo_t *q, qitem_t *i)
{
   int lockpl = LOCK(&q->grab,plhi);
   i->next = q->latest;
   q->latest = i;
   UNLOCK(&q->grab,lockpl);
}
qitem_t *poplifo(lifo_t *q)
{
   int lockpl = LOCK(&q->grab,plhi);
```

```
    qitem_t *ret = q->latest;
    q->latest = ret->next;
    UNLOCK(&q->grab,lockpl);
    return ret;
}
```

This is a typical use of basic locks: to ensure that for a brief period only one process in the system can update a queue. Basic locks are optimized for such uses, but in order to get optimal performance they are restricted to these uses. In particular, if you seize a basic lock and hold it over a function call that can sleep, the system can deadlock.

Long-Term Locks

Sometimes you need a lock that can be held for a longer period, over a call to a function that can sleep. IRIX provides three types of such locks: mutex locks, sleep locks, and reader-writer locks.

Using Mutex Locks

Mutex locks are designed for mutual exclusion (as the name suggests). The IRIX implementation of mutex locks is compatible with the *kmutex_t* lock type of SunOS™, but optimized for use in Silicon Graphics hardware systems. The mutex functions are summarized in Table 9-17.

Table 9-17 Functions for Mutex Locks

Function Name	Header Files	Can Sleep?	Purpose
MUTEX_ALLOC(D3)	types.h & kmem.h & ksynch.h	Y	Allocate and initialize a mutex lock.
MUTEX_INIT(D3)	types.h & ksynch.h	N	Initialize an existing mutex lock.
MUTEX_DESTROY(D3)	types.h & ksynch.h	N	Deinitialize a mutex lock.
MUTEX_DEALLOC(D3)	types.h & ksynch.h	N	Deinitialize and free a dynamically allocated mutex lock.

Table 9-17 (continued) Functions for Mutex Locks

Function Name	Header Files	Can Sleep?	Purpose
MUTEX_LOCK(D3)	types.h & kmem.h & ksynch.h	Y	Claim a mutex lock.
MUTEX_TRYLOCK(D3)	types.h & ksynch.h	N	Conditionally claim a mutex lock.
MUTEX_UNLOCK(D3)	types.h & ksynch.h	N	Release a mutex lock.
MUTEX_WAITQ(D3)	types.h & ksynch.h	N	Get the number of processes blocked by mutex lock.
MUTEX_ISLOCKED(D3)	types.h & ksynch.h	N	Test if a mutex lock is owned.
MUTEX_MINE(D3)	types.h & ksynch.h	N	Test if a mutex lock is owned by this process.

Although allocation and deallocation functions are supplied, a *mutex_t* type is a small object that is normally allocated as a static variable or as a field of a structure. The MUTEX_INIT() operation prepares a statically-allocated *mutex_t* for use.

Once initialized, a mutex lock is used to gain exclusive use of the resource with which you have associated it. The mutex lock has the following important advantages over a basic lock:

- The mutex lock can safely be held over a call to a function that sleeps.
- The mutex lock supports the inquiry functions MUTEX_WAITQ, MUTEX_ISLOCKED, and MUTEX_MINE.
- When a debugging kernel is used (see "Including Lock Metering in the Kernel Image" on page 246) a mutex lock can be instrumented to keep statistics of its use.

The mutex lock implementation provides *priority inheritance*. When a low-priority process owns a mutex lock and a high-priority process attempts to seize the lock and is blocked, the process holding the lock is temporarily given the higher priority of the blocked process. This hastens the time when the lock can be released, so that a low-priority process does not needlessly impede a higher-priority process.

In order to implement priority inheritance and retain high performance, the mutex lock is subject to the following restrictions:

- A mutex lock can be locked and unlocked only by an upper-half driver routine; that is, from code that has a process context. A mutex lock cannot be locked or unlocked in an interrupt routine.
- A mutex lock must be unlocked by the same process that locked it. It cannot be locked in one process identity and unlocked in another.

Because of these restrictions, a mutex lock can only be used to mediate between upper-half driver entry points. It is very effective for this purpose; you can use mutex locks to coordinate the use of global variables between upper-half entry points of a driver in a multiprocessor design.

When you need mutual exclusion between an upper-half entry point and the interrupt handler, use a basic lock. Resources that are shared with an interrupt handler should never be in use for more than a brief period. When your design requires a lock to be seized by one process and released in another, use a sleep lock or semaphore.

Using Sleep Locks

IRIX supports sleep lock functions that are compatible with SVR4. These functions are summarized in Table 9-18.

Table 9-18 Functions for Sleep Locks

Function Name	Header Files	Can Sleep?	Purpose
SLEEP_ALLOC(D3)	types.h & kmem.h & ksynch.h	Y	Allocate and initialize a sleep lock.
SLEEP_DEALLOC(D3)	types.h & ksynch.h	N	Deinitialize and deallocate a dynamically allocated sleep lock.
SLEEP_INIT(D3)	types.h & ksynch.h	N	Initialize an existing sleep lock.
SLEEP_DESTROY	types.h & ksynch.h	N	Deinitialize a sleep lock.

Table 9-18 (continued) Functions for Sleep Locks

Function Name	Header Files	Can Sleep?	Purpose
SLEEP_LOCK(D3)	types.h & ksynch.h & param.h	Y	Acquire a sleep lock, waiting if necessary until the lock is free.
SLEEP_LOCKAVAIL(D3)	types.h & ksynch.h	N	Query whether a sleep lock is available.
SLEEP_LOCK_SIG(D3)	types.h & ksynch.h & param.h	Y	Acquire a sleep lock, waiting if necessary until the lock is free or a signal is received.
SLEEP_TRYLOCK(D3)	types.h & ksynch.h	N	Try to acquire a sleep lock, returning a code if it is not free.
SLEEP_UNLOCK(D3)	types.h & ksynch.h	N	Release a sleep lock.

Although allocation and deallocation functions are supplied, a *sleep_t* type is a small object that is normally allocated as a static variable or as a field of a structure. The SLEEP_INIT() operation prepares a statically-allocated *sleep_t* for use. (In IRIX 6.2, a *sleep_t* is identical to a *sema_t*, but this situation could change in a future release.)

A sleep lock is similar to a mutex lock in that it is used for mutual exclusion between processes, and can be held across a function call that sleeps. A sleep lock does not have either the advantages or the restrictions of a mutex lock:

- A sleep lock can be seized by one process and released by another.
- A sleep lock can be set in an upper-half entry point and released in an interrupt routine.
- A sleep lock does not provide priority inheritance. When a low-priority process holds a sleep lock, a higher-priority process can be blocked, causing a *priority inversion*.
- A sleep lock does not support the instrumentation or the query functions supported for mutex locks.

Reader/Writer Locks

Reader/writer locks are similar to sleep locks in that they are designed for mutually exclusive control of resources for relatively long periods of time. However, Reader/Writer locks are optimized for the case in which the resource is often used by processes that only interrogate it (readers), and only rarely used by processes that modify it (writers).

Reader/writer locks compatible with SVR4 are introduced in IRIX 6.2. The functions are summarized in Table 9-19.

Table 9-19 Functions for Reader/Writer Locks

Function Name	Header Files	Can Sleep?	Purpose
RW_ALLOC(D3)	types.h & kmem.h & ksynch.h	Y	Allocate and initialize a reader/writer lock.
RW_DEALLOC(D3)	types.h & ksynch.h	N	Deallocate a reader/writer lock.
RW_INIT(D3)	types.h & ksynch.h	N	Initialize an existing reader/writer lock.
RW_DESTROY(D3)	types.h & ksynch.h	N	Deinitialize an existing reader/writer lock.
RW_RDLOCK(D3)	types.h & ksynch.h & param.h	Y	Acquire a reader/writer lock as reader, waiting if necessary.
RW_TRYRDLOCK(D3)	types.h & ksynch.h	N	Try to acquire a reader/writer lock as reader, returning a code if it is not free.
RW_TRYWRLOCK(D3)	types.h & ksynch.h	N	Try to acquire a reader/writer lock as writer, returning a code if it is not free.

Table 9-19 (continued) Functions for Reader/Writer Locks

Function Name	Header Files	Can Sleep?	Purpose
RW_UNLOCK(D3)	types.h & ksynch.h	N	Release a reader/writer lock as reader or writer.
RW_WRLOCK(D3)	types.h & ksynch.h & param.h	Y	Acquire a reader/writer lock as writer, waiting if necessary.

Although allocation and deallocation functions are supplied, a *mrlock_t* type is a small object that is normally allocated as a static variable or as a field of a structure. The RW_INIT() operation prepares a statically-allocated *mrlock_t* for use.

A process that intends to modify a resource uses RW_WRLOCK to claim it. This process waits until the resource is not in use by any process, then it gains exclusive access. Only one process is allowed to hold a reader/writer lock as a writer. All other processes, readers or writers, wait until the writer releases the lock.

A process that intends only to interrogate a resource uses RW_RDLOCK to gain access. If a writer holds the lock, the process waits. When the lock is free, or is held only by other readers, the process continues. More than one reader can hold a reader/writer lock at one time. It is also valid for a reader to "double-trip" a reader/writer lock; that is, claim it two or more times. The reader must release the lock as many times as it claimed the lock.

A reader/writer lock serves the same basic purpose as a sleep lock, but it is more efficient in a multiprocessor when there are frequent, read-only uses of a resource.

Priority Level Functions

In traditional UNIX systems, one set of functions served all purposes of synchronization and locking: the set-priority-level, or **spl**, functions. These functions are still available in IRIX, and are summarized in Table 9-20.

Table 9-20 Functions to Set Interrupt Levels

Function Name	Header Files	Can Sleep?	Purpose
splbase(D3)	ddi.h	N	Block no interrupts.
spltimeout(D3)	ddi.h	N	Block only timeout interrupts.
spldisk(D3)	ddi.h	N	Block disk interrupts.
splstr(D3)	ddi.h	N	Block STREAMS interrupts.
spltty(D3)	ddi.h	N	Block disk, VME, serial interrupts.
splhi(D3)	ddi.h	N	Block all I/O interrupts.
spl0(D3)	ddi.h	N	Same as **splbase()**.
splx(D3)	ddi.h	N	Restore previous interrupt level.

These functions are commonly found in device drivers being ported from uniprocessors. Such drivers rely on the use of **splhi()** to gain exclusive use of a global resource.

The **spl** functions are supported by IRIX and they are effective in a uniprocessor driver. However, in a multiprocessor, the functions affect only the interrupt handling of the current CPU. Other CPUs in the system continue to handle interrupts, including interrupts initiated by the driver that called **splhi()**.

A driver that is not multiprocessor-aware (one that does not have D_MP in its *pfx***devflag** constant; see “Driver Flag Constant” on page 140) runs only in CPU 0 of a multiprocessor, so in this case the **spl** functions are still effective. Since they set the interrupt level on CPU 0 where the driver runs, and since the driver’s interrupts can only be handled on CPU 0, the use of **splhi()** gives the driver exclusive use of its resources.

A driver that is multiprocessor-aware uses basic locks, synchronization variables, and other tools to control access to resources, and never uses an **spl** function. This improves performance in a multiprocessor, does not harm performance in a uniprocessor, and reduces the latency of all interrupts.

Waiting for Time to Pass

The kernel offers functions for timed delays, as summarized in Table 9-21.

Table 9-21 Functions for Timed Delays

Function Name	Header Files	Can Sleep?	Purpose
delay(D3)	ddi.h	Y	Delay for a specified number of clock ticks.
drv_hztousec(D3)	ddi.h	N	Convert clock ticks to microseconds
drv_usectohz(D3)	ddi.h	N	Convert microseconds to clock ticks.
drv_usecwait(D3)	ddi.h	N	Busy-wait for a specified interval.
dtimeout(D3)	ddi.h & ksynch.h	N	Schedule a function execute on a specified processor after a specified length of time.
itimeout(D3)	ddi.h & ksynch.h	N	Schedule a function to be executed after a specified number of clock ticks.
fast_itimeout(D3)	ddi.h & ksynch.h	N	Same as **itimeout()** but takes an interval in "fast ticks."
fasthzto(D3)	types.h & time.h	N	Returns the value of a *struct timeval* as a count of "fast ticks."
timeout(D3)	ddi.h & ksynch.h	N	Schedule a function to be executed after a specified number of clock ticks.
untimeout(D3)	ddi.h	N	Cancel a previous itimeout or fast_itimeout request.
untimeout_func(D3)	ddi.h	N	Cancel a previous itimeout or fast_itimeout request by function name.

Time Units

The basic time unit is the “tick.” Its value can differ between hardware platforms and between versions of IRIX. The **drvhztousec()** and **drvusectohz()** functions convert between ticks and microseconds in the current system. Use them in order to schedule a delay in a portable mannter. (However, the timer function precision is the tick, not the microsecond.)

The “fast tick” is a fraction of a tick. Like the tick, the fast tick’s value can differ between systems. Use **fasthzto()** to convert from microseconds to fast ticks.

Timer Support

Timer support is based on the idea of a “callback” function. You specify the following to **dtimeout()**, **itimeout()**, **timeout()** or **fast_itimeout()**:

- an interval in clock ticks or fast ticks
- a function to be called at the expiration of the interval
- one or more arguments to be passed to the function
- a priority (interrupt) level at which the function should run

After a delay of at least the length requested, the function is called. The function is entered asynchronously. On a uniprocessor, it can interrupt execution of an upper-half routine. On a multiprocessor, it can execute concurrently with an upper-half routine or with an interrupt handler. You should not rely on the priority level of the function for mutual exclusion (see “Priority Level Functions” on page 213 for an explanation).

The difference between **itimeout()** and **timeout()** is that the latter takes no argument values to be passed to the function when it is called. In order to get a repeated series of timer events, start a new timeout from the callback function.

The **untimeout()** and **untimeout_func()** functions cancel a pending timeout. In a loadable driver that has an *pfx***unload()** entry point, cancel any pending timeouts before unloading.

The STREAMS_TIMOUT macro supplies similar timeout capability for a STREAMS driver (see “Special Considerations for Multiprocessing” on page 547).

Short-Term Delay Support

In rare circumstances, a driver needs to pause briefly between two hardware operations. For example, the Silicon Graphics support for external interrupts in the Challenge and Onyx computers sometimes needs to set a high output level, wait for a brief, precise interval, then set a low output level.

The **drv_usecwait()** function supports this type of very short, precisely-timed delay. It "spins" for a specified number of microseconds, then returns to the caller. The CPU does nothing else during this period, so clearly a delay of more than a few microseconds can interfere with other work. Furthermore, if interrupts are disabled during the wait, the response to another interrupt is delayed also—the delay contributes directly to the "latency" of interrupt handling.

Waiting for Memory to Become Available

Whenever you request memory of any kind, you must allow for the possibility that the memory will not be available. When you allocate memory in bulk (see "General-Purpose Allocation" on page 187) using **kmem_alloc()** you have the option of receiving a null response, or of waiting for the memory to be available.

When you request memory for specific object types (see "Allocating Objects of Specific Kinds" on page 189) there is usually no choice; the functions sleep until they can acquire an object of the requested type.

Within a STREAMS driver you have the ability to schedule a callback function to be entered when memory for a message buffer becomes available (see the bufcall(D3) reference page).

Waiting for Block I/O to Complete

The *pfx***strategy()** routine initiates the I/O operation to fill a buffer based on a *buf_t* structure. Then it has to wait for the I/O to complete. The functions for managing this synchronization are summarized in Table 9-22.

Table 9-22 Functions for Synchronizing Block I/O

Function Name	Header Files	Can Sleep?	Purpose
biodone(D3)	ddi.h	N	Release buffer after I/O and wake up waiting process.
bioerror(D3)	ddi.h	N	Manipulate error fields in a *buf_t*.
biowait(D3)	ddi.h	Y	Suspend process pending completion of I/O.
geterror(D3)	ddi.h	N	Retrieve error number from a *buf_t*.
physiock(D3)	ddi.h	Y	Validate a raw I/O request and pass to a strategy function.
uiophysio(D3)	ddi.h	Y	Validate a raw I/O request and pass to a strategy function.
undma(D3)	ddi.h	?	Unlock physical memory after I/O complete
userdma(D3)	ddi.h	?	Lock physical memory in user space.small number of

How the strategy() Entry Point Is Called

The *pfx***strategy()** entry point is called directly from the filesystem or virtual memory management, or it can be called indirectly from a *pfx***read()** or *pfx***write()** entry point (see "Calling Entry Point strategy() From Entry Point read() or write()" on page 153).

Strategies of the strategy() Entry Point

Typically the *pfx***strategy()** routine must interact with its interrupt handler. The *pfx***strategy()** routine can be designed in either of two ways, synchronous or asynchronous.

The synchronous *pfx***strategy()** routine initiates every I/O operation. Its interrupt handler is responsible only for detecting and signalling the completion of one I/O. The *pfx***strategy()** routine proceeds as follows:

1. Lock the data buffer in memory using **userdma()**.
2. Place the address of the *buf_t* where the *pfx***intr()** entry point can find it.
3. Program the device (see “Setting Up a DMA Transfer” on page 198) and initiate the I/O activity.
4. Call **biowait()**.

When the interrupt handler is entered, the handler uses **bioerror()** if necessary, and **biodone()** to signal the completion of the I/O. Then it exits. The strategy code, which is waiting in the call to **biowait()**, regains control following the call to **biodone()**, and can use **geterror()** to check the results.

The asynchronous *pfx***strategy()** routine only initiates the first I/O operation of a series, and never waits. It proceeds as follows:

1. Lock the data buffer in memory using **userdma()**.
2. Append the address of the *buf_t* to a queue shared with the interrupt handler.
3. If the queue was empty, no I/O is in progress. Call a subroutine that programs the device and initiates the I/O.
4. Return to the caller. The caller (a filesystem or paging system or **uiophysio()**) waits using **biowait()**.

When the interrupt occurs, the handler proceeds as follows:

1. The first queued *buf_t* has completed. Remove it from the queue.
2. Apply **bioerror()** if necessary, and **biodone()** to the *buf_t*. This releases the caller of the strategy routine from **biowait()**.
3. If any operations remain in the queue, call a subroutine to program and initiate the next one.

Waiting for a General Event

There are causes for synchronization other than time, block I/O, and memory allocation. For example, there is no defined interface comparable to **biowait()**/**biodone()** to mediate between an interrupt handler and the *pfx***read()** or *pfx***write()** entry points. You must design a mechanism of your own, using either a synchronization variable or the **sleep()**/**wakeup()** function pair.

Using sleep() and wakeup()

The **sleep()** and **wakeup()** function pair are the simplest, oldest, and least efficient of the general synchronization mechanisms. They are summarized in Table 9-23.

Table 9-23 Functions for Synchronization: sleep/wakeup

Function Name	Header Files	Can Sleep?	Purpose
sleep(D3)	ddi.h & param.h	Y	Suspend execution pending an event.
wakeup(D3)	ddi.h	N	Waken a process waiting for an event.

Used carefully, these functions are suitable for simple character device drivers. However, when you are writing new code or converting a driver to multiprocessing you should avoid them and use synchronization variables instead (see "Using Synchronization Variables" on page 220).

The basic concept is that the upper-layer routine calls **sleep**(*n*) in order to wait for an event that is keyed to an arbitrary address *n*. Typically *n* is a pointer to a data structure related to an I/O operation. The interrupt handler executes **wakeup**(*n*) to cause the sleeping process to resume execution.

The main reason to avoid **sleep()** is that, in a multiprocessor system, it is hard to ensure that sleeping always begins before **wakeup()** is called. The usual intended sequence of events is as follows:

1. Upper-half routine initiates a device operation that will lead to an interrupt.
2. Upper-half routine executes **sleep**(*n*).
3. Interrupt occurs, and handler executes **wakeup**(*n*).

In a multiprocessor-aware driver (one with D_MP in its *pfx***devflag** constant; see "Driver Flag Constant" on page 140), there is a small chance that the interrupt can occur, calling **wakeup**(*n*), before the **sleep**(*n*) call has been completed. Because **sleep()** has not been called, the **wakeup()** is lost. When the **sleep()** call completes, the process sleeps forever. Synchronization variables are designed to handle this case.

Using Synchronization Variables

Synchronization variables, a feature of UNIX SVR4, are supported by IRIX beginning with release 6.2. These functions are summarized in Table 9-24.

Table 9-24 Functions for Synchronization: Synchronization Variables

Function Name	Header Files	Can Sleep?	Purpose
SV_ALLOC(D3)	types.h & sema.h	Y	Allocate and initialize a synchronization variable.
SV_DEALLOC(D3)	types.h & sema.h	N	Deinitialize and deallocate a synchronization variable.
SV_INIT	types.h & sema.h	N	Initialize an existing synchronization variable.
SV_DESTROY	types.h & sema.h		Deinitialize a synchronization variable.
SV_BROADCAST(D3)	types.h & sema.h	N	Wake all processes sleeping on a synchronization variable.
SV_SIGNAL(D3)	types.h & sema.h	N	Wake one process sleeping on a synchronization variable.
SV_WAIT(D3)	types.h & sema.h	Y	Sleep until a synchronization variable is signalled.
SV_WAIT_SIG(D3)	types.h & sema.h	Y	Sleep until a synchronization variable is signalled or a signal is received.

A synchronization variable is a memory object of type *sv_t*, representing the occurrence of an event. You can allocate objects of this type dynamically, or declare them as static variables or as fields of structures.

One or more processes may wait for an event using SV_WAIT(). An interrupt handler or timer callback function can signal the occurrence of an event using SV_SIGNAL (to wake up only one waiting process) or SV_BROADCAST (to wake up all of them).

SV_WAIT is specifically designed to handle the difficult case that arises when the driver needs to initiate an I/O operation and then sleep, and do these things in such a way that it always begins to sleep before the SV_SIGNAL can possibly be issued. The procedure is done as follows:

1. The driver seizes a basic lock (see "Basic Locks" on page 205) or a mutex lock (see "Using Mutex Locks" on page 207)that is also used by the interrupt handler.

   A LOCK() call returns an integer that is needed later.
2. The driver initiates an I/O operation that can lead to an interrupt.
3. The driver calls SV_WAIT, passing the lock it holds and an integer, either the value returned by LOCK() or a zero if the lock is a mutex lock.
4. In one indivisible operation, SV_WAIT releases the lock and begins waiting on the synchronization variable.
5. The interrupt handler or other process is entered, and seizes the lock.

   This step ensures that, if the interrupt handler or other process is entered preceding the SV_WAIT call, it will not proceed until SV_WAIT has completed.
6. The interrupt handler or other process does its work and calls SV_SIGNAL to release the waiting driver.

This process is sketched in Example 9-2.

Example 9-2 Skeleton Code for Use of SV_WAIT

```
lock_t seize_it;
sv_t wait_on_it;
initiator(...)
{
   int lock_cookie;
   for( as often as necessary )
   {
      lock_cookie = LOCK(&seize_it,PL_ZERO);
      [do something that causes a later interrupt]
      SV_WAIT(&wait_on_it, 0, &seize_it, lock_cookie);
      [interrupt has been handled]
   }
}

void handler(...)
{
   int lock_cookie = LOCK(&seize_it,PL_ZERO);
   [handle the interrupt]
   SV_SIGNAL(&seize_it);
   UNLOCK(&seize_it);
}
```

If it is necessary to use a semaphore as the lock, the header file sys/sema.h declares versions of SV_WAIT that accept a semaphore and a synchronization variable. The combination of a mutual exclusion object and a synchronization variable ensures that even in a multiprocessor, the interrupt handler cannot exit before the driver has entered a predictable wait state.

Tip: When a debugging kernel is used, you can display statistics about the use of a given synchronization variable. See "Including Lock Metering in the Kernel Image" on page 246.

Semaphores

The *semaphore* is a generalized tool that can be used for both mutual exclusion and for waiting. The IRIX kernel support for semaphores is summarized in Table 9-25.

Table 9-25 Functions for Semaphores

Function Name	Header Files	Can Sleep?	Purpose
cpsema(D3)	sema.h & types.h	N	Conditionally perform a "P" or wait semaphore operation.
cvsema(D3)	sema.h & types.h	N	Conditionally perform a "V" or release semaphore operation.
freesema(D3)	sema.h & types.h	N	Free the resources associated with a semaphore.
initnsema(D3)	sema.h & types.h	N	Initialize a semaphore to a given value.
initnsema_mutex(D3)	sema.h & types.h	N	Initialize a semaphore to a value of 1.
psema(D3)	sema.h & types.h & param.h	Y	Perform a "P" or wait semaphore operation.

Table 9-25 (continued) Functions for Semaphores

Function Name	Header Files	Can Sleep?	Purpose
valusema(D3)	sema.h & types.h	N	Return the value associated with a semaphore.
vsema(D3)	sema.h & types.h	N	Perform a "V" or signal semaphore operation.

Conceptually, a semaphore contains an integer. The "P" operation claims the semaphore, decrementing its count by 1 (mnemonic: dePlete). If the count is 0 or less, the process waits until the count is greater than 0 before it decrements the semaphore and returns.

The "V" operation increments the semaphore count (mnemonic: reViVe) and wakens any process that is waiting.

Tip: When a debugging kernel is used, you can display statistics about the use of a given semaphore. See "Including Lock Metering in the Kernel Image" on page 246.

Note: In releases before IRIX 6.2, **initnsema_mutex()** was used to initialize a semaphore in a special way that got the performance of a basic lock in a multiprocessor. Since IRIX 6.2, this function is simply a macro that initializes the semaphore to a count of 1.

Using a Semaphore for Mutual Exclusion

To use a semaphore for locking, initialize it to 1. (This reflects the idea that a process calling a locking function expects to continue.) When you require exclusive use of the associated resource, call **psema()**. Typically this finds a semaphore count of 1, reduces it to 0, and returns.

When you are finished with the resource, call **vsema()** to increment the semaphore count, and release any process that is blocked in a **psema()** call for the same semaphore.

For locking, a semaphore is comparable to a sleep lock. In some systems, the performance of semaphore operations may not be as good as the performance of a mutex lock. In other systems, mutex locks may be implemented using semaphores.

Using a Semaphore for Waiting

To use a semaphore for waiting, initialize it to 0. Then call **psema()**. Because the semaphore count is 0, the process waits. When the desired event occurs, typically in the interrupt handler, call **vsema()** to release the waiting process.

This synchronization method is as reliable as a synchronization variable, but it has slightly different behavior. When a synchronization variable is used correctly (see "Using Synchronization Variables" on page 220), if the interrupt handler is entered before the SV_WAIT call completes, the interrupt handler waits on a LOCK call.

When a semaphore is used, if the interrupt handler is entered before the **psema()** call completes, the **vsema()** operation is done immediately and the interrupt handler continues without waiting. The fact that **vsema()** was called is stored as a count within the semaphore, where **psema()** will find it. Because the semaphore can contain this state information, the interrupt handler does not have to be synchronized in time using a lock.

Note: In releases before IRIX 6.2, the **vpsema()** function was used in a way similar to synchronization variables are used: to release one semaphore and wait on another in an atomic operation. This function is no longer supported; replace it with syncronization variable.

Chapter 10

Building and Installing a Driver

After a kernel-level driver has been designed and coded, it must be compiled, linked, and installed. The topics in this chapter describe the major steps of this process, as follows:

- "Defining Device Numbers" on page 226 covers the choice of major and minor device numbers.
- "Defining Device Special Files" on page 227 describes options for creating the file or files controlled by the driver.
- "Compiling and Linking" on page 228 covers the compiler and linker options used for driver modules.
- "Configuring a Nonloadable Driver" on page 232 describes the configuration files used to set up a driver loaded at boot time.
- "Configuring a Loadable Driver" on page 237 describes the additional configuration needed for a loadable driver.

Defining Device Numbers

The topics "Major Device Number" on page 34 and "Minor Device Number" on page 35 cover the purpose and use of the device numbers, which can be summarized as follows:

- Both numbers are encoded in the inode of a device special file in */dev.*
- The major number selects the device driver.
- The minor number specifies the logical unit, and can encode device features.
- Both numbers are passed as a parameter to driver entry points.

An important part of creating and installing a device driver is the selection of device numbers and the definition of device special files.

Selecting a Major Number

You must select a major number to stand for your driver. The numbers that already exist are listed in *sys/major.h*. However, the major number should not be coded into the driver. Typically the driver code does not need to know its major number, and if it does, the driver should discover its major number dynamically. A method of doing this is discussed under "Variables Section" on page 235.

A driver is associated with its major number in the *master.d* configuration file. When the driver discovers this number dynamically, the system administrator is free to change major numbers in */var/sysgen/master.d* files to correct conflicts between one product and another.

It is possible to let the *lboot* command choose a major number dynamically. This is discussed under "Configuring for a Dynamic Major Number" on page 241.

Selecting Minor Numbers

Each device minor number comprises 18 bits of information that are passed to driver entry points. The format of minor numbers can be coded into the driver. You design the content of these numbers to give your driver the information it needs about each device special file.

Examine the */dev/MAKEDEV* script to see some techniques for assembling minor numbers dynamically based on the hardware inventory and other commands.

Defining Device Special Files

As described under “Device Special Files” on page 32, the association between a device and a driver is established by encoding the major device number in the *inode* of a device special file in the */dev* directory. Without at least one device special file, a device can never be opened.

Static Definition of Device Special Files

The system administrator can create device special files using *mknod* or *install* (see “Making Device Files” on page 37). This can be done manually, or through an installation script, or as an exit operation of the software manager program. The device special files can be created at any time—whether or not the device driver is installed, and whether or not the hardware exists. The special device files continue to exist until the administrator removes them.

Dynamic Definition of Device Special Files

In a more sophisticated installation you might want to have the device special files created, or recreated, dynamically each time the system boots. This is the purpose of */dev/MAKEDEV* (see “The Script MAKEDEV” on page 36); it removes and redefines device special files based on information in the hardware inventory.

Much of the logic in */dev/MAKEDEV* depends on information reported by the *hinv* command. Through IRIX 6.2, there is no OEM interface for drivers to the hardware inventory (see “Hardware Inventory” on page 29), so at this time there is no opportunity for your driver to pass information through the inventory to */dev/MAKEDEV*.

However, there are other possibilities. For example, if your driver supports a control unit with an unknown number of minor devices, you could statically create a single device special file to represent the control unit. You could write the driver so that an **ioctl()** function directed to this file returns the count of actual minor devices.

Compiling and Linking

You compile a kernel device driver to an ELF binary (in IRIX 5.3 and before, the COFF binary format was used) not using shared libraries. The compile options differ between 32-bit and 64-bit modules.

Using /var/sysgen/Makefile.kernio

The file */var/sysgen/Makefile.kernio* is a sample Makefile for compiling kernel modules. You can include it from a Makefile of your own to set the definitions of compiler variables and compiler options appropriately for different CPUs and module types.

The *Makefile.kernio* file tests the following environment variables:

CPUBOARD	Set to the type of CPU used in the target system, for example IP19 or IP22.
COMPILATION_MODEL	Set to 64 for a 64-bit kernel module, or to 32 for a 32-bit kernel module.

The purpose of the rules in *Makefile.kernio* is to set compiler variables and compiler options into a Make variable CFLAGS. Other Make variables would be set by your own Makefile.

Note: *Makefile.kernio* is designed for nonloadable drivers. In particular it sets the compiler option *-G8*, which is valid for nonloadable drivers. A loadable driver must use *-G0*.

Compiler Variables

The compiler variables listed in Table 10-1 are tested in system header files. They are usually defined on the compiler command line. The rules in */var/sysgen/Makefile.kernio* set definitions of the variables appropriately for different CPU types.

Table 10-1 Compiler Variables Tested by System Header Files

Variable	Meaning
_K32U32	Compile for a 32-bit kernel running (only) 32-bit user programs.
_K32U64	Compile for a 32-bit kernel running 32-bit or 64-bit user programs (not supported in IRIX 6.2).
_K64U64	Compile for a 64-bit kernel running 32-bit or 64-bit user programs.
_KERNEL	Compile for a kernel module, not a user program.
STATIC=static	Use of pseudo-declarator STATIC is converted to real static.
JUMP_WAR	Compile workaround code for R4000 branch on end of page bug.
PROBE_WAR	Compile workaround code for R4000 bug which requires TLBprobe instruction to be performed in uncached mode.
BADVA_WAR	Compile workaround code for R4000 badvaddr bug.
TFP	Target machine is the R8000.
R4000	Target machine is the R4000.
R3000	Target machine is the R3000 (not supported after IRIX 5.3).
IP*nn*	Target machine uses the IP*nn* CPU module: IP17, IP19, IP20, IP22, IP26 are currently supported.
_PAGESZ=16384	Compile for a kernel using 16K memory pages (with IRIX 6.2 this implies _K64U64 also defined).
_PAGESZ=4096	Compile for a kernel using 4K memory pages (with IRIX 6.2 this implies _K32U32 also defined).
_MIPS3_ADDRSPACE	Kernel for a MIPS III (R8000) machine.
EVEREST	Compile for a Challenge or Onyx system.
MP	Compile for a multiprocessor.

Compile Options, 32-Bit Kernel

The *cc* and *ld* options shown in Table 10-2 are needed to compile and link a kernel-level driver for a 32-bit kernel in IRIX 6.2.

Table 10-2 Compiler Options for 32-Bit Kernel Modules

Option	Purpose
-non_shared	Do not compile for shared libraries (dynamic linking).
-elf	Compile and link an ELF binary.
-32 -mips2	Create a 32-bit executable for the MIPS II architecture.
-G 8	In a nonloadable driver, use the global table for objects up to 8 bytes.
-G 0	In a loadable driver, do not use the global table. Refer to the gp_overflow(5) reference page for a discussion of the global table.
-O2	Maximum recommended optimization level.
-r	Linker to retain symbols (needed by loadable drivers only).
-d	Force definition of common storage even though *-r* used.
-Wc,-pic0	Do not allocate stack space used by shared objects.
-jalr	Generate **jalr** instructions for subroutine calls rather than **jal**, which allows only a 26-bit target and so cannot address all kernel virtual storage.

Compile Options, 64-Bit Kernel

The *cc* and *ld* options listed in Table 10-3 are needed to compile and link a kernel-level driver for a 64-bit kernel in IRIX 6.2.

Table 10-3 Compiler Options for 64-Bit Kernel Modules

Option	Purpose
-non_shared	Do not compile for shared libraries (dynamic linking).
-elf	Compile and link an ELF binary.
-64 -mips3	Create a 32-bit executable for the MIPS III architecture. You can use *-mips4* instead, but only in a system based on the R10000 processor.
-G 8	In a nonloadable driver, use the global table for objects up to 8 bytes.
-G 0	In a loadable driver, do not use the global table. Refer to the gp_overflow(5) reference page for a discussion of the global table.
-O2	Maximum recommended optimization level.
-TENV:kernel *-TENV:misalignment=1*	Execution environment options for 64-bit compiler.
-OPT:space *-OPT:quad_align_branch_targets=FALSE* *-OPT:quad_align_with_memops=FALSE* *-OPT:unroll_times=0*	Specific optimization constraints for 64-bit compiler.

Configuring a Nonloadable Driver

When the driver is not loadable, it is linked as part of the IRIX kernel. The following steps are needed to make the driver usable:

1. Place the driver executable file in */var/sysgen/boot*.
2. Place a descriptive file in */var/sysgen/master.d*.
3. Place a directive file in */var/sysgen/system* (or simply add a line to */var/sysgen/system/irix.sm*).
4. Run *autoconfig* to generate a new kernel.
5. Reboot the system.

Some of these steps are elaborated in the following topics.

How Names Are Used in Configuration

The naming of a kernel-level driver begins in a file in */var/sysgen/system*, such as */var/sysgen/system/irix.sm*. Names are used as follows:

- A USE, INCLUDE, or VECTOR statement in */var/sysgen/system* specifies a name, for example

  ```
  USE hypothetical
  ```
- This statement directs *lboot* to read a file of the same name in */var/sysgen/master.d*, for example, */var/sysgen/master.d/hypothetical*.
- The file in */var/sysgen/master.d* specifies the prefix for driver entry points, for example *hypo_*.
- The same name with the suffix *.o*, is searched for in */var/sysgen/boot*—for example, */var/sysgen/boot/hypothetical.o*. This object file is linked with the kernel.
- The public symbols in the object file are searched for names that start with the prefix, for example **hypo_edtinit()**. These are noted in the kernel switch table so the driver can be called as needed.

Placing the Object File in /var/sysgen/boot

The */var/sysgen/boot* directory, where the kernel object modules reside, is actually a symbolic link to one of the directories */usr/cpu/sysgen/IP*nn*boot*, where *nn* is the number of one of the CPU modules supported by the current release of IRIX (see "CPU Modules" on page 5). When you place the object file of a driver in */var/sysgen/boot*, you actually place it in the CPU directory for the system in use.

Describing the Driver in /var/sysgen/master.d

You describe your driver in a file with the name of the driver in */var/sysgen/master.d*. The format of these files is described in two places: the master(4) reference page, and in */var/sysgen/master.d/README*. In addition, you can examine the many examples in the distributed system.

Descriptive Line

The first noncomment line of the master file contains a series of fields, delimited by white space, to describe the driver. These fields are:

Flags	Described in the text.
Prefix	The string of 1-14 characters that identify the public symbols of driver entry points.
Major number	The major number, or a comma-separated list of major numbers, found in device special files that are managed by this driver.
Number of sub-devices	Size of the driver's static arrays of device information, or given as a hyphen "-" when not known.
Dependencies	A list of other modules that must be in the kernel for this driver to work—usually omitted except for SCSI drivers.

Of the many flags that can be specified in the first field, the following are most frequently used:

b or *c*	Block (b) or character (c) device. One or the other is essential for any device driver.
f or *m*	STREAMS driver (f) or module (m).
n or *p*	Multiprocessor-aware (n) or uniprocessor-only (p) driver. This flag must correspond to the presence or absence of D_MP in the *pfx*<b>devflag</b> global.
s	Software driver, either a pseudo-device or a SCSI driver.

The *s* (software-only) flag tells lboot not to attempt to probe for hardware. This is the case with software-only (pseudo-device) drivers, and with SCSI drivers. If *lboot* tries to probe for a SCSI device, it fails, and assumes that the device is not present, and does not include your SCSI device driver.

Additional flags (d, R, N, D) for loadable drivers are discussed later under “Configuring a Loadable Driver” on page 237.

Listing Dependencies

The descriptive line ends with a comma-separated list of other loadable kernel modules on which this driver depends. The *lboot* command makes sure that, if it fails to load a dependent module, it also will not load this module.

In most cases, an OEM driver does not have any dependencies. However, a SCSI driver (see Chapter 15, “SCSI Device Drivers”) should list the name *scsi*, since it depends on the inner SCSI driver. A STREAMS driver might list the name of a STREAMS support module such as clone (see “Support for CLONE Drivers” on page 552).

It is possible for you to design a driver in the form of multiple, loadable modules. In that case, you would name your support modules in this field.

Stubs Section

Noncomment lines that follow the descriptive line and precede a line beginning "$" are used by library modules—not by device drivers or STREAMS drivers. Each such line specifies an entry point that this module provides, and which is used by the kernel or some other loadable module. These lines instruct *lboot* in how to create a harmless "stub" function in the event that this driver is not included in the kernel—for example, because it is specified by an EXCLUDE line in the system file. The format is discussed in the master(4) reference page.

Since a device or STREAMS driver provides only standard entry points that are accessed via the switch tables rather than by linking, drivers do not need to define any stubs.

Variables Section

Following the descriptive line (and the stubs section, if any), you can place a line that begins with "$" and, following this, you can write definitions of global variables using C syntax. This text is compiled into a module linked with the kernel. You refer to these variables as *extern* in the driver source code.

The advantage of defining global variables in the master file is that the initializing expressions for these variables can include the major device number and the number of supported sub-devices, as specified in the descriptive line of the file. This is how you make these items available to your driver without coding them into the driver. In the source code of the driver you can write

```
extern major_t myMajNum;
extern int myDevLimit;
```

In the master file you can write

```
$$$
major_t myMajNum = ##E;
int myDevLimit = ##C;
```

The global *myDevLimit* is compiled with the number of devices from the descriptive line. The global *myMajNum* is compiled with the major number. (In a loadable driver this technique requires one additional step; see "Master File for Loadable Drivers" on page 238.)

Configuring a Kernel

Once you have placed the binary in */var/sysgen/boot* and the master file in */var/sysgen/master.d*, you can configure and generate a new kernel. This is done using the *autoconfig* command, which in turn calls *lboot* to actually create a new bootable file.

The *lboot* program only loads modules that are specified in a file in */var/sysgen/system*. One command is required to specify the new driver; the command is one of the following:

VECTOR	To specify hardware details, to request a hardware probe at boot time, to load the driver and invoke *pfx***edtinit()**.
INCLUDE	To load the driver and invoke *pfx***init()**.
USE	To load the driver and invoke *pfx***init()** only if the master file exists in *master.d*.

The form of these commands is detailed in the system(4) reference page. In addition, you should examine the distributed files in */var/sysgen/system*, especial *irix.sm*, which contains many comments on the meaning and use of different entries. Specific uses of the VECTOR statement are discussed in the following topics:

- The form of VECTOR lines for VME devices is discussed under “Configuring the System Files” on page 311.
- The form of VECTOR lines for EISA devices is discussed under “Configuring IRIX” on page 436.
- The form of VECTOR lines for GIO devices is discussed under “Configuring a GIO Device” on page 514.

You could place the VECTOR, USE, or INCLUDE line for your driver in *irix.sm*. However, since *lboot* treats all files in */var/sysgen/system* as a single file, you can create a small file unique to your driver. The name of this file is not significant, but a good name is the driver name with the suffix *.sm*.

Generating a Kernel

The *autoconfig* script invokes *lboot* to read the system files, read the master files, and link all the kernel executables. Provided there are no errors, the output is a new file */unix.install*. At boot time this file is moved to the name */unix* and used as the kernel.

During the testing period you may want to keep the original kernel file available as */unix.original*. A simple way to retain this file is to create a link to it using the *ln* command.

Configuring a Loadable Driver

You compile and configure a loadable driver very much as you would a nonloadable driver. The differences are as follows:

- You provide an additional global variable with the public name *pfx***mversion**.
- You use a few different compile options.
- You decide when the driver should be loaded, and use appropriate flags in the descriptive line in the master file.

For more background on loadable modules, see the mload(4) and ml(1) reference pages.

Public Global Variables

To be loadable, a driver must specify a *pfx***devflag** entry point, and it may not contain the D_OLD flag (see “Driver Flag Constant” on page 140).

Any loadable module must define a public name *pfx***mversion**, declared as follows:

```
#include <sys/mload.h>
char *pfxmversion = M_VERSION;
```

Note the exact spelling of the variable; it is easy to overlook the letter “m” after the prefix. If the variable does not exist or is incorrectly spelled, an attempt to load the driver will fail.

Compile Options for Loadable Drivers

Use the *-G 0* option when compiling and linking a loadable driver, since the global option table is not available to a loadable driver (see "Compile Options, 32-Bit Kernel" on page 230 and "Compile Options, 64-Bit Kernel" on page 231).

In a loadable driver, link using the *-r* and *-d* options to retain the symbol table yet generate a bss segment.

Master File for Loadable Drivers

The file in */var/sysgen/master.d* for a loadable driver has different flags.

Descriptive Line

In the flags field of the descriptive line of the master file (see "Descriptive Line" on page 233), you specify that the driver is loadable, and when it should be loaded. The possible flags are as follows:

d	Specifies that this is a loadable driver.
R	Auto-register the module (discussed in text).
D	Load, then unload, at boot time, in order to let the driver initialize the hardware inventory.
N	Prevent this module from being automatically unloaded even when it has a *pfx***unload()** entry point.

When the *d* flag is given for an included module, *lboot* parses the master file for the driver. Global variables defined in the variables section of the master file (see "Variables Section" on page 235) are defined and included in the kernel. However, object code of the driver is not included in the kernel, and the names of its entry points are not entered into the kernel switch table.

Such a driver has to be manually loaded with the *ml* or *lboot* command before it can be used; and it cannot be used from the miniroot.

Registration

A loadable module is "registered" by having a stub entry placed in the *pfx***open()** column of a kernel switch table, indicating it exists but is not loaded. Registration can be done manually, after bootstrap, or automatically during bootstrap.

Registration is done automatically by for each master descriptive file that contains the *d* and *R* flags. Autoregistration is done at bootstrap phase 2. It is initiated by the script */etc/rc2/S23autoconfig*. Registration can be done manually at any time after bootstrap by using the *ml* or *lboot* command with the *reg* option (see the ml(1M) and lboot(1M) reference pages).

Once it has been registered, a driver is loaded automatically the first time a process attempts to open a device special file with the major device number of this driver. You can also load a registered driver in advance of any use with the *ml* or *lboot* command—loading implies registration.

Loading

A registered, loadable driver can be loaded by the *lboot* and *ml* commands. When a driver is loaded, the following steps are taken:

1. The object file header is read.
2. Memory is allocated for the module's text, data, and bss segments.
3. The module's text and data are read.
4. The module's text and data are relocated. References to kernel names and to global variables named in the master file are resolved.
5. The module entry points are noted in the appropriate kernel switch table.
6. The module's *pfx***init()** or *pfx***edtinit()** entry point is called, depending on whether the module is specified by an INCLUDE or a VECTOR statement in the system file (see "Initialization Entry Points" on page 143).
7. The module's *pfx***start()** entry point, if any, is called.

Space allocated for the module's text, data, and bss is located in either *k0seg* or *k2seg*. Static buffers in loadable modules are not necessarily physically contiguous in memory.

When the *D* flag is included in the descriptive line in the descriptive file, the module is loaded at boot time and then unloaded. This gives the module a chance to update the hardware inventory.

If errors occur when a module is loaded, see the mload(4) reference page for a list of possible causes.

Unloading

A module can be unloaded only when it provides an *pfx***unload()** entry point (see "Entry Point unload()" on page 167). The *N* flag can be specified in the master file to prevent automatic unloading in any case.

A loaded module is automatically unloaded following a delay after the last close of a device it supports. The delay is configurable using *systune*, as the *module_unld_delay* variable (see the systune(1) reference page). You can use *ml* to specify an unloading delay for a particular module.

The *lboot* or *ml* command can be used to unload a module before the delay expires, or when the *N* flag prevents automatic unloading.

Just before unloading, the kernel invokes the driver's *pfx***unload()** entry point. Then the module is removed from memory, and returned to registered status. It is up to the *pfx***unload()** entry point to decide whether unloading can be permitted. Experience has shown that most of the problems with loadable drivers arise from unloading and reloading. The precautions to take are described under "Entry Point unload()" on page 167.

Configuring for a Dynamic Major Number

You can place a hyphen ("-") in the third field of the descriptive line; that is, the field for the major device number (see "Descriptive Line" on page 233). When the line also contains the *b* or *c* flag, indicating a device driver, a hyphen in the third field means that *lboot* is to assign some unused major device number dynamically when the module is registered.

Since a device driver can only be called by opening a device special file, and since a device special file has to be defined based on a major device number, how can the device special files be created?

The assigned major number can be discovered using the *ml* command. The command

```
ml list -r
```

writes a list of all registered modules, including their major numbers. The following procedure can be used to make the assignment of the major device number completely dynamic.

1. Make the device driver loadable, specifying the *d*, *R*, and *D* flags.
2. Specify a hyphen for the major number.
3. In the driver, get the major number dynamically, if indeed the driver needs to know its major number at all.
4. In a script executed from *rc2.d* (just as the */dev/MAKEDEV* script is executed), call *ml* to list registered drivers.
5. Parse the output of *ml* using UNIX utilities to extract the major device number from the line describing your driver.
6. Execute *mknod* or *install* commands to create special device files using the discovered major number.

Testing and Debugging a Driver

As a critical system component, a driver deserves careful testing, but because it is part of the kernel, the normal testing tools are not available. This chapter describes the available testing tools and methods, in the following major topics:

- "Preparing the System for Debugging" on page 243 describes how to set up the kernel for use of the debugging tools.
- "Producing Diagnostic Displays" on page 249 covers the kernel functions your driver can use to generate diagnostic output as it executes.
- "Using symmon" on page 252 describes the use of the standalone debugger.
- "Using idbg" on page 262 describes some uses of the kernel-display command.

Preparing the System for Debugging

The standalone debugger *symmon* is a key tool for driver programming. It must be installed in the volume header of the boot disk. In order for it to be useful you must boot a "debugging" kernel, that is, one that retains symbols, and contains the display modules, that are used by debugging tools. Normally these modules and symbols are eliminated to save space. You modify the *irix.sm* file to enable debugging, and then generate a new kernel.

All these steps should be performed before you attempt to install your device driver.

Placing symmon in the Volume Header

The *symmon* standalone debugger resides in the volume header of a disk—not in a normal IRIX filesystem. The volume header is disk partition 0. It always contains a label record (sgilabel), and the standalone shell *sash* that manages the bootstrap operation. On some systems it may also contain the *ide* program, a PROM-level diagnostic program. If *symmon* is to be available, it, too, must be placed in the volume header.

Normally you acquire *symmon* by installing the debugging kernel feature (eoe.sw.kdebug) in the IRIX Developer Option software distribution. You can verify that this feature has been installed by executing the command

```
versions eoe.sw.kdebug
```

The response should confirm the presence of this component (it does not show *symmon* by name). When you install the kernel debug feature, the *symmon* program file is copied to the volume header of the current boot disk automatically.

You can verify the presence of *symmon* in the volume header through the use of *dvhtool* (described in the dvhtool(1) reference page). The results should be similar to the display in Example 11-1. The response to the "l" (list) command shows that the volume header of this disk contains *sgilabel*, *ide*, *sash*, and *symmon*.

Example 11-1 Verifying Presence of symmon

```
# dvhtool
Volume? (/dev/rvh) /dev/rvh
Command? (read, vd, pt, dp, write, bootfile, or quit): vd
(d FILE, a UNIX_FILE FILE, c UNIX_FILE FILE, g FILE UNIX_FILE or l)?
        l
Current contents:
        File name        Length     Block #
        sgilabel            512           2
        ide              281600         278
        sash             281600         828
        symmon           248320        1378
(d FILE, a UNIX_FILE FILE, c UNIX_FILE FILE, g FILE UNIX_FILE or l)?

Command? (read, vd, pt, dp, write, bootfile, or quit): quit
#
```

In the event you need to install *symmon* in the volume header of a disk without using the software manager, you can copy the standalone program to the volume header using *dvhtool*. However, you first need to get a copy of the program in the form of a UNIX file.

Starting from a volume that currently has a copy of *symmon* (verified as in Example 11-1), use *dvhtool* to extract a copy of *symmon* into a convenient spot.

```
dvhtool -v g symmon /var/tmp/symmon.IPxx
```

There is a unique version of *symmon* for each CPU module, so it is a good idea to qualify the filename with the CPU module type. Once the program is available as a normal file, you can use *dvhtool* to install it in the volume header of some other disk.

In the event there is not enough room in partition 0 (the volume header) of the target disk, it is safe to use *dvhtool* to delete the *ide* program from the volume header. The *ide* application can be booted manually from a CDROM if it is ever required.

Enabling Debugging in irix.sm

In order to make debugging symbols available in the kernel, you must make two changes, one required and one optional, in the file */var/sysgen/system/irix.sm*. As superuser, make a hard link to the file */var/sysgen/system/irix.sm* as *irix.sm.nondebug*. This enables you to return easily to a nondebugging kernel.

Including Symbols in the Kernel Image

Edit */var/sysgen/system/irix.sm*. Near the end, note the lines that resemble the following:

```
* Compilation and load flags
*   To load a kernel that can be co-resident with symmon
*   (for breakpoint debugging) replace LDOPTS
*   with the following.  You must also INCLUDE prf and idbg.
*
*LDOPTS: -non_shared -N -e start -G 8 -elf -woff 84 -woff 47 -woff 17
-mips2 -o32  -nostdlib  -T 88069000
```

The active LDOPTS statement (the one without an initial asterisk) appears a few lines later. Remove the asterisk from the front of the debugging LDOPTS to make it active. Insert an asterisk to convert the original LDOPTS into a comment.

Including idbg in the Kernel Image

The symbol-display routines used by the command-line kernel display tool, *idbg*, are contained in optional kernel modules. (See "Using idbg" on page 262.) You can change */var/sysgen/system/irix.sm* so that support for *idbg* is always present in the kernel. Alternatively, you can load these modules manually with *ml* before you use them (see the ml(1) reference page).

If you are entering an extended debugging period, make the modules permanent. Look for the lines in */var/sysgen/system/irix.sm* that resemble the following:

```
*
* Kernel debugging tools (see profiler(1M) and idbg(1M))
*
EXCLUDE: idbg
EXCLUDE: dmiidbg, grioidbg, xfsidbg, xlvidbg, cachefsidbg
USE : prf
```

Change these lines, if necessary, so that both *idbg* and *prf* are marked USE, not EXCLUDE. Verify that the file */var/sysgen/boot/idbg.o* exists. It is normally installed with the debugging kernel feature.

Parts of the *idbg* support that are unique to particular filesystems are in the other modules listed in this area of *irix.sm*. Modules such as *xlvidbg* are useful to Silicon Graphics developers but are not likely to be helpful to developers of third-party drivers. However, it does no harm to change those modules from EXCLUDE to USE also.

Including Lock Metering in the Kernel Image

In addition to the display support included by the idbg modules, you can include a module that supports lock metering. This module keeps statistics on the use of of each semaphore, basic lock, and reader/writer lock, and displays the statistics through idbg commands. To enable this facility, locate the lines in */var/sysgen/system/irix.sm* that resemble the following:

```
* Required kernel modules
...
* ksync - kernel synchronization routines (mutex_lock, sv_wait,
psema...)
*   or
*   ksync_metered  - metered kernel synchronization routines
...
*
```

```
KERNEL: kernel
INCLUDE: os, disp, mem, zero
INCLUDE: ksync
EXCLUDE: ksync_metered
```

Reverse the state of the two “ksync” lines so that ksync is excluded and ksync_metered is included.

Generating a Debugging Kernel

Run the *autoconfig* command to generate a new kernel that will reflect the changes made in *irix.sm*. The result is a new kernel file, */unix.install*, that will be renamed to */unix* and used when the system is booted. This kernel can support *idbg* but is not yet ready for standalong debugging with *symmon*.

The *setsym* command copies the symbol table from a kernel file and stores it as data within the kernel, so that *symmon* can find it. After autoconfig has created */unix.install*, apply the setsym command to it, as follows:

```
#setsym /unix.install
```

If this command returns an error message about “symbol table overflow,” it means you have neglected to activate the debugging LDOPTS statement in */var/sysgen/irix.sm*.

Tip: You can use *setsym* with the *-d* option to generate a list of all symbols in the kernel being modified. The list is very long; direct it to a file for later reference.

At this time, you may wish to create a link to the current, nondebugging kernel so you can retrieve it easily. You can also return to a nondebugging kernel by restoring the original *irix.sm* file and running *autoconfig* again.

Specifying a Separate System Console

In order to use the standalone debugger, you must have an ASCII terminal as a separate system console device. Install a terminal next to the system or workstation and connect it to the first serial port. Verify its operation as a terminal by logging in to the system. You may have to modify the file */etc/inittab* so that the line for the alternate console is active (see the inittab(4) reference page).

When you have verified that the terminal works, use the *nvram* command to change the nonvolatile RAM variable console from a letter “g” to a letter “d,” as follows:

```
# nvram console
g
# nvram console d
# nvram console
d
```

The *nvram* command is used to report and change the contents of the nonvolatile RAM variables used by the boot PROM and standalone shell (see the nvram(1) reference page).

Verifying the Debugging Tools

After performing the preceding steps, restart the system. Messages from *sash* appear on the attached terminal, rather than on the graphics screen. If *symmon* is present, it announces itself on the console terminal also.

To verify operation of *idbg*, issue the idbg command and display the process list:

```
# idbg
idbg> plist
active process list:
34:672:”xdm” pri(60) SLEEP flags: load uload siglck recalc sv
0:0:”sched” ndpri(39) SLEEP flags: sys nwake load uload sv
31:193:”inetd” pri(60) SLEEP flags: load uload siglck recalc sv
...
```

To verify operation of *symmon*, press control-A at the console terminal. The prompt string DBG: should appear. At this time the system is frozen and no longer responds to mouse or keyboard input. Type the letter c (for continue) and press return. The system returns to life.

Producing Diagnostic Displays

Normally a device or STREAMS driver produces display output in only two cases:

- To advise the operator or administrator of a serious problem.
- To display debugging information during software development.

Both of these purposes are served by the **cmn_err()** function. It brings to a kernel-level module the abilities that a user-level process gets from **printf()** and **syslog()**.

Using cmn_err

The details of **cmn_err()** usage are in the cmn_err(D3) reference page. The function prototype and the constant values it uses are declared in *sys/cmnerr.h*.

In summary, **cmn_err()** takes two or more arguments:

- A severity code that specifies how the message should be treated when it is written to the system log.
- A message string, which can have substitution points in the style of **printf()**.
- As many numeric values as are needed to substitute into the message string.

The first character of the message string specifies the destination of the message, either an in-memory buffer or the system log, or both.

Displaying to the System Log

The message is sent to the the system log daemon whenever the first message character (after substitution) is not an exclamation mark (“!”). The message is written only to the system log when the first message character is a circumflex (“^”).

This is basically the same service that a user-level process receives from the **syslog()** function. (Compare the syslog(3) and cmn_err(D3) reference pages, and examine the *sys/cmnerr.h* header file; the relationship is clear.) The first argument to **cmn_err()** is a severity code which corresponds to one of the severity codes supported by **syslog()**: CE_WARN equals LOG_WARN, and so on.

Use **cmn_err()** to write log messages to record serious errors (with CE_ALERT severity) or to advise the administrator of conditions that should be changed (using CE_NOTE).

Displaying to the Circular Message Buffer

The message is stored in the next available position in a circular buffer in kernel memory whenever the first message character (after substitution) is not a circumflex (“^”). The message is stored only in the memory buffer when the first message character is an exclamation mark (“!”).

The name of the circular buffer (as a symbol to *idbg* or *symmon*) is *putbuf*. The contents of *putbuf* can be displayed with the *pb* command of either *idbg* or *symmon* (see“Using symmon” on page 252 and “Using idbg” on page 262), or in a post-mortem dump using *icrash* (see “Using icrash” on page 269). Use **cmn_err()** to store debugging trace data in the circular buffer, and extract it after a stop or breakpoint with *symmon*, or use *idbg* to look at it while the system is running.

The size of the buffer can be configured by changing the kernel variable *putbufsz*, as shown in the dialog in Example 11-2.

Example 11-2 Setting Kernel putbuf Size

```
# systune -i
Updates will be made to running system and /unix.install
systune-> putbufsz
        putbufsz = 1024 (0x400)
systune-> putbufsz 4096
        putbufsz = 1024 (0x400)
        Do you really want to change putbufsz to 4096 (0x1000)? (y/n)
y
In order for the change in parameter putbufsz to become effective,
reboot the system
systune-> quit
```

Using cmn_err() Through Macros

The inventive C programmer can think of many ways to invoke **cmn_err()** using macros. One method is illustrated in the example driver displayed in Chapter 12, “Driver Example.” It contains the code shown in Example 11-3.

Example 11-3 Debugging Macros Using cmn_err()

```
#ifdef DEBUG
#define DBGMSG0(s) cmn_err(CE_DEBUG,s)
#define DBGMSG1(s,x) cmn_err(CE_DEBUG,s,x)
#define DBGMSG2(s,x,y) cmn_err(CE_DEBUG,s,x,y)
#define DBGMSG3(s,x,y,z) cmn_err(CE_DEBUG,s,x,y,z)
#else
#define DBGMSG0(s)
#define DBGMSG1(s,x)
#define DBGMSG2(s,x,y)
#define DBGMSG3(s,x,y,z)
#endif
```

Calls to the DBGMSG macros evaluate to nothing unless the DEBUG variable is defined to the compiler. Macro use could be more elaborate. For example, suppose that you want every debugging macro to start with the same string, and that you do not want to have the tedium of coding the newline at the end of every debug message text. You could write macros like the one shown in Example 11-4.

Example 11-4 More Elaborate Debugging Macro

```
#ifdef DEBUG
char *_dbg_prefix = "mydriver: %s\n";
#define DBGPREFIX(s) cmn_err(CE_DEBUG,_dbg_prefix,s)
...
#else
#define DBGPREFIX(s)
...
#endif
```

Using printf()

You can call the **printf()** function from a kernel module. The kernel version of **printf()** is basically a call to **cmn_err()** with severity CE_CONT. In general it is better to use **cmn_err()** explicitly.

Using ASSERT

The **assert()** macro is familiar to many C programmers; it terminates a program with a message if its argument evaluates to false (see the assert(3X) reference page). This normal assert**()** macro does not work in a kernel module because the normal C library is not available. However, a similar function is available as the ASSERT() macro in the header file *sys/debug.h*.

The ASSERT() macro compiles to null code unless the compiler variable DEBUG is not only defined, but defined as YES. When it compiles to executable code, ASSERT() tests its argument. If the argument evaluates to false, a kernel panic is forced.

Clearly ASSERT() must be used with care, testing conditions that are truly essential to the integrity of the system. When reporting conditions that are merely operational errors, use a call to **cmn_err()** with the CE_WARN option.

Using symmon

The *symmon* program is a standalone debug monitor that can display and modify memory, and stop, start, and trace execution, without using any kernel facilities. Using *symmon* you can set breakpoints in your driver, single-step its execution, and display the contents of driver and kernel variables.

The facilities of *symmon* are unsophisticated compared to the high-level debuggers you might use to debug a user-level application. For example, *symmon* does not understand C syntax, so it cannot display data structures as structures. Execution tracing is done at the level of machine instructions, not at the level of C statements.

However, you can use *symmon* to examine the operations of a kernel module in a running system, and resume execution of the system. This is an invaluable facility when debugging a new driver.

How symmon Is Entered

When the system boots a debugging kernel with *symmon* installed, control can pass into the debug monitor under several different circumstances:

- Early in the bootstrap process, if certain environment variables are set in the stand-alone shell (see “Entering symmon at Boot Time” on page 254).
- Whenever a control-A character is typed at the system console terminal.
- Whenever a breakpoint is reached or a watchpoint is tripped (see “Commands to Control Execution Flow” on page 257).
- Whenever a kernel module calls the kernel function **debug**(*uchar_t *msg*).
- When a non-maskable interrupt (NMI) is detected.
- When a kernel panic is detected or forced with **cmn_err()**.

When *symmon* gains control, it displays its “DBG:” prompt at the console terminal and waits for a command.

To resume execution at the point of interruption, enter the *c* (continue) command.

Using symmon in a Uniprocessor Workstation

In a single-processor workstation such as the Indy or Indigo^2, no IRIX execution takes place while *symmon* is running. The mouse and keyboard are unresponsive. (One keystroke may be stored in the keyboard hardware to be processed when the system resumes execution.) As a result, time-dependent processes can fail; for example, the system clock is not updated. Network interrupts are not taken, so if the workstation is acting as an NFS server, it will appear to be dead to other systems.

Using symmon in a Multiprocessor Workstation

In a multiprocessor, the CPU that was interrupted runs *symmon* and nothing else. For example, the CPU that executes the breakpoint, or the CPU that handles the DUART interrupt that returns the control-A character, or the CPU in which **debug()** was called, comes under the control of *symmon*. Other CPUs continue to execute normally. However, if the *symmon* CPU holds a lock, other CPUs may come to a halt waiting for the lock to be released.

The *symmon* breakpoint table is shared by all CPUs. A breakpoint set from one CPU can be taken by another CPU, or by multiple other CPUs. It is possible to run multiple instances of *symmon* concurrently. The output from all instances of *symmon* is multiplexed onto the system console terminal. However, only one CPU at a time issues the DBG: prompt. Use the *cpu* command with no argument to find out which CPU is prompting. Use the *cpu* command with a cpu number to switch to a different CPU. (See "Commands to Control Execution Flow" on page 257.)

Tip: To make multiprocessor execution more predictable, reduce the number of available CPUs. Use the PROM monitor to disable one or more CPUs before bootstrap.

Entering symmon at Boot Time

You can cause the kernel to stop during initialization and enter symmon during the bootstrap process. In order to do this, you must use the miniroot to set environment variables.

1. Restart the system, for example by giving the commands *sync* and *halt*. Eventually, the 5-item PROM menu is displayed at the console terminal.
2. Select item 5, "Enter the Command Monitor."
3. Set one or both of the environment variables *dbgstop* and *symstop* to 1, using commands such as the following:

   ```
   >> setenv symstop 1
   ```
4. Return to the PROM menu by entering the command *exit*.
5. Select menu item 1, "Start System."

In either case, *symmon* seizes the system and displays its DBG: prompt at the system console during bootstrap. When the *dbgstop* variable is set, *symmon* takes control of the system very early in the bootstrap process. Symbolic names are not initialized at this point. However, breakpoints can be set and memory can be displayed using explicit addresses.

When the *symstop* variable is set, *symmon* takes control after symbols are defined, but before driver initialization is begun. At this stop, you can display memory and set breakpoints based on entry point names of your driver.

Commands of symmon

The exact set of commands supported by *symmon* changes from release to release and from CPU model to CPU model. Many *symmon* commands are useful only to Silicon Graphics engineers who are debugging hardware and kernel problems. For a complete list of commands, see the symmon(1M) reference page, or enter *symmon* and give the *help* command. You can use control-S and control-Q on the console terminal to pause the scrolling display.

The commands described in this section are generally useful and are available on all CPU models under IRIX 6.2. These commands can be grouped into the following categories:

- Conversion between symbols and memory addresses.
- Execution control, including commands for stopping, starting, and setting breakpoints.
- Display and modification of memory, including the display of machine registers and of system data structures such as the *buf_t* and *proc_t* objects.
- Management of the virtual memory system and the TLB.

Syntax of Command Elements

The *symmon* commands all have the same form: a keyword, usually followed by one or more arguments separated by spaces.

Many commands take an address value. An address argument value can have one of the following forms:

Decimal number	A number starting with 1-9 is a decimal number, for example *4095*.
Octal number	A number starting with 0 and a digit is an octal number, for example *033*.
Hex number	A number starting with 0x is a hexadecimal number, for example *0xffff8000*.
Binary number	A number starting with 0b is a binary number, for example *0b0100*.

Symbol	A word starting with a non-digit is looked up as a symbol in the kernel symbol table, and its address is the value; for example *dk_open*.
Register	A word starting with "$" is taken as a register name, and the contents of the register as of the last interrupt is used as the argument value; for example *$a2*.
Value and offset	A value plus or minus a number is a value, for example *$a2-0x100* or *dk_open+128*.

Some commands accept a range of addresses. A range can be written in one of two ways:

- As *value1*:*value2*, meaning an inclusive range of addresses from *value1* through *value2*, for example *prtbuf:prtbuf+4096*.
- As *value1*#*count2*, meaning a range of count2 bytes beginning at value1, for example *prtbuf#4096*.

The register names that *symmon* accepts and shows in various displays are the conventional names used in MIPS assembly language programming. Refer to the *MIPSpro Assembly Language Programmer's Guide* and the processor manuals listed under "Additional Reading" on page xxxix.

Commands for Symbol Conversion and Lookup

The commands summarized in Table 11-1 are used to convert between symbolic names and their corresponding addresses.

Table 11-1 Commands for Symbol Conversion and Lookup

Command	Example	Operation
hx *name*	**hx dk_read** dk_read 0xffffffff882b0510	The name is looked up on the symbol table and if it is found, its address is displayed.
lkaddr *addr*	**lkaddr 0x882b0510** 0x882af910 lockdisptab 0x882b0510 dk_read 0x882b051c dk_write	Symbols near to the specified *addr* are listed. Use this command to find out the symbolic location of an unexpected stop.

Table 11-1 (continued) Commands for Symbol Conversion and Lookup

Command	Example	Operation
lkup *letters*	**hx dk_rea** 0x880d5f10 dk_readcap 0x882b0510 dk_read 0x332b0528 dk_readcapacity	Every symbol that contains the specified *letters* at any point is listed. There is no way to anchor the search to the beginning or end of the name.
msyms *ident*	**msyms 13** Symbols for module 13 (prefix tcl) tclinit 0xc0403d9c tclmversion 0xc0405fe0	The symbols for the loadable module *ident* are listed. Use the *ml* command with no arguments to list all modules and their ident numbers.
nm *addr*	**nm 0xc0403da0** 0xc0403da0 tclinit+0x4	The symbol nearest to the specified *addr* is listed.

Note: When symmon displays an address it normally shows a full 64 bits. In a 32-bit kernel, the most-significant 32 bits of a kernel virtual address are all-binary-1, from extension of the sign bit of the 32-bit address—as shonw in the example of hx in Table 11-1. When you enter an address to a command in a 32-bit system, you need to type only the significant 32-bit value.

Commands to Control Execution Flow

The commands summarized in Table 11-2 are used to stop, start, and single-step execution in the kernel.

Table 11-2 Commands to Control Execution

Command	Example	Operation
brk	**brk**	List all breakpoints currently set.
brk *addr*	**brk dk_read**	Set a breakpoint at the specified *addr*.
c	**c**	Restart execution at the point of interruption in the current CPU.
c *cpuid* [*cpuid*]... **c all**	**c 0**	Restart execution in the specified CPU, or in all stopped CPUs. Available in multiprocessors only.

Table 11-2 (continued) Commands to Control Execution

Command	Example	Operation
call *addr* [*args*]	**call geteminor 0**	Call a kernel function and report the contents of the result register on return.
cpu	**cpu**	Displays the cpu ID of the currently-executing CPU. Available in multiprocessors only.
cpu *cpuid*	**cpu** 0	Force *symmon* execution to the specified CPU. That CPU must be executing *symmon*. Other CPUs executing *symmon* wait. Available in multiprocessors only.
goto *addr*	**goto geteminor**	Set a temporary breakpoint at *addr* and then continue execution as for the *c* command (in effect "go until *addr* is reached").
quit	**quit**	Return to the boot PROM, forcing an instant reboot.
s [*count*]	**s 8**	Single-step through 1 or *count* instructions, displaying each instruction and the register contents it uses. A branch and the instruction in "delay slot" following it count as 1. Steps into subroutines.
S [*count*]	**S 8**	Single-step through 1 or *count* instructions as for the *s* command, but do not step into subroutines.
unbrk *n*	**unbrk 2**	Remove break point number *n*. Use *brk* with no argument to list break points by number.
wpt {r | w | rw} *physaddr*	**wpt r 0x0841f608**	Set a hardware watchpoint on a physical address.

Tip: One way to force a memory dump from *symmon* is the command *call dumpsys*.

Following a break or a watchpoint, use the *bt* command to display the stack history and use *printreg* to display the registers (see "Commands to Display Memory" on page 260).

The hardware watchpoint used by the *wpt* command uses hardware registers in the MIPS R4000 and R10000 processors (the R8000 does not support the watchpoint registers). When a read or write access is addressed to any byte in the doubleword specified by the physical address, *symmon* gains control and displays the instruction that is attempting the access on the console terminal.

The argument of *wpt* must be a physical memory address and a multiple of 8. Use *tlbvtop* to get the physical equivalent of an address in a user address space (see "Commands to Manage Virtual Memory" on page 259). In a 32-bit kernel, the physical equivalent of an address in kernel space is obtained by changing the most significant hex digit to 0.

Commands to Manage Virtual Memory

The commands summarized in Table 11-3 are used to display and manage the virtual memory translation system.

Table 11-3 Commands to Manage Virtual Memory

Command	Example	Operation
cacheflush *range*	**cacheflush $6:$6+4096**	Flush both the instruction and data caches when they contain data that falls in *range*.
tlbdump [*lo:hi*]	**tlbdump 1:3**	Display the contents of the TLB registers. When a range of numbers is given, the registers from *lo* through *hi*-1 are displayed.
tlbflush [*lo:hi*]	**tlbflush**	Flush (nullify) the TLB registers specified. The registers are reloaded as required during subsequent execution.
tlbpid	**tlbpid** Current dbgmon pid = 79	Display the process slot number of the process whose context is in the TLB.
tlbvtop *addr*	**tlbptov 0xffffc000**	Display the TLB register that maps *addr*.

Commands to Display Memory

The commands summarized in Table 11-4 are used to display memory or variables.

Table 11-4 Commands to Display Memory

Command	Example	Operation
bt [*frames*]	**bt 4**	Display the calling function, the arguments, and the name of the called function for up to *frames* stack frames. Most useful after a break or interrupt.
dis *range*	**dis geteminor**	Disassemble and display the instructions over the specified range.
dump [-b | -h | -w] [-o | -d | -x | -c] *range*	dump 0xc0000000	Display memory over a specified range. The options -b, -h, and -w specify how memory is grouped, as units of 1, 2, or 4 bytes. The options -o, -d, -x, and -c specify translation into octal, decimal, hex and character.
kp [*routine*]	**kp plist**	Invoke a kernel print routine loaded with the idbg kernel module. If no routine is given, all available names are displayed.
printregs	**printregs**	Display all the registers as they were when the debugger was entered.
string *range* [*max*]	**string $v1 0x80**	Display memory as an ASCII string in quotes. Display stops at the first null byte, or, when *max* is specified, after at most *max* bytes.

The display routines available to the *kp* command are discussed under “Using idbg” on page 262. The names that *idbg* accepts as commands are all available under *symmon* through the *kp* command.

Use the *dump* command under *symmon*. Under *idbg*, use the *hd* command for the same purpose.

Utility Commands

The commands summarized in Table 11-5 are general-purpose utilities.

Table 11-5 Utility Commands

Command	Example	Operation
calc	**calc**	Starts a simple stack-oriented calculator (see text).
clear	**clear**	Clear the screen of the system console terminal.
help	**help**	List one-line summaries of all available commands. Use control-S and control-Q to control the scrolling of the display.
g [-b|-h|-w | -d] [*addr* | $*regname*]	**g $a1** 0x882fadf8: 4294967295 0xffffffff	Display one byte, halfword, word or doubleword (default word) of memory, or the contents of one register at the time symmon was entered, in decimal and hex.
p [-b|-h|-w | -d] [*addr* | $*regname*] *value*	**p -w 0xc0000000 4095**	Write a byte, halfword, word, or doubleword (default word) into a saved register or into memory at the specified address.

Using idbg

The *idbg* command is a utility that provides much of the display capability of *symmon* but from a command line, without stopping the system. Many details of *idbg* use are covered in the idbg(1M) reference page. Keep in mind that all idbg commands are available under the standalone debugger through the kp command (see “Commands to Display Memory” on page 260).

Loading and Invoking idbg

Superuser privilege is required to invoke *idbg*, because it maps kernel memory. The command is ineffective unless its support modules have been made part of the kernel. This can be done permanently by changing the *irix.sm* file (see “Including idbg in the Kernel Image” on page 246). Alternatively, you can load the needed modules dynamically using the *ml* command, as follows:

```
# ml ld -i /var/sysgen/boot/idbg.o
```

Dynamic loading is discussed at more length in the idbg(1M) and ml(1M) reference pages.

When the support modules are loaded, idbg can be invoked in three styles.

Invoking idbg for Interactive Use

Invoking the command with no arguments causes it to enter interactive mode, prompting for one command after another from standard input, as shown in Example 11-5.

Example 11-5 Invoking idbg Interactively

```
# idbg
idbg> plist 187
pid 187 is in proc slot 31
idbg> quit
#
```

The command terminates when the command quit is entered, or when control-D (standard input end of file) is typed.

Invoking idbg with a Log File

Invoking the command with the *-r* option and a filename causes it to write all its output to the specified file, as shown in Example 11-6.

Example 11-6 Invoking idbg with a Log File

```
# idbg -r /var/tmp/idbg.save
idbg> plist 187
pid 187 is in proc slot 31
idbg> proc 31
proc: slot 31 addr 0x8832db30 pid 187 ppid 1 uid 0 abi IRIX5
 SLEEP flags: load uload siglck recalc sv
...
idbg> ^D
# cat /var/tmp/idbg.save
pid 187 is in proc slot 31
proc: slot 31 addr 0x8832db30 pid 187 ppid 1 uid 0 abi IRIX5
 SLEEP flags: load uload siglck recalc sv
...
#
```

You can use this method to collect a series of displays in a single file as you test a driver.

Invoking idbg for a Single Command

You can invoke *idbg* with a command on the command line. The output of the single command is written to standard output, where it can be captured or piped to another program. Example 11-7 shows one simple use of this feature.

Example 11-7 Invoking idbg for a Single Command

```
# idbg plist | fgrep -c tcsh
3
#
```

Since the displays of *idbg* are very rich, there are endless opportunities to use this mode to generate data within shell scripts, and to process it using tools such as *awk* and *perl*. Using *perl* you could write an intelligent display routine that showed the status of your driver's private data structures using your own terminology and display format.

Commands of idbg

Almost all *idbg* commands are concerned with displaying kernel memory data in different ways. There are commands to display almost every type of kernel data.

The vocabulary of commands changes from release to release, and can change within releases by software patches. Also, the commands available depend on which support modules are loaded; for example lock and semaphore meters cannot be displayed unless the ksynch_meter module is loaded (see "Including Lock Metering in the Kernel Image" on page 246). Only a few commands are listed in the idbg(1M) reference page.

The commands summarized in this book are generally useful and available on all platforms in the current release of IRIX. For a complete (but cursory) list, use the command itself.

```
# idbg help | lp
```

In general, commands take zero or one argument. Typically the argument is a number, which can be any of the following:

- A kernel symbol, optionally +offset
- A number in hexadecimal (starting with 0x)
- A number in octal (starting with 0)
- A number in decimal.

The number is interpreted in the context of the command: sometimes it represents a process ID (pid), sometimes a process "slot" number or a buffer number. Often commands treat positive numbers as slot numbers or table indexes, while negative numbers are treated as addresses in kernel space.

Commands to Display Memory and Symbols

The commands summarized in Table 11-6 are used to display memory based on specific addresses or symbols, and to display the addresses for kernel symbols.

Table 11-6 Commands to Display Memory and Symbols

Command	Operation
dsym *addr* [*length*]	Dump memory by words, starting at *addr*. When a word of memory data is reasonably close to the value of a kernel symbol, the symbol plus offset is displayed instead of the hex value.
hd *addr* [*length*]	Dump memory in bytes, with ASCII translation, starting at *addr.* When *length* is given, it is a count of words (not bytes) to be displayed.
pb	Display the strings in the circular putbuf (see “Displaying to the Circular Message Buffer” on page 250).
string *addr* [*max*]	Display memory as an ASCII string. Display stops at the first null byte, or, when *max* is specified, after at most *max* bytes.

When you display the circular buffer, there is no special indication to show which line is the newest. You have to deduce the boundary between the newest and oldest lines from the content.

Commands to Display Process Information

The commands summarized in Table 11-7 are concerned with displaying the status of processes. Processes are recorded in an array of “slots.” The *plist* command gives the process slot number for a given process ID. The other commands take slot numbers.

Table 11-7 Commands to Display Process Information

Command	Operation
eframe [*addr* | *slot*]	Diplay the contents of an exception frame. With no argument, displays the last exception taken for the current process. Else displays the exception associated with the process specified either by address (negative number) or process table slot number (positive number)
pchain *slot*	Display the slot numbers of sibling processes to the process in *slot*.

Table 11-7 (continued) Commands to Display Process Information

Command	**Operation**
plist [0 | *pid*]	With no argument, displays a one-line summary of every active process slot, including slot number and process ID. When the argument is 0, displays all inactive process slots. With a nonzero PID, displays the slot containing that process.
ptree [*addr* | *pid*]	With a *pid* (number greater than zero), finds the process structure for that process. Else uses the process structure at *addr*. Displays the command name and command arguments for that process and for all processes that descend from it.
proc [*addr* | *slot*]	Display all the fields of a process structure specified either by address (negative number) or process table slot number (positive number).
signal [*addr* | *slot*]	Display information about pending signals for the process specified either by address (negative number) or process table slot number (positive number)
slpproc [-2 | -4 | -8]	Displays a summary of all processes with p_stat of SSLEEP or SXBRK. When an argument is given, its absolute value is used as a mask: 2 ignores processes in **wait()**; 4 ignores processes without upages; 8 ignores processes on a sleep semaphore.
ubt *slot*	Display a backtrace of the call stack of the sleeping process in the specified slot.
user [*addr* | *slot*]	Display the user area associated with the process specified either by address (negative number) or process table slot number (positive number)

Commands to Display Locks and Semaphores

The commands summarized in Table 11-8 display the state of semaphores and locks of different kinds, including metering information when the metered-lock module is included in the kernel.

Table 11-8 Commands to Display Locks and Semaphores

Command	Operation
;ock *addr*	Display the state of the spinlock at *addr*. This command is available only in multiprocessor systems.
mrlock *addr*	Display the state of the reader/writer lock at *addr*.
mutex *addr*	Display the state of the mutual exclusion lock at *addr*.
sema *addr*	Display the state of the semaphore at *addr*.
smeter *addr*	Display metering information about the semaphore at *addr*. When *addr* is positive, it is taken as an index to the semaphore metering array.
sv *addr*	Display the state of the synchronizing variable at *addr*, including waiting processes and metering information.

Commands to Display I/O Status

The commands summarized in Table 11-9 can be used to display the status of an I/O device or driver.

Table 11-9 Commands to Display I/O Status

Command	Operation
file [*addr*]	When *addr* is omitted, displays a summary of all entries of the kernel table of open files. When *addr* is the address of a file structure, displays only that entry.
scsi *addr*	Display the contents of the *scsi_request* structure at *addr*.
uio *addr*	Display the contents of the *uio_t* object at *addr*.

Commands to Display buf_t Objects

The commands summarized in Table 11-10 are used to display the state of *buf_t* objects and the queue of *buf_t* objects maintained by the kernel.

Table 11-10 Commands to Display buf_t Objects

Command	Operation
buf [*addr*]	If *addr* is omitted, print the entire buffer chain. When *addr* is supplied as the address of a buf_t, dump that structure.
findbuf *blkno*	Display any *buf_t* in the buffer chain with *b_blkno* containing *blkno*.
qbuf *eminor*	Find and display all *buf_t* objects that are queued to the device with external minor number *eminor*.

Commands to Display STREAMS Structures

The commands summarized in Table 11-11 are concerned with displaying STREAMS data structures such as message buffers.

Table 11-11 Commands to Display STREAMS Structures

Command	Operation
datab *addr*	Display the contents of the STREAMS data block at *addr*.
mbuf *addr*	Display the contents of the STREAMS *mbuf* structure at *addr*.
modinfo *addr*	Display the contents of the module info structure at *addr*.
msgb *addr*	Display the contents of the STREAMS message block at *addr*.
qband *addr*	Display the contents of the *qband_t* object at *addr*.
qinfo *addr*	Display the contents of the *qinit* structure at *addr*.
strh *addr*	Display the contents of the *stdata* structure at *addr*.
strfq *addr*	Display the contents of the *queue_t* object at *addr*.

Commands to Display Network-Related Structures

The commands summarized in Table 11-12 display data structures that are related in one way or another to networking and network device drivers.

Table 11-12 Commands to Display Network-Related Structures

Command	Operation
ifnet *addr*	Display the contents of the ifnet object at *addr*.
rawcb *addr*	Display the contents of the *rawcb* structure at *addr*.
rawif *addr*	Display the contents of the *rawif* structure at *addr*.
sock *addr*	Display the *sockbuf* structure at *addr*. When *addr* is positive, it is taken as a physical address; otherwise it is a kernel address.

Using icrash

The *icrash* program is a post-mortem analysis tool for system crashes. You can use icrash to generate a wide variety of reports and displays based on a kernel panic dump from a crashed system. Study the icrash(1M) reference page for the current release. For example, you can display the *putbuf* message buffer using the *stat* command of *icrash*.

Chapter 12

Driver Example

This chapter displays the code of a complete device driver. The driver implements a "RAM drive," a block of memory that simulates a disk drive. Since it has no hardware dependencies, this example driver can be used for experimentation in any IRIX system.

Note: It is not sensible to use a RAM drive in a system like IRIX, where there is an effective implementation of virtual memory. The RAM drive only occupies memory that is better used as buffers for the paging system. This driver is useful only as a test-bed for experiments with the driver-kernel interface and with *symmon* and other debugging tools. Do not use this driver in a production system.

Installing the Example Driver

Use the following steps to install and test the example driver. Each step is expanded in the following topics.

1. Obtain the source code files.
2. Compile the source to obtain an object file.
3. Set up the appropriate configuration files.
4. Reboot the system and verify driver operation.
5. Install special device files and make and mount a filesystem.

Obtaining the Source Files

The example driver consists of the following four source files:

ramdrive.c	Source code of the executable module.
ramdrive.h	Header file containing a few declarations.
ramdrive	Descriptive file to place in */var/sysgen/master.d*
ramdrive.sm	Example VECTOR line for */var/sysgen/system*

These files (and other example code from this book) are available from the Silicon Graphics FTP server, ftp://ftp.sgi.com/sgi/

Alternatively, a patient person could recreate the files by copying and pasting from the online manual. However, owing to bugs in the Insight viewer support for X-paste, this is an error-prone process and is not recommended in the current release.

Compiling the Example Driver

Compile *ramdrive.c* using the techniques described under "Compiling and Linking" on page 228. An example compilation is shown in Example 12-1,

Example 12-1 Compiling the Example Driver for a 32-bit Kernel

```
% set CFLAGS = "-DDEBUG -D_K32U32 -D_KERNEL -DSTATIC=static
-D_PAGESZ=4096 -D_MIPS2 -DIP22 -DR4000 -G 0 -non_shared -elf -xansi
-fullwarn -32 -mips2 -Wc,-pic0"
% cc $CFLAGS  -c ramdrive.c
```

When the driver is compiled with the -DDEBUG option, all its informational displays are enabled. Without that option, it only displays messages related to serious errors during initialization.

Configuring the Example Driver

Before you configure the example driver into the kernel, you should set the system with a debugging kernel, as described under “Preparing the System for Debugging” on page 243.

Configure the example driver to IRIX by copying files as follows:

- Copy the *ramdrive.o* file to */var/sysgen/boot*.
- Edit the *ramdrive* descriptive file and make any necessary changes. For example:
  - The file specifies major device number 77. If there is another driver in the system using this major number, change 77 to any unused number in the range 60-79 (see “Major Device Number” on page 34).
  - The fourth option, #DEV, is set to 4. This controls how many unique minor devices the driver supports.
- Copy the edited *ramdrive* descriptive file to */var/sysgen/master.d*.
- Edit the *ramdrive.sm* file and make any desired changes. For example,
  - Change a *base*= value to change the size of a RAM drive
  - Add or remove VECTOR lines to change the number of minor devices that are initialized.
- Copy the edited *ramdrive.sm* file to */var/sysgen/system*.
- Reboot the system.

If you compiled the example driver with -DDEBUG, it displays several informational lines to the system console from its **rd_init()**, **rd_start()**, and **rd_edtinit()** entry points. The display from **rd_edtinit()** gives the address of the *rd_info_t* structure. You can display that area with *idbg*. Example 12-2 shows the result of displaying a simulated volume header structure, using an address displayed from **rd_edtinit()** (the actual address will differ).

Example 12-2 Displaying Simulated Volume Header Using idbg

```
# idbg hd 0x8841f604 0x80
0x8841f604: 0b e5 a9 41  00 00 00 01  00 00 00 00  00 00 00 00 ...A............
0x8841f614: 00 00 00 00  00 00 00 00  00 00 00 00  02 00 00 00 ................
0x8841f624: 00 01 00 00  00 00 00 08  02 00 00 01  00 00 00 00 ................
0x8841f634: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f644: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f654: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f664: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f674: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f684: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f694: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f6a4: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f6b4: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f6c4: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f6d4: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f6e4: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f6f4: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f704: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f714: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f724: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f734: 00 00 00 00  00 00 00 00  00 00 0f ff  00 00 00 01 ................
0x8841f744: 00 00 00 03  00 00 00 00  00 00 10 00  00 00 00 03 ................
0x8841f754: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f764: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f774: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f784: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 0f ff ................
0x8841f794: 00 00 00 01  00 00 00 03  00 00 00 01  00 00 00 00 ................
0x8841f7a4: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f7b4: 00 00 10 00  00 00 00 00  00 00 00 06  00 00 00 00 ................
0x8841f7c4: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f7d4: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f7e4: 00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00 ................
0x8841f7f4: 00 00 00 00  00 00 00 00  f0 19 16 a5  00 00 00 00 ................
```

Creating Device Special Files

For each VECTOR line in */var/sysgen/system/ramdrive.sm* you can create a pair of device special files, one block device and one character device. Each file's inode contains the major device number established in descriptive file */var/sysgen/master.d/ramdrive*, and a minor number between 0 and 1 less than the limit established in the descriptive file.

The preferred command is *install* (see the install(1) reference page). The commands in Example 12-3 show the creation of the character and block devices for minor device number 0. The files created are */dev/ramblk0* and */dev/ramchr0*.

Example 12-3 Install Command to Create Device Special File

```
# install -u root -g sys -f /dev -blk 77,0 ramblk0
# install -u root -g sys -f /dev -chr 77,0 ramchr0
```

In addition to device special files, you need to create ordinary directories that can be used as mount points for the mounted filesystem, for example

```
# mkdir /RAM0
```

The example driver supports as many minor device numbers as you specify in the descriptive file */var/sysgen/master.d/ramdrive*. You can create as many device special files that contain the example driver's major number as you like—or as few. Device special files are independent of the driver configuration until they are opened. However,

- A device with a minor number in excess of the limit set in */var/sysgen/master.d/ramdrive* cannot be initialized with a VECTOR line in */var/sysgen/system/ramdrive.sm*, nor can it be opened.
- A device whose minor number was not initialized by a VECTOR line cannot be opened.

Verifying Driver Operation

You can verify operation of the driver by creating an EFS file system on one of its devices. As a first step, check the simulated volume header contents using *prtvtoc*, as shown in Example 12-4.

Example 12-4 Applying prtvtoc to a RAM Drive of 2 MB

```
# prtvtoc /dev/ramchr0
ramdrive open: flag 1 copen 1 bopen 0 xopen 0
DIOCGETVH on 20185088
ioctl copy kmem 8841f604 -> usr 10006868 for 512
ramdrive close: flag 1 copen 0 bopen 0 xopen 0
* /dev/ramchr0 (bootfile "")
*     512 bytes/sector
*       8 sectors/track
*       1 tracks/cylinder
*     512 cylinders
*       0 cylinders occupied by header
*     512 accessible cylinders
* No space unallocated to partitions
Partition  Type  Fs   Start: sec   (cyl)    Size: sec    (cyl)  Mount
Directory
 0         raw               1 (   0.1)        4095  ( 511.9)
 7         raw               1 (   0.1)        4095  ( 511.9)
 8      volhdr               0  (   0)           1  (   0.1)
10      volume               0  (   0)         4096   ( 512)
```

You can make an EFS filesystem on a RAM drive device by applying *mkfs* to the character special device (*mkfs* is applied to the character device because it uses **ioctl()**, which is not supported for block devices). This is shown in Example 12-5. The voluminous debugging displays have been truncated in the display.

Example 12-5 Making a Filesystem on a RAM Drive

```
# mkfs -t efs /dev/ramchr0
ramdrive open: flag 1 copen 1 bopen 0 xopen 0
DIOCGETVH on 20185088
ioctl copy kmem 8841f604 -> usr 10010c50 for 512
ramdrive close: flag 1 copen 0 bopen 0 xopen 0
ramdrive open: flag 1 copen 1 bopen 0 xopen 0
DIOCGETVH on 20185088
ioctl copy kmem 8841f604 -> usr 10010c50 for 512
ramdrive close: flag 1 copen 0 bopen 0 xopen 0
ramdrive open: flag 1 copen 1 bopen 0 xopen 0
DIOCGETVH on 20185088
ioctl copy kmem 8841f604 -> usr 10010c50 for 512
ramdrive close: flag 1 copen 0 bopen 0 xopen 0
ramdrive open: flag 1 copen 1 bopen 0 xopen 0
ramdrive close: flag 1 copen 0 bopen 0 xopen 0
ramdrive open: flag 3 copen 1 bopen 0 xopen 0
```

```
mkfs_efs: /dev/ramchr0: blocks=4095 inodes=616
mkfs_efs: /dev/ramchr0: sectors=8 cgfsize=4091
mkfs_efs: /dev/ramchr0: cgalign=1 ialign=1 ncg=1
mkfs_efs: /dev/ramchr0: firstcg=3 cgisize=154
mkfs_efs: /dev/ramchr0: bitmap blocks=1
rd_write entered for dev 20185088
rd_strategy: edev 20185088, flags 8018, blkno 3
           : offset 600, count 13400, dmaadr c026b610
        : write c026b610 to c0000600 for 13400
rd_write entered for dev 20185088
rd_strategy: edev 20185088, flags 18, blkno 0
           : offset 0, count 200, dmaadr 8b292cc0
        : write 8b292cc0 to c0000000 for 200
rd_read entered for dev 20185088
rd_strategy: edev 20185088, flags 19, blkno 3
           : offset 600, count 200, dmaadr c2c38a40
        : read c0000600 to c2c38a40 for 200
(many debugging displays omitted)
rd_write entered for dev 20185088
rd_strategy: edev 20185088, flags 8018, blkno ffe
           : offset 1ffc00, count 200, dmaadr c022def0
        : write c022def0 to c01ffc00 for 200
ramdrive close: flag 3 copen 0 bopen 0 xopen 0
```

After making a filesystem you can mount the filesystem, as shown in Example 12-6. You specify the block device when mounting an EFS filesystem, because the filesystem code calls the *pfx***size()** and *pfx***strategy()** driver entry points, which are supplied only by a block driver.

Example 12-6 Mounting a RAM Drive Filesystem

```
# mount -t efs /dev/ramblk0 /RAM0
ramdrive open: flag 3 copen 0 bopen 1 xopen 0
rd_size entered for dev 20185088
rd_strategy: edev 20185088, flags 9, blkno 1
           : offset 200, count 200, dmaadr 889f5800
        : read c0000200 to 889f5800 for 200
rd_strategy: edev 20185088, flags 9, blkno 2
           : offset 400, count 200, dmaadr 889f5a00
        : read c0000400 to 889f5a00 for 200
rd_strategy: edev 20185088, flags 8, blkno 2
           : offset 400, count 200, dmaadr 889f5a00
        : write 889f5a00 to c0000400 for 200
```

```
rd_strategy: edev 20185088, flags 9, blkno 3
           : offset 600, count 1000, dmaadr 88ef6000
        : read c0000600 to 88ef6000 for 1000
rd_strategy: edev 20185088, flags 1000008, blkno 1
           : offset 200, count 200, dmaadr 889f5800
        : write 889f5800 to c0000200 for 200
# df /RAM0
Filesystem             Type  blocks     use     avail  %use Mounted on
/dev/ramblk0            efs     3937       22     3915   1  /RAM0
```

Example Driver Source Files

The four source files of the example driver are displayed in the following topics:

- "Descriptive File" on page 278 displays the */var/sysgen/master.d* file that describes the driver to *lboot*.
- "System File" on page 279 displays the */var/sysgen/system* file that contains the VECTOR statements to initialize the driver.
- "Header File" on page 279 displays the driver's header file.
- "Source File" on page 281 displays the source file.

Descriptive File

```
*
* IRIX 6.2 Example driver "ramdrive" -- not for production use
*
* Flags used:
* b: block type device
* c: character type device (yes, both)
* d: dynamically loadable kernel module
* n: driver is semaphored
* s: software device driver
* w: driver is prepared to perform any cache write back operation
*
* External major number (SOFT) is an arbitrary choice from
* the range of numbers reserved for customer drivers.
*
* #DEV is passed in to the driver and used to configure its info array.
*
```

```
*FLAG   PREFIX  SOFT    #DEV    DEPENDENCIES
*bcdnsw  rd_     77      4
bcnsw  rd_     77      4

$$$

int rd_e_major = ##E;
int rd_numdevs = ##D;
int rd_ctrlrs = ##C;
```

System File

```
*
* Lboot config file for IRIX 6.2 example driver "ramdrive"
*   base= size of RAM drive, e.g. 0x00200000 is 2MB
*   ctlr= minor device number, 0 to 3
* Store as /var/sysgen/system/ramdrive.sm
*
VECTOR: module=ramdrive ctlr=0 base=0x00100000
*VECTOR: module=ramdrive ctlr=1 base=0x00080000
```

Header File

```
/**************************************************************************
 *                                                                        *
 *               Copyright (C) 1993, Silicon Graphics, Inc.               *
 *                                                                        *
 *  These coded instructions, statements, and computer programs  contain  *
 *  unpublished  proprietary  information of Silicon Graphics, Inc., and  *
 *  are protected by Federal copyright law.  They  may  not be disclosed  *
 *  to  third  parties  or copied or duplicated in any form, in whole or  *
 *  in part, without the prior written consent of Silicon Graphics, Inc.  *
 *                                                                        *
 **************************************************************************/

/**************************************************************************
| This is ramdrive.h, containing declarations that are used in both the
| driver and in the unit-test application code.  Aside from the unit-test
| module, no user-level code ever needs these declarations.  The ram drive
| is accessed through the file system like any other disk.
**************************************************************************/

/**************************************************************************
```

```
| The driver name for lboot and configuration is "ramdrive."
| The driver prefix is "rd_"
*************************************************************************/

#define DRIVER_PFX "rd_"
#define DRIVER_NAME "ramdrive"

/************************************************************************
| MAX_RD_DEVS declares the maximum number of distinct devices supported.
| Ram drive device special files are /dev/dsk/ramblk<n> for block devices
| and /dev/rdsk/ramchr<n> for character devices.  In each case <n> is the
| device minor number, between 0 and MAX_RD_DEVS-1.
************************************************************************/

#define MAX_RD_DEVS 4

/************************************************************************
| An array of MAX_RD_DEVS structures of the following type is maintained
| in the driver. VECTOR lines for up to MAX_RD_DEVS devices are written
| in /var/sysgen/system/ramdrive.sm, causing that many entries to the
| rd_edtinit() entry point, each entry initializing one structure.
|
| base:    address of allocated memory for the “drive."  If NULL, this
|          minor number has not been initialized or failed initialization.
|
| size:    size of the allocated memory in bytes, always rounded down to
|          a multiple of IO_NBPP.
|
| copen:   count of successful character opens, cleared to 0 in rd_close().
| bopen:   count of successful block opens (0 or 1), cleared in rd_close().
| xopen:   nonzero when an FEXCL open has succeeded.
|
| nmmap:   count of rd_map() entries, decremented in rd_unmap().  When the
|          any of copen, bopen, and nmaps over all devices is nonzero, the
|          driver returns EBUSY to the rd_unload() entry point.
|
| queue:   semaphore used to serialize access for reading and writing.
|          (Note however that in a multiprocessor, a user process can
|          perform an unsynchronized write to a mapped character device.)
|
| vh:      volume header structure prepared in edtinit(), and used to
|          initialize block 0 when “formatting" the drive.  The block-0
|          version can be modified by /etc/mkfs.
************************************************************************/
```

```
typedef struct rd_info {
    caddr_t     *base;
    off_t       size;
    __uint32_t  copen, bopen, xopen, nmmap;
    sema_t      queue;  /* requires sys/sema.h */
    struct volume_header vh; /* requires sys/dvh.h */
} rd_info_t;
```

Source File

```
/***************************************************************************
 *                                                                         *
 *                  Copyright (C) 1993, Silicon Graphics, Inc.             *
 *                                                                         *
 *  These coded instructions, statements, and computer programs  contain  *
 *  unpublished  proprietary  information of Silicon Graphics, Inc., and  *
 *  are protected by Federal copyright law.  They  may  not be disclosed  *
 *  to  third  parties  or copied or duplicated in any form, in whole or  *
 *  in part, without the prior written consent of Silicon Graphics, Inc.  *
 *                                                                         *
 **************************************************************************/
/***************************************************************************
| This sample IRIX device driver implements a “ram disk" -- a block of
| kernel memory accessed as if it were a disk.  The driver supports both
| block and character interfaces and is loadable and unloadable.
|
| N   N   OO   TTTTT  EEEEE      It does not make sense to use a ram disk
| NN  N  O  O    T    E          in a system like IRIX that implements
| N N N  O  O    T    EEE   ::   effective virtual memory.  This device
| N  NN  O  O    T    E          driver is useful as an example because
| N   N   OO     T    EEEEE ::   it has no hardware dependencies, and so
|                                can be tried out in any IRIX system.
| However, this driver SHOULD NOT be employed in a production system!
| It WILL NOT give better performance.  It WILL consume kernel memory
| that would be better used for buffers.
|**************************************************************************/

#include <sys/ddi.h>        /* gets also sys/types.h and sys/buf.h */
#include <sys/conf.h>       /* for driver flags D_MP etc */
#include <sys/kmem.h>       /* kmem_alloc and friends */
#include <sys/sema.h>       /* the rd_info_t contains a semaphore */
#include <sys/dvh.h>        /* the rd_info_t contains struct volume_header */
#include “ramdrive.h"       /* declare rd_info_t, etc. */
#include <sys/edt.h>        /* declare edt_t for edtinit() */
```

```
#include <sys/errno.h>      /* error codes to return */
#include <sys/cmn_err.h>    /* cmn_err() and related constants */
#include <sys/cred.h>       /* cred_t for prototypes */
#include <sys/dkio.h>       /* DKIOC* constants for ioctl */
#include <sys/param.h>      /* NBPSC bytes per sector */
#include <sys/immu.h>       /* IONBPP bytes per I/O page, btod() */
#include <sys/file.h>       /* FEXCL and other open flags */
#include <sys/open.h>       /* OTYP_CHR, OTYP_BLK */
#include <sys/region.h>     /* for vhandl_t */
#include <sys/mload.h>      /* only for M_VERSION */
/***************************************************************************
| Debug display macros: one each for cmn_err calls with 0, 1, 2, or 3
| variable arguments.
|***************************************************************************/
#ifdef DEBUG
#define DBGMSG0(s) cmn_err(CE_DEBUG,s)
#define DBGMSG1(s,x) cmn_err(CE_DEBUG,s,x)
#define DBGMSG2(s,x,y) cmn_err(CE_DEBUG,s,x,y)
#define DBGMSG3(s,x,y,z) cmn_err(CE_DEBUG,s,x,y,z)
#define DBGMSG4(s,x,y,z,w) cmn_err(CE_DEBUG,s,x,y,z,w)
#else
#define DBGMSG0(s)
#define DBGMSG1(s,x)
#define DBGMSG2(s,x,y)
#define DBGMSG3(s,x,y,z)
#define DBGMSG4(s,x,y,z,w)
#endif
/***************************************************************************
| Driver flag: this driver is MP-safe.  Also version flag for mload.
|***************************************************************************/
unsigned rd_devflag = D_MP;
char *rd_mversion = M_VERSION;
/***************************************************************************
| Array of rd_info_t objects, one per allowed minor device.  We rely on
| the loader to ensure these static globals are zero until initialized!
| Also defined: two convenience macros for frequent expressions.
|***************************************************************************/
static rd_info_t *rd_array;
#define INFOPTR(dev) &rd_array[geteminor(dev)]
#define VALIDIO(prd,off,len) (((off_t)(off) + (off_t)(len)) <= prd->size)
/***************************************************************************
| rd_basic() is called from rd_edtinit() to allocate the rd_array based
| on the global rd_numdevs, an integer set to ##D in the configuration
| file /var/sysgen/master.d/ramdrive.  Also display the other available
| globals for debugging purposes.
```

```
|**************************************************************************/
extern int rd_e_major, rd_numdevs, rd_ctrlrs;
int
rd_basic(void)
{
    if (!rd_array)
    {
        register int size;
        DBGMSG3("ramdrive basic: ##E=%d, ##D=%d, ##C=%d\n",
            rd_e_major, rd_numdevs, rd_ctrlrs);
        if (size = rd_numdevs*sizeof(rd_info_t))
            rd_array = (rd_info_t *)kmem_zalloc(size,KM_SLEEP);
        else
            cmn_err(CE_ALERT,"ramdrive: confused");
    }
    return (0 != rd_array);
}
/**************************************************************************
| rd_init() is included solely to demonstrate that this entry point
| can be called in addition to rd_edtinit() and rd_start().
|**************************************************************************/
int
rd_init(void)
{
    DBGMSG0("rd_init entry point called\n");
    return 0;
}
/**************************************************************************
| rd_start() is included solely to demonstrate that it, too can be called
| in addition to rd_edtinit() and rd_init().
|**************************************************************************/
int
rd_start(void)
{
    DBGMSG0("rd_start entry point called\n");
    return 0;
}
/**************************************************************************
| rd_format() is a subroutine of both rd_edtinit() and rd_ioctl() which
| "formats" the ramdrive to zeros with a reasonable volume header.
| The volume header (set in both the info struct and "sector 0")
| describes standard SGI partitions:
|   10 == the whole "drive"
|    8 == the volume header, only one sector in this case
|    7 == all sectors except the volume header
|    0 == data ("root") same as 7
|    1 == swap contains 0 sectors
| For versimilitude we arbitrarily say we have 1 track/cylinder
| and 8 sectors/track.  This assumes that nsectors is a multiple of 8,
| which is a good bet when the allocated size is a multiple of IO pages
| and sectors are 512 bytes.
|***********************************************************************/
void
rd_format(register rd_info_t *prd)
{
    register struct volume_header *pvh = &prd->vh;
    register int nsectors = btod(prd->size);/* immu.h */

    bzero((void *)pvh,sizeof(struct volume_header));
    pvh->vh_magic = VHMAGIC; /* in sys/dvh.h */
    pvh->vh_rootpt = 0;
    pvh->vh_swappt = 1;
    pvh->vh_dp.dp_cyls = nsectors/8; /* number of cylinders */
    pvh->vh_dp.dp_trks0 = 1; /* tracks/cyl */
    pvh->vh_dp.dp_secs = 8; /* sectors/track */
    pvh->vh_dp.dp_secbytes = NBPSCTR; /* param.h */
    pvh->vh_dp.dp_interleave = 1;

    pvh->vh_pt[10].pt_firstlbn = 0;
    pvh->vh_pt[10].pt_nblks = nsectors;
    pvh->vh_pt[10].pt_type = PTYPE_VOLUME;
    pvh->vh_pt[ 8].pt_firstlbn = 0;
    pvh->vh_pt[ 8].pt_nblks = 1;
    pvh->vh_pt[ 8].pt_type = PTYPE_VOLHDR;
    pvh->vh_pt[ 8].pt_firstlbn = 0;
    pvh->vh_pt[ 7].pt_firstlbn = 1;
    pvh->vh_pt[ 7].pt_nblks = nsectors-1;
    pvh->vh_pt[ 7].pt_type = PTYPE_RAW;
    pvh->vh_pt[ 0] = pvh->vh_pt[ 7];
    pvh->vh_pt[ 1].pt_firstlbn = nsectors;
    pvh->vh_pt[ 1].pt_nblks = 0;
    pvh->vh_pt[ 1].pt_type = PTYPE_RAW;

    pvh->vh_csum = -vh_checksum(pvh);
    bzero(prd->base,prd->size); /* clear all sectors */
    bcopy(pvh,prd->base,sizeof(prd->vh)); /* vh in sec 0 */
}
/***********************************************************************
| rd_edtinit() is called whenever the driver is loaded, once for each
| VECTOR that names this driver.  A typical VECTOR line would be:
```

```
|     VECTOR module=ramdrive ctrl=2 base=0x00040000
| which says, initialize minor number 2 for a size of 256K.
|************************************************************************/
int
rd_edtinit(register edt_t *pedt)
{
    register rd_info_t *prd;
    register __psint_t size;
    register int nsectors;
    register int ctlr = pedt->e_ctlr;
    /*
    || If this is the first time, allocate the rd_array of info structures.
    || Exit immediately if that fails.
    */
    if (!rd_basic())
    {
        return ENODEV;
    }
    DBGMSG3("ramdrive edtinit bustype %d adap %d ctlr %d\n",
        pedt->e_bus_type, pedt->e_adap, pedt->e_ctlr);
    DBGMSG3("                 e_space[0] iopaddr %x size %x vaddr %x\n",
        pedt->e_space[0].ios_iopaddr,pedt->e_space[0].ios_size,
        pedt->e_space[0].ios_vaddr);
    /*
    || Diagnose and reject an invalid minor dev# from VECTOR ctlr=
    */
    if (ctlr > rd_numdevs)
    {
        cmn_err(CE_ALERT,"ramdrive: ctlr=%d invalid minor dev#",ctlr);
        return ENODEV;
    }
    /*
    || Address the info structure and diagnose multiple initialization
    */
    prd = INFOPTR(ctlr);
    if (prd->base)
    {
        cmn_err(CE_ALERT,"ramdrive: duplicate VECTOR for ctlr=%d",ctlr);
        return EBUSY;
    }
    /*
    || The desired size of the ramdrive is encoded as the base=# value,
    || which is passed as the ios_vaddr value in the edt_t.
    || Diagnose 0 size (omitted base=). Round the size to a
    || multiple of *memory* (not necessarily I/O) pages.
    */
    size = (__psint_t) pedt->e_space[0].ios_vaddr;
    if ((0 == size)||(-1 == size))
    {
        cmn_err(CE_ALERT,
        "ramdrive: no size (base=) specified for ctlr=%d",ctlr);
        return EINVAL;
    }
    size = (size + (NBPP-1)) & (-NBPP); /* in sys/immu.h */
    /*
    || Allocate the kernel memory. Report an error if not possible.
    */
    prd->size = size;
    prd->base = kmem_alloc(size,KM_SLEEP);
    if (!prd->base)
    {
        cmn_err(CE_ALERT,
        "ramdrive: unable to allocate %x bytes for dev %d",size,ctlr);
        return ENOMEM;
    }
    nsectors = btod(size); /* immu.h bytes to disk sectors */
    DBGMSG3("ramdrive: dev# %d allocated %x = %x sectors\n",
                ctlr,size,nsectors);
    /*
    || Initialize the semaphore.
    */
    initnsema(&prd->queue,1,"ramdrive");
    /*
    || Initialize the "volume."
    */
    rd_format(prd);
    DBGMSG2("              info at 0x%x  vh at 0x%x \n",
        prd, (__psint_t)(&prd->vh) );
    return 0;
}
/***************************************************************************
| rd_open() is called for each open() of a character device /dev/ramchr<n>,
| and during a mount of a block device /dev/ramblk<n>.  We can distinguish
| between types of open from the otyp.
|***************************************************************************/
int
rd_open(dev_t *pdev, int oflag, int otyp, cred_t *pcred)
{
    register rd_info_t *prd = INFOPTR(*pdev);
    register int error = 0;
```

```
/*
|| Make sure the device being opened was initialized by a VECTOR.
*/
if (!prd->base)
{
    cmn_err(CE_NOTE,"ramdrive: open of uninitialized dev %d",*pdev);
    return ENODEV;
}
/*
|| Seize the device semaphore so that prd->rd_info can be updated
|| without error on a multiprocessor.
*/
psema(&prd->queue,PZERO+1 | PCATCH);
/*
|| Implement FEXCL (exclusive) open for a privileged process only.
|| Exclusivity applies to the entire minor device, under both its
|| block and character special devices.
*/
if (oflag & FEXCL)
{
    if (drv_priv(pcred)) /* not privileged */
    {
        DBGMSG0("ramdrive: reject FEXCL with EPERM\n");
        error = EPERM;
    }
    else if (prd->copen+prd->bopen+prd->nmmap) /* current use? */
    {
        DBGMSG0("ramdrive: reject FEXCL with EBUSY\n");
        error = EBUSY;
    }
    else
    {
        prd->xopen = oflag; /* note device open exclusively */
    }
}
else /* nonexclusive request can be blocked by exclusive open */
{
    if (prd->xopen)
    {
        DBGMSG0("ramdrive: reject normal open for exclusivity\n");
        error = EBUSY;
    }
}
if (!error)
{
        /*
        || Count the open so we don't unload with open devices.
        */
        if (otyp & OTYP_CHR)
            ++prd->copen;
        else
            ++prd->bopen;
        DBGMSG4("ramdrive open: flag %x copen %d bopen %d xopen %d\n",
            oflag, prd->copen, prd->bopen, prd->xopen);
    }
    vsema(&prd->queue);
    return error;
}
/***************************************************************************
| rd_close() is not called for each close() but for the final close of a
| given device (character or block).  Clear the respective count of opens
| and note whether exclusivity is being given up.  Since a close() in
| one CPU could happen concurrently with an open() in another CPU, we
| need to grab the semaphore before updating the rd_info.
| NOTE: the flag passed to close does not contain FEXCL even if it was
| given in the flag passed to open.
|***************************************************************************/
int
rd_close(dev_t dev, int flag, int otyp, cred_t *pcred)
{
    register rd_info_t *prd = INFOPTR(dev);
    psema(&prd->queue,PZERO+1 | PCATCH);
    if (flag & FEXCL)
    {
        /* this is never entered */
    }
    if (otyp & OTYP_CHR)
    {
        prd->copen = 0;
    }
    else
    {
        prd->bopen = 0;
    }
    /* if all opens are closed, an exclusive one is closed */
    prd->xopen = 0;
    vsema(&prd->queue);
    DBGMSG4("ramdrive close: flag %x copen %d bopen %d xopen %d\n",
            flag, prd->copen, prd->bopen, prd->xopen);
    return 0;
```

```
}
/*************************************************************************
| rd_ioctl() is called for ioctl(2), which can only be used on a character
| device. Disk ioctl command numbers for are in sys/dkio.h.
| DIOCREADCAPACITY: supported just for fun.
| DIOCGETVH: supported because /etc/mkfs and other tools use it (which
| explains why you apply mkfs to the character, not the block, device).
| DIOCSETVH: allows a program to change the "volume header" info.
| DIOCFORMAT: clears the device contents to 0, rewrites the vol header.
|
| The DIOC(S|G)ETVH calls use only the info in the per-device structure
| in memory. We make no attempt to keep that info in step with the
| contents of sector 0 of the simulated media.  This is consistent with
| other current IRIX disk drivers.  This has the implications that:
|   - you can change the driver's idea of the disk geometry on the fly,
|     without actually formatting the disk, this is useful for scsi.
|   - if you want to make a permanent change in the volume header,
|      -- one, that's a bad idea, use dvhtool(1) instead, but
|      -- two, if you insist, you need both a write to sector 0 and
|         a call to ioctl(,DIOCSETVH) to keep the driver up to date.
|
| Neither DIOCSETVH nor DIOCFORMAT hold the semaphore. You are strongly
| advised to do an exclusive open before calling them (but mkfp doesn't).
|************************************************************************/
int
rd_ioctl(dev_t dev, int cmd, caddr_t arg, int mode, cred_t *pcred, int *rval)
{
    register rd_info_t *prd = INFOPTR(dev);
    register int error = 0;
    register caddr_t kmemadr;
    register int len = 0;
    register int dir = 0; /* copyout */
    int capacity;
    switch(cmd)
    {
    case DIOCGETVH:
        {
            kmemadr = (caddr_t)(&prd->vh);
            len = sizeof(prd->vh);
            DBGMSG1("DIOCGETVH on %d\n",dev);
            break;
        }
    case DIOCREADCAPACITY:
        {
            capacity = prd->size/NBPSCTR;
            kmemadr = (caddr_t)(&capacity);
            len = sizeof(capacity);
            DBGMSG2("DIOCREADCAPACITY on %d = %d\n",
                  dev,capacity);
            break;
        }
    case DIOCSETVH:
        {
            kmemadr = (caddr_t)(&prd->vh);
            len = sizeof(prd->vh);
            dir = 1; /* copyin */
            DBGMSG1("DIOCSETVH on %d done\n",dev);
            break;
        }
    case DIOCFORMAT:
        {
            rd_format(prd);
            DBGMSG1("DIOCFORMAT done on %d!\n",dev);
            break;
        }
    default:
        {
            DBGMSG2("ramdrive invalid ioctl %x on %d\n",cmd,dev);
            error = EINVAL;
        }
    } /* switch(cmd) */
    /*
    || Perform the copy to or from user space if needed.
    */
    if ((!error) && (len))
    {
        if (!dir)
        {
            DBGMSG3("ioctl copy kmem %x -> usr %x for %d\n",
                kmemadr, arg, len);
            error = copyout(kmemadr,arg,len);
        }
        else
        {
            DBGMSG3("ioctl copy usr %x -> kmem %x for %d\n",
                arg, kmemadr, len);
            error = copyin(arg,kmemadr,len);
        }
#ifdef DEBUG
        if (error)
```

```
            DBGMSG1("error %d on ioctl copy\n",error);
#endif
    }
    *rval = error; /* ensure user gets correct code */
    return error;
}
/***************************************************************************
| I/O Operations:
|
| rd_strategy() performs all actual I/O.  Called directly by file systems
| to read and write full I/O page units aligned on I/O page boundaries.
| Called indirectly to implement character I/O in any length and alignment.
|
| rd_read() and rd_write are called by read()/write() to a character
| device. They defer to rd_strategy via uiophysio().  This is consistent
| with the operation of other IRIX disk drivers.
|
| The strategy code simply does a bcopy. This is highly unrealistic.
| A real device driver would have to deal with efficient sequencing of
| track numbers and with asynchronous interrupts.
|***************************************************************************/
int
rd_strategy(register struct buf *pbuf)
{
    register rd_info_t *prd = INFOPTR(pbuf->b_edev);
    register __psint_t offset = pbuf->b_blkno * NBPSCTR;
    register __psint_t count = pbuf->b_bcount;
    register caddr_t target = (caddr_t)((__psint_t)prd->base)+offset;
    DBGMSG3("rd_strategy: edev %d, flags %x, blkno %x\n",
                    pbuf->b_edev,pbuf->b_flags,pbuf->b_blkno);
    DBGMSG3("         : offset %x, count %x, dmaadr %x\n",
                    offset,count,(caddr_t)pbuf->b_dmaaddr);
    if (!VALIDIO(prd,offset,count))
    {
        DBGMSG0("rejecting strategy with ENOSPC\n");
        pbuf->b_error = ENOSPC;
        iodone(pbuf);
        return 0;
    }
    /*
    || Ensure that pbuf->b_dmaaddr is a valid kernel address.
    || This is never needed when called via uiophysio, only when
    || called from the file system or paging subsystem.  (Goodness!
    || wouldn't it be fun to use a ramdrive for swapping?)
    || NOTE: while a simple bp_mapin() call works, this approach
    || would impose unnecessary overhead in a real driver when
    || the device does not support scatter/gather.
    */
    if (!BP_ISMAPPED(pbuf))
    {
        bp_mapin(pbuf);
        DBGMSG1("        : after bp_mapin dmaadr %x\n", pbuf->b_dmaaddr);
    }
    /*
    || Grab the device semaphore. Note: this ensures consistency
    || between reads and writes, but does not control modifications
    || made through memory-mapped access.
    */
    psema(&prd->queue,PZERO+1 | PCATCH);
    /*
    || Perform the "read" or "write."
    */
    if (pbuf->b_flags & B_READ)
    {
        DBGMSG3("        : read %x to %x for %x\n",
            target,pbuf->b_dmaaddr,pbuf->b_bcount);
        bcopy(target,pbuf->b_dmaaddr,pbuf->b_bcount);
    }
    else
    {
        DBGMSG3("        : write %x to %x for %x\n",
            pbuf->b_dmaaddr,target,pbuf->b_bcount);
        bcopy(pbuf->b_dmaaddr,target,pbuf->b_bcount);
    }
    vsema(&prd->queue);
    iodone(pbuf);
    return 0;
}
int
rd_read(dev_t dev, uio_t *puio, cred_t *pcred)
{
    DBGMSG1("rd_read entered for dev %d\n",dev);
    return uiophysio(rd_strategy,0,dev,B_READ,puio);
}
int
rd_write(dev_t dev, uio_t *puio, cred_t *pcred)
{
    DBGMSG1("rd_write entered for dev %d\n",dev);
    return uiophysio(rd_strategy,0,dev,B_WRITE,puio);
}
```

```
int
rd_size(dev_t dev)
{
    DBGMSG1("rd_size entered for dev %d\n",dev);
    return rd_array[geteminor(dev)].size/NBPSCTR;
}
/*************************************************************************
| Memory mapping: rd_map() (one "m") is called to implement an mmap()
| request on a character device.  We permit read and write mappings, which
| means that in a multiprocessor, one CPU could be updating the kernel
| memory that represents the medium while another CPU executes a read()
| on the same memory.
|
| Since a map can persist after the corresponding FD is closed, we
| keep track of mappings separately from opens.
*************************************************************************/
int
rd_map(dev_t dev, vhandl_t *pvh, off_t off, int len, int prot)
{
    register rd_info_t *prd = INFOPTR(dev);
    int error;

    DBGMSG3("map request on %d at %x for %x\n",dev,off,len);
    if (VALIDIO(prd,off,len))
    {
        error = v_mapphys(pvh,prd->base+off,len);
#ifdef DEBUG
        if (error)
            DBGMSG1("v_mapphys returns %d\n",error);
#endif
    }
    else
    {
        DBGMSG0("rejecting map with ENOSPC\n");
        error = ENOSPC;
    }
    if (!error)
        ++prd->nmmap;
    return error;
}
rd_unmap(dev_t dev, vhandl_t *pvh)
{
    register rd_info_t *prd = INFOPTR(dev);
    if (prd->nmmap)
    {
        --prd->nmmap;
        DBGMSG2("unmap on %d, map count now %d\n",dev,prd->nmmap);
    }
    else
    {
        DBGMSG1("unmap on %d when map count 0 ?!?!?!?\n",dev);
    }
    return 0;
}
/*************************************************************************
| Unload support: rd_unload() is called when ml(1) is asked to unload
| this driver.  We test to make sure that none of our devices that have
| been initialized, are in use.  When any are in use, we return EBUSY
| and so will not be unloaded.
*************************************************************************/
int
rd_unload(void)
{
    int j;
    for (j = 0; j<rd_numdevs; ++j)
    {
        if (( rd_array[j].base )
        &&  ( rd_array[j].copen
            ||rd_array[j].bopen
            ||rd_array[j].nmmap) )
        {
            DBGMSG1("rejecting unload because dev %d busy\n",j);
            return EBUSY;
        }
    }
    DBGMSG0("accepting unload, byeeeee\n");
    return 0;
}
```

PART FOUR

VME Device Drivers

Chapter 13: VME Device Attachment
How the VME bus is configured in the Challenge and Power Challenge systems.

Chapter 14: Services for VME Drivers
Kernel functions available specifically to VME device drivers.

Chapter 13

VME Device Attachment

Several Silicon Graphics computer systems support the VME bus. This chapter gives a high-level overview of the VME bus, and describes how the VME bus is attached to, and operated by, each system.

This chapter contains useful background information if you plan to control a VME device from a user-level program. It contains important details on VME addressing if you are writing a kernel-level VME device driver.

- "Overview of the VME Bus" on page 298 summarizes the history and features of the VME bus architecture.
- "VME Bus in Silicon Graphics Systems" on page 300 gives an overview of how the VME bus is integrated into Silicon Graphics computer systems.
- "VME Bus Addresses and System Addresses" on page 304 discusses the relationship between addresses on the VME bus and addresses in the physical address space of the system.
- "Configuring VME Devices" on page 310 tells how to configure a device so that IRIX can recognize it and initialize its device driver.
- "VME Hardware in Challenge and Onyx Systems" on page 313 documents the hardware details of the VME implementation on those systems.

More information about VME device control appears in these chapters:

- Chapter 4, "User-Level Access to VME and EISA," covers PIO and DMA access from the user process.
- Chapter 14, "Services for VME Drivers," discusses the kernel services used by a kernel-level VME device driver, and contains an example.

Overview of the VME Bus

The VME bus dates to the early 1980s. It was designed as a flexible interconnection between multiple master and slave devices using a variety of address and data precisions, and has become a popular standard bus used in a variety of products. (For ordering information on the standards documents, see “Standards Documents” on page xxxviii.)

In Silicon Graphics systems, the VME bus is treated as an I/O device, not as the main system bus.

VME History

The VME bus descends from the VERSAbus, a bus design published by Motorola, Inc., in 1980 to support the needs of the MC68000 line of microprocessors. The bus timing relationships and some signal names still reflect this heritage, although the VME bus is used by devices from many manufacturers today.

The original VERSAbus design specified a large form factor for pluggable cards. Because of this, it was not popular with European designers. A bus with a smaller form factor but similar functions and electrical specifications was designed for European use, and promoted by Motorola, Phillips, Thompson, and other companies. This was the VersaModule European, or VME, bus. Beginning with rev B of 1982, the bus quickly became an accepted standard.

VME Features

A VME bus is a set of parallel conductors that interconnect multiple processing devices. The devices can exchange data in units of 8, 16, 32 or 64 bits during a bus cycle.

VME Address Spaces

Each VME device identifies itself with a range of bus addresses. A bus address has either 16, 24, or 32 bits of precision. Each address width forms a separate address space. That is, the same numeric value can refer to one device in the 24-bit address space but a different device in the 32-bit address space. Typically, a device operates in only one address space, but some devices can be configured to respond to addresses in multiple spaces.

Each VME bus cycle contains the bits of an address. The address is qualified by sets of address-modifier bits that specify the following:

- the address space (A16, A24, or A32)
- whether the operation is single or a block transfer
- whether the access is to what, in the MC68000 architecture, would be data or code, in a supervisor or user area (Silicon Graphics systems support only supervisor-data and user-data requests)

Master and Slave Devices

Each VME device acts as either a bus master or a bus slave. Typically a bus master is a device with some level of programmability, usually a microprocessor. A disk controller is an example of a master device. A slave device is typically a nonprogrammable device like a memory board.

Each data transfer is initiated by a master device. The master

- asserts ownership of the bus
- specifies the address modifier bits for the transfer, including the address space, single/block mode, and supervisor/normal mode
- specifies the address for the transfer
- specifies the data unit size for the transfer (8, 16, 32 or 64 bits)
- specifies the direction of the transfer with respect to the master

The VME bus design permits multiple master devices to use the bus, and provides a hardware-based arbitration system so that they can use the bus in alternation.

A slave device responds to a master when the master specifies the slave’s address. The addressed slave accepts data, or provides data, as directed.

VME Transactions

The VME design allows for four types of data transfer bus cycles:

- A read cycle returns data from the slave to the master.
- A write cycle sends data from the master to the slave.
- A read-modify-write cycle takes data from the slave, and on the following bus cycle sends it back to the same address, possibly altered.
- A block-transfer transaction sends multiple data units to adjacent addresses in a burst of consecutive bus cycles.

The VME design also allows for interrupts. A device can raise an interrupt on any of seven *interrupt level*s. The interrupt is acknowledged by a bus master. The bus master interrogates the interrupting device in an interrupt-acknowledge bus cycle, and the device returns an interrupt vector number.

In Silicon Graphics systems, it is always the Silicon Graphics VME controller that acknowledges interrupts. It passes the interrupt to one of the CPUs in the system.

VME Bus in Silicon Graphics Systems

The VME bus was designed as the system backplane for a workstation, supporting one or more CPU modules along with the memory and I/O modules they used. However, no Silicon Graphics computer uses the VME bus as the system backplane. In all Silicon Graphics computers, the main system bus that connects CPUs to memory is a proprietary bus design, with higher speed and sometimes wider data units than the VME bus provides. The VME bus is attached to the system as an I/O device.

This section provides an overview of the design of the VME bus in any Silicon Graphics system. It is sufficient background for most users of VME devices. For a more detailed look at the Challenge or Onyx implementation of VME, see “VME Hardware in Challenge and Onyx Systems” on page 313.

The VME Bus Controller

A VME bus controller is attached to the system bus to act as a bridge between the system bus and the VME bus. This arrangement is shown in Figure 13-1.

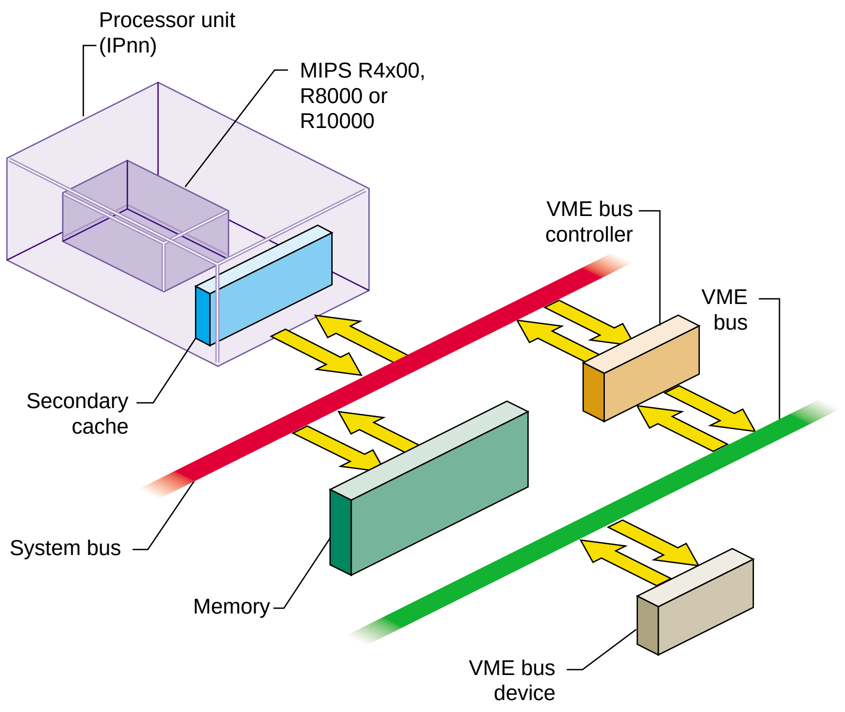

Figure 13-1 Relationship of VME Bus to System Bus

On the Silicon Graphics system bus, the VME bus controller acts as an I/O device. On the VME bus, the bus controller acts as a VME bus master.

The VME controller has several tasks. Its most important task is mapping; that is, translating some range of physical addresses in the Silicon Graphics system address space to a range of VME bus addresses. The VME controller performs a variety of other duties for different kinds of VME access.

VME PIO Operations

During programmed I/O (PIO) to the VME bus, software in the CPU loads or stores the contents of CPU registers to a device on the VME bus. The operation of a CPU load from a VME device register is as follows:

1. The CPU executes a load from a system physical address.
2. The system recognizes the physical address as one of its own.
3. The system translates the physical address into a VME bus address.
4. Acting as a VME bus master, the system starts a read cycle on the VME bus.
5. A slave device on the VME bus responds to the VME address and returns data.
6. The VME controller initiates a system bus cycle to return the data packet to the CPU, thus completing the load operation.

A store to a VME device is similar except that it performs a VME bus write, and no data is returned.

PIO input requires two system bus cycles—one to request the data and one to return it—separated by the cycle time of the VME bus. PIO output takes only one system bus cycle, and the VME bus write cycle run concurrently with the next system bus cycle. As a result, PIO input always takes at least twice as much time as PIO output. (For details, see "VME PIO Bandwidth" on page 66.)

VME DMA Operations

A VME device that can act as a bus master can perform DMA into memory. The general sequence of operations in this case is as follows:

1. Software in the Silicon Graphics CPU uses PIO to program the device registers of the VME device, instructing it to perform DMA to a certain VME bus address for a specified length of data.
2. The VME bus master initiates the first read, write, block-read, or block-write cycle on the VME bus.
3. The VME controller, responding as a slave device on the VME bus, recognizes the VME bus address as one that corresponds to a physical memory address in the system.

4. If the bus master is writing, the VME controller accepts the data and initiates a system bus cycle to write the data to system memory.

   If the bus master is reading, the VME controller uses a system bus cycle to read data from system memory, and returns the data to the bus master.

5. The bus master device continues to use the VME controller as a slave device until it has completed the DMA transfer.

During a DMA transaction, the VME bus controller operates independently of any CPU. CPUs in the system execute software concurrently with the data transfer. Since the system bus is faster than the VME bus, the data transfer takes place at the maximum data rate that the VME bus master can sustain.

Operation of the DMA Engine

In Silicon Graphics Crimson systems, the VME controller supports DMA by VME bus master devices. However, in the Challenge and Onyx lines, the VME controller contains an additional "DMA Engine" that can be programmed to perform DMA-type transfers between memory and a VME device that is a slave, not a bus master. The general course of operations in a DMA engine transfer is as follows:

1. The VME bus controller is programmed to perform a DMA transfer to a certain physical address for a specified amount of data from a specified device address in VME address space.
2. The VME bus controller, acting as the VME bus master, initiates a block read or block write to the specified device.
3. As the slave device responds to successive VME bus cycles, the VME bus controller transfers data to or from memory using the system bus.

The DMA engine transfers data independently of any CPU, and at the maximum rate the VME bus slave can sustain. In addition, the VME controller collects smaller data units into blocks of the full system bus width, minimizing the number of system bus cycles needed to transfer data. For both these reasons, DMA engine transfers are faster than PIO transfers for all but very short transfer lengths. (For details, see "DMA Engine Bandwidth" on page 73.)

VME Bus Addresses and System Addresses

Devices on the VME bus exist in one of the following address spaces:

- The 16-bit space (A16) contains numbers from 0x0000 to 0xffff.
- The 24-bit space (A24) contains numbers from 0x00 0000 to 0xff ffff.
- The 32-bit space (A32) uses numbers from 0x0000 0000 to 0xffff ffff.
- The 64-bit space (A64), defined in the revision D specification, uses 64-bit addresses.

The Silicon Graphics system bus also uses 32-bit or 64-bit numbers to address memory and other I/O devices on the system bus. In order to avoid conflicts between the meanings of address numbers, certain portions of the physical address space are reserved for VME use. The VME address spaces are mapped, that is, translated, into these ranges of physical addresses.

The translation is performed by the VME bus controller: It recognizes certain physical addresses on the system bus and translates them into VME bus addresses; and it recognizes certain VME bus addresses and translates them into physical addresses on the system bus.

Even with mapping, the entire A32 or A64 address space cannot be mapped into the physical address space. As a result, no Silicon Graphics system provides access to all of the VME address spaces. Only parts of the VME address spaces are available at any time. The mapping methods of the Silicon Graphics Crimson series differ from the methods of the later Challenge and Onyx series of machines.

User-Level and Kernel-Level Addressing

In a user-level program you can perform PIO and certain types of DMA operations (see Chapter 4, "User-Level Access to VME and EISA"), but you do not program these in terms of the physical addresses mapped to the VME bus. Instead, you call on the services of a kernel-level device driver to map a portion of VME address space into the address space of your process. The requested segment of VME space is mapped dynamically to a segment of your user-level address space—a segment that can differ from one run of the program to the next.

In a kernel-level device driver, you program PIO and DMA operations in terms of specific addresses in kernel space—memory addresses that are mapped to specified addresses in the VME bus address space. The mapping is either permanent, established by the system hardware, or dynamic, set up temporarily by a kernel function.

Note: The remainder of this chapter has direct meaning only for kernel-level drivers, which must deal with physical mappings of VME space.

PIO Addressing and DMA Addressing

The addressing needs of PIO access and DMA access are different.

PIO deals in small amounts of data, typically single bytes or words. PIO is directed to device registers that are identified with specific VME bus addresses. The association between a device register and its bus address is fixed, typically by setting jumpers or switches on the VME card.

DMA deals with extended segments of kilobytes or megabytes. The addresses used in DMA are not fixed in the device, but are programmed into it just before the data transfer begins. For example, a disk controller device can be programmed to read a certain sector and to write the sector data to a range of 512 consecutive bytes in the VME bus address space. The programming of the disk controller is done by storing numbers into its registers using PIO. While the registers respond only to fixed addresses that are configured into the board, the address to which the disk controller writes its sector data is just a number that is programmed into it each time a transfer is to start.

The key differences between addresses used by PIO and addresses used for DMA are these:

- PIO addresses are relatively few in number and cover small spans of data, while DMA addresses can span large ranges of data.
- PIO addresses are closely related to the hardware architecture of the device and are configured by hardware or firmware, while DMA addresses are simply parameters programmed into the device before each operation.

In Crimson systems, portions of the physical address space are permanently mapped to portions of the VME address space. PIO is performed to these designated, fixed physical addresses. Some DMA addresses are also assigned fixed mappings, but additional segments are assigned dynamically as needed.

In Challenge and Onyx systems, all VME mappings are dynamic, assigned as needed. Kernel functions are provided to create and release mappings between designated VME addresses and kernel addresses.

It is important to realize that the kernel functions provided for dynamic mapping in the Challenge or Onyx systems also work in the Crimson line. A device driver that uses these functions is portable to all systems supported by IRIX 6.2.

PIO Addressing in Challenge and Onyx Systems

The Challenge and Onyx systems and their Power versions support from one to five VME buses. It is impossible to fit adequate segments of five separate A16, A24, and A32 address spaces into fixed mappings in the 40-bit physical address space available in these systems.

The VME controller in Challenge and Onyx systems uses programmable mappings. The IRIX kernel can program the mapping of twelve separate 8 MB “windows” on VME address space on each bus (a total of 96 MB of mapped space per bus). The kernel sets up VME mappings by setting the base addresses of these windows as required. A kernel-level VME device driver asks for and uses PIO mappings through the functions documented in “Mapping PIO Addresses” on page 325.

Note: The same kernel functions can be called to set up mappings for PIO addresses in the Crimson series. It is not necessary to write code that depends on the fixed addresses documented in the following section.

PIO Addressing in the Crimson Series

Systems in the Silicon Graphics Crimson series can have one or two independent VME buses. The first bus is bus 0 and the second is bus 1. For PIO purposes, all or part of the VME A16, A24, and A32 address spaces from each bus are assigned to fixed mappings in system physical address space.

Note: When you read a device driver written specifically for the Crimson series you may see use of the addresses documented in this section. When writing a new driver, you are urged to avoid the use of these fixed addresses, and instead use the functions for dynamic assignment of PIO maps discussed under “PIO Addressing in Challenge and Onyx Systems” on page 306. The end result will be the same, but the driver will be portable to all systems.

Crimson Mapping of A16 Space

The A16 address space for each bus is mapped in full as shown in Table 13-1.

Table 13-1 VME A16 Addressing in Crimson Systems

VME A16 Address Range	Address Modifier	Kernel Address VME Bus 0	Kernel Address VME Bus 1
0x0000-0xFFFF	0x29 (normal data)	0xB7C0 0000–0xB7C0 FFFF	0xB780 0000–0xB780 FFFF
0x0000-0xFFFF	0x2D (supervisor data)	0xB7C1 0000–0xB7C1 FFFF	0xB781 0000–0xB781 FFFF

Crimson Mapping of A24 Space

The upper 8 MB of the A24 address space for supervisory and program data access is mapped as shown in Table 13-2.

Table 13-2 VME A24 Addressing in Crimson Systems

VME A24 Address Range	Address Modifier	Kernel Address VME Bus 0	Kernel Address VME Bus 1
0x80 0000–0xFF FFFF	0x39 (normal data)	0xB280 0000–0xB2FF FFFF	0xF080 0000–0xF0FF FFFF
0x80 0000–0xFF FFFF	0x3D (supervisor data)	0xB380 0000–0xB3FF FFFF	0xF180 0000–0xF1FF FFFF

Crimson Mapping of A32 Space

The Crimson architecture allows up to 256 MB of physical address space for mapping a segment of VME A32 addresses. When the system contains only one VME bus, a single 256 MB segment of the A32 space is mapped as shown in Table 13-3.

Table 13-3 VME A32 Addressing in Crimson Systems (One Bus)

VME A32 Address Range	Address Modifier	Kernel Address
0x1000 0000–0x1FFF FFFF	0x09 (extended normal data)	0xD000 0000–0xDFFF FFFF
0x1000 0000–0x1FFF FFFF	0x0D (extended supervisor data)	0xE000 0000–0xEFFF FFFF

When the system contains two buses, the available space is divided so that a segment of 128 MB from each bus is mapped, as shown in Table 13-4.

Table 13-4 VME A32 Addressing in Crimson Systems (Two Buses)

VME A32 Address Range	Address Modifier	Kernel Address VME Bus 0	Kernel Address VME Bus 1
0x1800 0000–0x1FFF FFFF	0x09 (extended normal data)	0xD800 0000–0xDFFF FFFF	0xD000 0000–0xD7FF FFFF
0x1800 0000–0x1FFF FFFF	0x0D (extended supervisor data)	0xE800 0000–0xEFFF FFFF	0xE000 0000–0xE800 0000

DMA Addressing

DMA is supported only for the A24 and A32 address spaces. DMA addresses are always assigned dynamically in the Challenge and Onyx systems.

The Crimson series supports two modes of DMA mapping: direct and mapped. In direct mode, the VME bus address is the same as the target physical address in memory. In mapped mode, a segment of VME bus addresses is mapped dynamically to a segment of kernel virtual addresses.

DMA Addressing in the Challenge and Onyx Systems

In order to establish a mapping for DMA use, a kernel-level driver should call the functions documented in “Mapping DMA Addresses” on page 330. These functions work in all systems supported by IRIX 6.2.

Direct DMA Addressing in the Crimson Series

When you examine a device driver written for the Crimson series you may find direct DMA addressing used. Under direct addressing, the target physical address of the buffer in memory is programmed into the VME bus master device as its target address in the VME address space. For example, a device driver could get the physical address of a page buffer in memory, and program that into a VME disk controller as the target address for a sector read or write.

Direct mapping is not recommended. It is hardware-dependent and nonportable. Although direct mapping is simple, it has the disadvantage that the target buffer must be in contiguous physical memory locations. When a buffer in kernel virtual memory spans two or more pages, the parts of the buffer are likely to be in noncontiguous page frames. In order to handle noncontiguous pages using direct mapping, the device driver must program a separate transfer for each page.

Mapped DMA Addressing in the Crimson Series

The Crimson series also supports DMA mapping. Using DMA mapping, the device driver calls a kernel function passing it a kernel virtual address and a length (which can exceed a page in size). The kernel function programs the VME bus controller with mapping information.

When a mapping is established, the device driver can use a kernel function to get the VME bus address that has been mapped to the first byte of the buffer. This address is programmed into the VME bus master as its target address.

A mapped target address in the Crimson always has a most significant bit of 1. When the VME bus controller, acting as a slave device, is told to read or write to an address with a most significant bit of 1, it recognizes a mapped address, and uses the mapping tables set up by the kernel to translate the VME bus address into a physical memory address.

Mapped DMA addressing has two important advantages. It takes care of the problem of discontiguous page frames—the VME bus master uses a contiguous range of VME addresses, and never knows that the VME bus controller is scattering or gathering data from different pages. And it is portable to all systems—the kernel functions work identically in Crimson and in Challenge and Onyx systems.

Configuring VME Devices

To install a VME device in a Silicon Graphics system, you need to configure the device itself to respond to PIO addresses in a supported range of VME bus addresses, and you need to inform IRIX of the device addresses.

Configuring Device Addresses

Normally a VME card can be programmed to use different VME addresses for PIO, based on jumper or switch settings on the card. The devices on a single VME bus must be configured to use unique addresses. Errors that are hard to diagnose can arise when multiple cards respond to the same bus address. Devices on different VME buses can use the same addresses.

Not all parts of each address space are accessible. The accessible parts are summarized in Table 13-5.

Table 13-5 Accessible VME Addresses in Each System

Address Space	Crimson Series	Challenge and Onyx Systems
A16	All	All
A24	0x80 0000–0xFF FFFF	0x80 0000–0xFF FFFF
A32	0x1000 0000–0x1FFF FFFF (one bus) 0x1800 0000–0x1FFF FFFF (2 buses)	0x0000 0000–0x7FFF FFFF (maximum of 96 MB in 8 MB units)

Within the accessible ranges, certain VME bus addresses are used by Silicon Graphics VME devices. You can find these addresses documented in the */var/sysgen/system/irix.sm* file. You must configure OEM devices to avoid the addresses used by Silicon Graphics devices that are installed on the same system.

Finally, on the Challenge and Onyx systems, take care to cluster PIO addresses in the A32 space so that they occupy at most a 96 MB span of addresses. The reasons are explained under “Fixed PIO Maps” on page 328.

Configuring the System Files

Inform IRIX and the device driver of the existence of a VME device by adding a VECTOR statement to a file in the directory */var/sysgen/system* (see “System Configuration Files” on page 39). The syntax of a VECTOR statement is documented in two places:

- The */var/sysgen/system/irix.sm* file itself contains descriptive comments on the syntax and meaning of the statement, as well as numerous examples.
- The system(4) reference page gives a more formal definition of the syntax.

In addition to the VECTOR statement, you may need to code an IPL statement.

Coding the VECTOR Statement

The important elements in a VECTOR line are as follows:

bustype	Specified as *VME* for VME devices. The VECTOR statement can be used for other types of buses as well.
module	The base name of the device driver for this device, as used in the */var/sysgen/master.d* database (see “Master Configuration Database” on page 38 and “How Names Are Used in Configuration” on page 232).
adapter	The number of the VME bus where the device is attached—0 or 1 in a Crimson system; the bus number in a Challenge or Onyx machine.
ipl	The interrupt level at which the device causes interrupts, from 0 to 7.
vector	An 8-bit value between 1 and 254 that the device returns during an interrupt acknowledge cycle.
ctlr	The “controller” number is simply an integer parameter that is passed to the device driver at boot time. It can be used for example to specify a logical unit number.
iospace, *iospace2*, *iospace3*	Each *iospace* group specifies the VME address space, the starting bus address, and the size of a segment of VME address space used by this device.
probe or *exprobe*	Specify a hardware test that can be applied at boot time to find out if the device exists.

Use the *probe* or *exprobe* parameter to program a test for the existence of the device at boot time. If the device does not respond (because it is offline or because it has been removed from the system), the *lboot* command will not invoke the device driver for this device.

The device driver specified by the *module* parameter is invoked at its *pfx***edtinit()** entry point, where it receives most of the other information specified in the VECTOR statement (see "Entry Point edtinit()" on page 144).

Omit the *vector* parameter in either of two cases: when the device does not cause interrupts, or when it supports a programmable interrupt vector (see "Allocating an Interrupt Vector Dynamically" on page 332).

Use the *iospace* parameters to pass in the exact VME bus addresses that correspond to this device, as configured in the device. Up to three address space ranges can be passed to the driver. This does not restrict the device—it can use other ranges of addresses, but the device driver has to deduce their addresses from other information. The device driver typically uses this data to set up PIO maps (see "Mapping PIO Addresses" on page 325).

Using the IPL Statement

In a Challenge or Onyx system only, you can direct VME interrupts to specific CPUs. This is done with the IPL statement, also written into a file in */var/sysgen/system*. The IPL statement, which like the VECTOR statement is documented in both the system(4) reference page and the */var/sysgen/system/irix.sm* file itself, has only two parameters:

level	The VME interrupt level to be directed, 0 to 7 (the same value that is coded as *ipl*= in the VECTOR statement.
cpu	The number of the CPU that should handle all VME interrupts at this level.

The purpose of the IPL statement is to send interrupts from specific devices to a specific CPU. There are two contradictory reasons for doing this:

- That CPU is dedicated to handling those interrupts with minimum latency
- Those interrupts would disrupt high-priority work being done on other CPUs if they were allowed to reach the other CPUs.

The IPL statement cannot direct interrupts from a specific device; it directs all interrupts that occur at the specified level.

VME Hardware in Challenge and Onyx Systems

The overview topic, "VME Bus in Silicon Graphics Systems" on page 300, provides sufficient orientation for most users of VME devices. However, if you are designing hardware or a high-performance device driver specifically for the Challenge and Onyx systems, the details in this topic are important.

Note: For information on physical cabinets, panels, slot numbering, cables and jumpers, and data about dimensions and airflow, refer to the Owner's Guide manual for your machine. For example, see the *POWER CHALLENGE AND CHALLENGE XL Rackmount Owner's Guide* (007-1735) for the physical layout and cabling of VME busses in the large Challenge systems.

VME Hardware Architecture

The VME bus interface circuitry for Challenge and Onyx systems resides on a mezzanine board called the VMEbus Channel Adapter Module (VCAM) board. One VCAM board is standard in every system and mounts directly on top of the IO4 board in the system card cage.

The IO4 board is the heart of the I/O subsystem. The IO4 board supplies the system with a basic set of I/O controllers and system boot and configuration devices such as serial and parallel ports, and Ethernet.

In addition, the IO4 board provides these interfaces:

- two Flat Cable Interconnects (FCIs) for connection to Card Cage 3 (CC3)
- two SCSI-2 cable connections
- two Ibus connections

A Challenge or Onyx system can contain multiple IO4 boards, which can operate in parallel. (Early versions of the IO4 have a possible hardware problem that is described in Appendix B, "Challenge DMA with Multiple IO4 Boards.")

Main System Bus

The main set of buses in the Challenge and Onyx system architecture is the Everest address and data buses, Ebus for short. The Ebus provides a 256-bit data bus and a 40-bit address bus that can sustain a bandwidth of 1.2 GB per second.

The 256-bit data bus provides the data transfer capability to support a large number of high-performance RISC CPUs. The 40-bit address bus is also wide enough to support 16 GB of contiguous memory in addition to an 8 GB I/O address space.

Ibus

The 64-bit Ibus (also known as the HIO bus) is the main internal bus of the I/O subsystem and interfaces to the high-power Ebus through a group of bus adapters.The Ibus has a bandwidth of 320 MB per second that can sufficiently support a graphics subsystem, a VME64 bus, and as many as eight SCSI channels operating simultaneously.

Bus Interfacing

Communication with the VME and SCSI buses, the installed set or sets of graphics boards, and Ethernet takes place through the 64-bit Ibus. The Ibus interfaces to the main system bus, the 256-bit Ebus, through a set of interface control devices, an I address (IA) and four I data (ID). The ID ASICs latch the data, and the IA ASIC clocks the data from each ID to the Flat Cable Interface (FCI) through the F controller (or F chip).

Two FCI controllers (or F controllers) help handle the data transfers to and from an internal graphics board set (if present) and any VMEbus boards in optional CC3 applications. The SCSI-2 (S1) controller serves as an interface to the various SCSI-2 buses. The Everest peripheral controller (EPC) device manages the data movement to and from the Ethernet, a parallel port, and various types of on-board PROMs and RAM.

Maximum Latency

The worst-case delay for the start of a VME access, if all of the devices on the IO4 simulataneously request the IO channel for a 128 byte write and the VME adapter receives the grant last, the VME access start could be delayed for a total of about 2 microseconds. Only a VME read would suffer this delay; a VME write would not.

There is a another potential delay from an independent cause, which depends on the number of bus master (IO4 and CPU) boards on the system bus. If all the E-bus masters in a fairly large configuration hit the bus at once, a VME read or write could be held up by as much as 1 microsecond in a large system.

VME Bus Numbering

The Challenge and Onyx systems support up to five independent VME buses in a single system. The numbering of these buses is not sequential.

There is always a VME bus number 0. This is the bus connected to the system midplane. It is always connected by the primary IO4 board (the IO4 board attached to the highest slot on the system bus).

Bus numbers for other VME buses depend on the Ebus slot number where their IO4 is attached, and on the I/O adapter number of the VCAM card on the IO4. Each IO4 board supports adapter numbers from 1 to 7, but a VME bus can only be attached to adapter number 2, 3, 5, or 6. These four adapters are given VME index numbers of 0, 1, 2, and 3 respectively.

The bus number of a VME bus is given by E*4+A, where E is the Ebus slot of the IO4 card, and A is the index of the adapter on the IO4. A VME bus attached through adapter 3 on the IO4 in Ebus slot 13 is bus number 53, from the sum of adapter index 1 and 4 times the slot number.

VMEbus Channel Adapter Module (VCAM) Board

The VCAM board provides the interface between the Ebus and the VMEbus and manages the signal level conversion between the two buses. The VCAM also provides a pass-through connection that ties the graphics subsystem to the Ebus.

The VCAM can operate as either a master or a slave. It supports DMA-to-memory transactions on the Ebus and programmed I/O (PIO) operations from the system bus to addresses on the VMEbus. In addition, the VCAM provides virtual address translation capability and a DMA engine that increases the performance of non-DMA VME boards.

VMECC

The VMECC (VME Cache Controller) gate array is the major active device on the VCAM. The VMECC interfaces and translates host CPU operations to VMEbus operations (see Figure 13-2). The VMECC also decodes VMEbus operations to translate them to the host side.

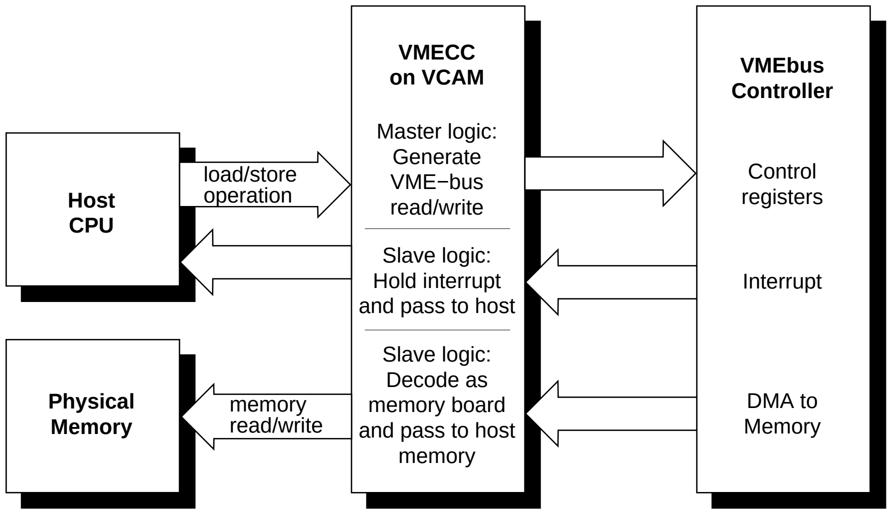
VMECC
on VCAM
Master logic: Generate VME−bus read/write
Slave logic: Hold interrupt and pass to host
Slave logic: Decode as memory board and pass to host memory
VMEbus Controller
Control registers
Interrupt
DMA to Memory
Host CPU
load/store operation
Physical Memory
memory read/write

Figure 13-2 VMECC, the VMEbus Adapter

The VMECC provides the following features:

- an internal DMA engine to speed copies between physical memory and VME space (see “Operation of the DMA Engine” on page 303)
- a 16-entry deep PIO FIFO to smooth writing to the VME bus from the host CPUs
- a built-in VME interrupt handler and built-in VME bus arbiter
- an explicit internal delay register to aid in spacing PIOs for VME controller boards that cannot accept back-to-back operations
- support for issuing A16, A24, A32, and A64 addressing modes as a bus master during PIO
- support for single-item transfers (D8, D16, D32, and D64) as a bus master during PIO
- support for response as a slave to A24, A32, and A64 addressing modes to provide DMA access to the Ebus
- support for single-item transfers (D8, D16, and D32) as a slave during DMA access to the Ebus
- support for block item transfers (D8, D16, D32, and D64) as a slave during DMA access to the Ebus

The VMECC also provides four levels of VMEbus request grants, 0-3 (3 has the highest priority), for DMA arbitration. Do not confuse these bus request levels with the interrupt priority levels 1-7. Bus requests prioritize the use of the physical lines representing the bus and are normally set by means of jumpers on the interface board.

F Controller ASIC

Data transfers between VME controller boards and the host CPU(s) takes place through the VMECC on the VCAM board, then through a flat cable interface (FCI), and onto the F controller ASIC.

The F controller acts as an interface between the Ibus and the Flat Cable Interfaces (FCIs). This device is primarily composed of FIFO registers and synchronizers that provide protocol conversion and buffer transactions in both directions and translate 34-bit I/O addresses into 40-bit system addresses.

Two configurations of the F controller are used on the IO4 board; the difference between them is the instruction set they contain. One version is programmed with a set of instructions designed to communicate with the GFXCC (for graphics); the other version has instructions designed for the VMECC. All communication with the GFXCC or VMECC ICs is done over the FCI, where the F controller is always the slave.

Both versions of the F controller ASICs have I/O error-detection and handling capabilities. Data errors that occur on either the Ibus or the FCI are recorded by the F controller and sent to the VMECC or GFXCC.

ICs must report the error to the appropriate CPU and log any specific information about the operation in progress. FCI errors are recorded in the error status register. This register provides the status of the first error that occurred, as well as the cause of the most recent FCI reset.

VMEbus Interrupt Generation

The VME bus supports seven levels of prioritized interrupts, 1 through 7 (where 7 has the highest priority). The VMECC has a register associated with each level. When the system responds to the VMEbus interrupt, it services all devices identified in the interrupt vector register in order of their VMEbus priority (highest number first). The following list outlines how a VMEbus interrupt is generated:

1. A VME controller board asserts a VME interrupt on one of the IRQ levels.
2. The built-in interrupt handler in the VMECC chip checks if the interrupt level is presently enabled by an internal interrupt mask.
3. The interrupt handler in the VMECC issues a bussed IACK (interrupt acknowledge) response and acquires the vector from the device. The 3-bit response identifies one of the seven VME levels.
4. If multiple VME boards are present, the bussed IACK signal is sent to the first VME controller as an IACKIN. When the first controller is not the requesting master, it passes the IACKIN signal to the next board (in the daisy-chain) as IACKOUT.
5. The requesting board responds to IACKIN by issuing a DTACK* (data acknowledge signal), blocking the IACKOUT signal to the next board, and placing an 8-bit interrupt vector number on the data bus.
6. The VMECC latches the interrupt vector, and an interrupt signal is sent over the FCI interface to the F-chip and is queued awaiting completion of other F-chip tasks.
7. The F controller ASIC requests the I-bus and sends the interrupt to the IA chip.
8. The IA chip requests the Ebus and sends the interrupt over the Ebus to the CC chip on the IP19/IP21 board.
9. The CC chip interrupts the R4400/R8000, provided that the interrupt level is not masked.

The time for this to complete is normally less than 3 microseconds, but will be queued to await completion of other VME activities.

VME Interface Features and Restrictions

The Challenge and Onyx VME interface supports all protocols defined in Revision C of the VME specification plus the A64 and D64 modes defined in Revision D. The D64 mode allows DMA bandwidths of up to 60 MB. This bus also supports the following features:

- seven levels of prioritized processor interrupts
- 16-bit, 24-bit, and 32-bit data addresses and 64-bit memory addresses
- 16-bit and 32-bit accesses (and 64-bit accesses in MIPS III mode)
- 8-bit, 16-bit, 32-bit, and 64-bit data transfer
- DMA to and from main memory

DMA Multiple Address Mapping

In the Challenge and Onyx series, a DMA address from a VME controller goes through a two-level translation to generate an acceptable physical address. This requires two levels of mapping. The first level of mapping is done through the map RAM on the IO4 board. The second level is done through the map tables in system memory. This mapping is shown in Figure 13-3.

Note: The second level mapping requires system memory to be reserved for the mapping tables. The current limit on the number of pages that is allocated for map tables is 16 pages and the maximum memory allotted for the map tables is 64 KB. The R4400 provides a 4 KB page size for 16 pages (4 KB * 16 pages= 64 KB). The R8000 provides a 16 KB page size for 4pages (16 KB * 4 pages = 64 KB).

Each second-leel map table entry corresponds to 4 KB of physical memory. In addition, each second-level map table entry is 4 bytes. With 64 KB of mapping table, there are a total of 16 K entries translating a maximum of 64 MB of DMA mapping, setting a limit of 64 MB that can be mapped at any time for each VME bus. This does not set any limit on the amount of DMA that can be done by a board during its operation.

Referring to the top of Figure 13-3, bits 32 and 33 from the IBus address come from the VMECC. These two bits determine a unique VMEbus number for systems with multiple VME buses. Of the remaining 32 bits (31 to 0), 12 are reserved for an offset in physical memory, and the other 20 bits are used to select up to 2^{20} or 1 million pages into the main memory table. However, as stated earlier only 64 KB is allocated for map tables.

As shown in Figure 13-3, thirteen bits go to the static RAM table. Recall that two of the thirteen bits are from the VMECC to identify the VMEbus number. The static RAM table then generates a 29-bit identifier into the main memory table. These 29 bits select a table in the main memory table. An additional nine bits select an entry or element within the table. A 00 (two zeros) are appended to form a 40-bit address into the main memory table.

The main memory table then generates 28-bit address which is then appended to the 12-bit offset of the IBus to form the final 40-bit physical address.

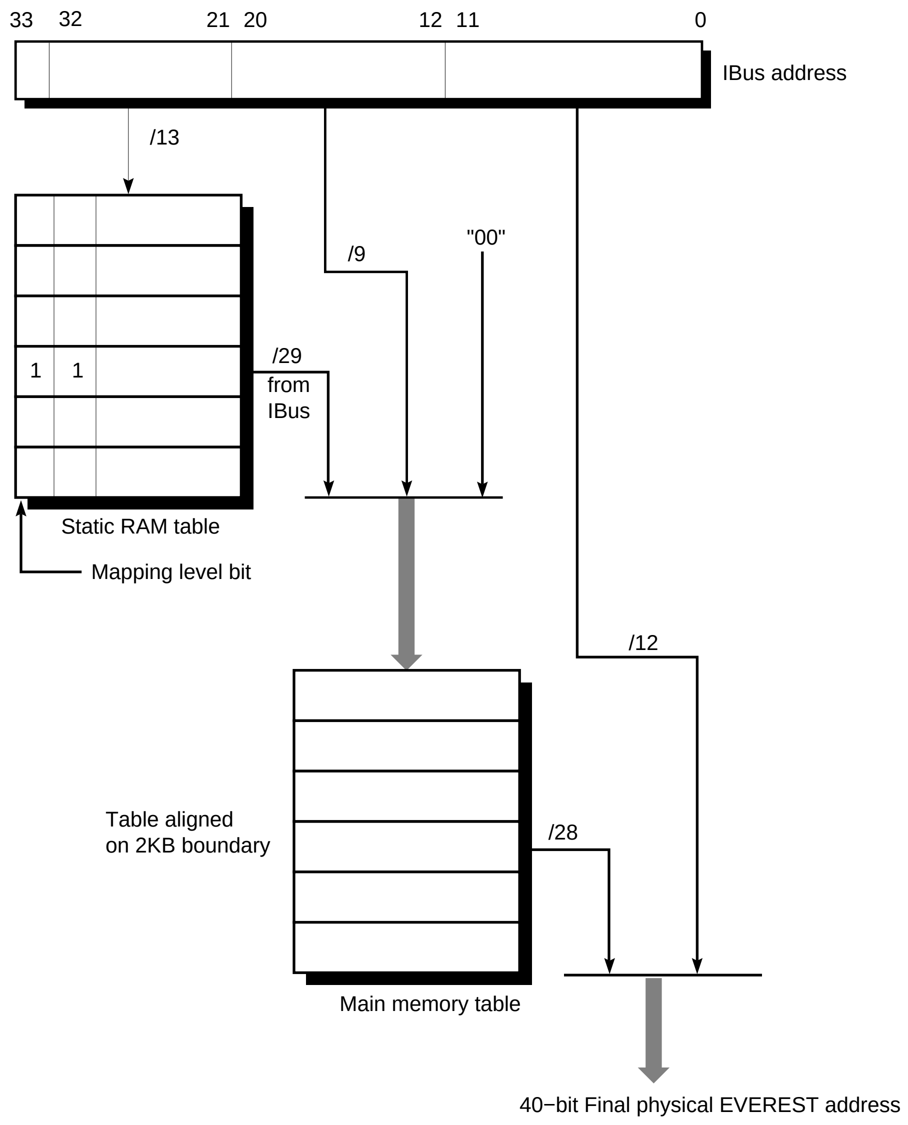

Figure 13-3 I/O Address to System Address Mapping

VME Interrupt Priority

Interrupts within the Challenge/Onyx architecture are managed by a set of interrupt vectors. An interrupt generated by an I/O device like a VME controller in turn generates an interrupt to the CPU on one of the previously assigned levels.

Each IRQ on each VME bus is assigned an internal interrupt level, and by default all these levels are at the same priority. If multiple interrupts arrive simultaneously within a single bus, for example IRQ 3 and IRQ 4 at once, priority is given to the higher-numbered IRQ.

All VME interrupts are made to go to CPU 0 (unless configured differently with IPL statements). This prevents one interrupt level from preempting the driver handling a different VME interrupt level.

VME Hardware Features and Restrictions

When designing an OEM hardware board to interface to the Challenge or Onyx VME bus, observe the following restrictions:

- Devices should require 8-bit interrupt vectors only. This is the only interrupt vector size that is supported by the VMECC or recognized by the IRIX kernel.
- Devices must not require UAT (unaligned transfer or tri-byte) access.
- Devices in slave mode must not require address modifiers other than Supervisory/Nonprivileged data access.
- While in master mode, a device must use only nonprivileged data access or nonprivileged block transfers.
- The Challenge or Onyx VME bus does not support VSBbus boards. In addition, there are no pins on the back of the VME backplane. This area is inaccessible for cables or boards.
- Metal face plates or front panels on VME boards may prevent the I/O door from properly closing and can possibly damage I/O bulkhead. (In some VME enclosures, a face plate supplies required EMI shielding. However, the Challenge chassis already provides sufficient shielding, so these plates are not necessary.)

Designing a VME Bus Master for Challenge and Onyx Systems

The following notes are related to the design of a VME bus master device to work with a machine in the Challenge/Onyx series. A VME bus master, when programmed using the functions described in "Mapping DMA Addresses" on page 330, can transfer data between itself and system memory.

The setup time at the start of a DMA transfer is as follows:

- First word of a read is delivered to the master in 3 to 8 microseconds.
- First word of a write is retrieved from the master in 1 to 3 microseconds.

The F controller does the mapping from A32 mode into system memory and automatically handles the crossing of page boundaries. The VME Bus Master is not required to make AS go high and then low on 4 KB boundaries. However, when using A64 addressing, the device may have to change the address on the 4 KB boundaries and cause a transition on AS low to high, and then back to low. This incurs new setup delays.

The important parts of the VME handshake cycle are diagrammed in Figure 13-4.

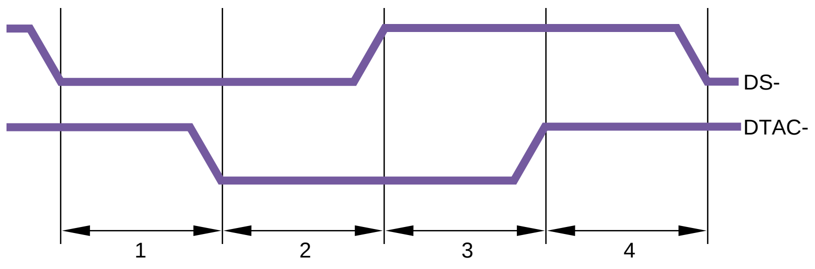

Figure 13-4 VMECC Contribution to VME Handshake Cycle Time

Intervals 1 and 3 represent the response latency in the bus slave (the VMECC). Intervals 2 and 4 represent the latency in the VME Bus Master. In the Challenge and Onyx systems,

- part 1 is approximately 40 nanoseconds
- part 3 is approximately 25 nanoseconds

The total contribution by the VMECC is approximately 65 nanoseconds. If the total of the four intervals can be held to 125 nanoseconds, the absolute peak transfer rate (in D64 mode) is 64 MB per second.

Note: Startup and latency numbers are averages and may occasionally be longer. The system design does not have any guaranteed latency.

The VME specification provides for a burst length of 265 bytes in D8, D16, and D32 modes, or 2 KB in D64. The burst length is counted in bytes, not transfer cycles.

Operating at this burst length, the duration of a single burst of 2 KB would be 256 transfers at 125 nanoseconds each, plus a startup of roughly 5 microseconds, giving a total of 37 microseconds per burst. Continuous, back-to-back bursts could achieve at most 55 MB per second.

However, the Challenge and Onyx VMECC uses a 20-bit burst counter allowing up to 2 MB in a burst of any data size. Suppose the bus master transfers 64 KB per burst, transferring 4-byte data words. The duration of a single burst would be 8,192 times 125 nanoseconds, plus 5 microseconds startup, or 1,029 microseconds per burst. Continuous bursts of this size achieve a data rate of 63.7 MB per second.

The use of long bursts violates the VME standard, and a bus master that depends on long bursts is likely not to work in other computers. If you decide to exceed the VME bus specifications, you should condition this feature with a field in a control register on the VME board, so that it can be disabled for use on other VME systems.

Chapter 14

Services for VME Drivers

This chapter provides an overview of the kernel services needed by a kernel-level VME device driver. It contains a complete example driver.

Kernel Services for VME

The kernel provides services for mapping the VME bus into the kernel virtual address space for PIO or DMA, and for transferring data using maps. It also provides services for allocating interrupt vector numbers.

Mapping PIO Addresses

A PIO map is a system object that represents the mapping from a location in the kernel's virtual address space to some small range of addresses on a VME or EISA bus. After creating a PIO map, a device driver can use it in the following ways:

- Use the specific kernel virtual address that represents the device, either to load or store data, or to map that address into user process space.
- Copy data between the device and memory without learning the specific kernel addresses involved.
- Perform bus read-modify-write cycles to apply Boolean operators efficiently to device data.

The kernel virtual address returned by PIO mapping is not a physical memory address and is not a bus address. The kernel virtual address and the VME or EISA bus address need not have any bits in in common.

The functions used with PIO maps are summarized in Table 14-1.

Table 14-1 Functions to Create and Use PIO Maps

Function	Header Files	Can Sleep	Purpose
pio_mapalloc(D3)	pio.h & types.h	Y	Allocate a PIO map.
pio_mapfree(D3)	pio.h & types.h	N	Free a PIO map.
pio_badaddr(D3)	pio.h & types.h	N	Check for bus error when reading an address.
pio_badaddr_val(D3)	pio.h & types.h	N	Check for bus error when reading an address and return the value read.
pio_wbadaddr(D3)	pio.h & types.h	N	Check for bus error when writing to an address.
pio_wbadaddr_val(D3)	pio.h & types.h	N	Check for bus error when writing a specified value to an address.
pio_mapaddr(D3)	pio.h & types.h	N	Convert a bus address to a virtual address.
pio_bcopyin(D3)	pio.h & types.h	Y	Copy data from a bus address to kernel's virtual space.
pio_bcopyout(D3)	pio.h & types.h	Y	Copy data from kernel's virtual space to a bus address.
pio_andb_rmw(D3)	pio.h & types.h	N	Byte read-and-write.
pio_andh_rmw(D3)	pio.h & types.h	N	16-bit read-and-write.
pio_andw_rmw(D3)	pio.h & types.h	N	32-bit read-and-write.
pio_orb_rmw(D3)	pio.h & types.h	N	Byte read-or-write.
pio_orh_rmw(D3)	pio.h & types.h	N	16-bit read-or-write.
pio_orw_rmw(D3)	pio.h & types.h	N	32-bit read-or-write.

A kernel-level device driver creates a PIO map by calling **pio_mapalloc()**. This function performs memory allocation and so can sleep. PIO maps are typically created in the *pfx***edtinit()** entry point, where the driver first learns about the device addresses from the contents of the *edt_t* structure (see "Entry Point edtinit()" on page 144).

The parameters to **pio_mapalloc()** describe the range of addresses that can be mapped in terms of

- the bus type, ADAP_VME or ADAP_EISA from *sys/edt.h*
- the bus number, when more than one bus is supported
- the address space, using constants such as PIOMAP_A24N or PIOMAP_EISA_IO from *sys/pio.h*
- the starting bus address and a length

This call also specifies a “fixed” or “unfixed” map. The usual type is “fixed.” For the differences, see “Fixed PIO Maps” on page 328 and “Unfixed PIO Maps” on page 329.

A call to **pio_mapfree()** releases a PIO map. PIO maps created by a loadable driver must be released in the *pfx***unload()** entry point (see “Entry Point unload()” on page 167 and “Unloading” on page 240).

Testing the PIO Map

The PIO map is created from the parameters that are passed. These are not validated by **pio_mapalloc()**. If there is any possibility that the mapped device is not installed, not active, or improperly configured, you should test the mapped address.

The **pio_badaddr()** and **pio_badaddr_val()** functions test the mapped address to see if it is usable for input. Both functions perform the same operation: operating through a PIO map, they test a specified bus address for validity by reading 1, 2, 4, or 8 bytes from it. The **pio_badaddr_val()** function returns the value that it reads while making the test. This can simplify coding, as shown in Example 14-1.

Example 14-1 Comparing pio_badaddr() to pio_badaddr_val()

```
unsigned int gotvalue;
piomap_t *themap;
/* Using only pio_badaddr() */
   if (!pio_badaddr(themap,CTLREG,4)
   {
      (void) pio_bcopyin(themap,CTLREG,&gotvalue,4,4,0);
      ...use "gotvalue"
/* Using pio_badaddr_val() */
   if (!pio_badaddr_val(themap,CTLREG,4,&gotvalue))
   {
      ...use "gotvalue"
```

The **pio_wbadaddr()** function tests a mapped device address for writeability. The **pio_wbadaddr_val()** not only tests the address but takes a specific value to write to that address in the course of testing it.

Using the Mapped Address

From a fixed PIO map you can recover a kernel virtual address that corresponds to the first bus address in the map. The **pio_mapaddr()** function is used for this.

You can use this address to load or store data into device registers. In the *pfx***map()** entry point (see "Concepts and Use of mmap()" on page 158), you can use this address with the **v_mapphys()** function to map the range of device addresses into the address space of a user process.

You cannot extract a kernel address from an unfixed PIO map, as explained under "Unfixed PIO Maps" on page 329.

Using the PIO Map in Functions

You can apply a variety of kernel functions to any PIO map, fixed or unfixed. The **pio_bcopyin()** and **pio_bcopyout()** functions copy a range of data between memory and a fixed or unfixed PIO map. These functions are optimized to the hardware that exists, and they do all transfers in the largest size possible (32 or 64 bits per transfer). If you need to transfer data in specific sizes of 1 or 2 bytes, use direct loads and stores to the mapped addresses.

The series of functions **pio_andb_rmw()** and **pio_orb_rmw()** perform a read-modify-write cycle on the VME bus. You can use them to set or clear bits in device registers. A read-modify-write cycle is faster than a load followed by a store since it uses fewer system bus cycles.

Fixed PIO Maps

On a Challenge or Onyx system, a PIO map can be either "fixed" or "unfixed." This attribute is specified when the map is created. (On a Crimson system, only fixed maps are created regardless of which type is requested.)

The Challenge and Onyx architecture provides for a total of 15 separate, 8 MB windows on VME address space for each VME bus. Two of these are permanently reserved to the kernel, and one window is reserved for use with unfixed mappings. The remaining 12 windows are available to implement fixed PIO maps.

When the kernel creates a fixed PIO map, the map is associated with one of the 12 available VME mapping windows. The kernel tries to be clever, so that whenever a PIO map falls within an 8 MB window that already exists, the PIO map uses that window. If the desired VME address is not covered by an open window, one of the twelve windows for that bus is opened to expose a mapping for that address.

It is possible in principle to configure thirteen devices that are scattered so widely in the A32 address space that twelve, 8 MB windows cannot cover all of them. In that unlikely case, the attempt to create the thirteenth fixed PIO map will fail for lack of a mapping window.

In order to prevent this, simply configure your PIO addresses into a span of at most 96 MB per bus (see “Configuring Device Addresses” on page 310).

Unfixed PIO Maps

When you create an unfixed PIO map, the map is not associated with any of the twelve mapping windows. As a result, the map cannot be queried for a kernel address that might be saved, or mapped into user space.

You can use an unfixed map with kernel functions that copy data or perform read-modify-write cycles. These functions use the one mapping window that is reserved for unfixed maps, repositioning it in VME space if necessary.

The *lboot* command uses an unfixed map to perform the *probe* and *exprobe* sequences from VECTOR statements (see “Configuring the System Files” on page 311). As a result, these probes do not tie up mapping windows.

Mapping DMA Addresses

A DMA map is a system object that represents a mapping between a buffer in kernel virtual space and a range of VME bus addresses. After creating a DMA map, a driver uses the map to specify the target address and length to be programmed into a VME bus master before a DMA transfer.

The functions that operate on DMA maps are summarized in Table 14-2.

Table 14-2 Functions That Operate on DMA Maps

Function	**Header Files**	**Can Sleep**	**Purpose**
dma_map(D3)	dmamap.h & types.h & sema.h	N	Load DMA mapping registers for an imminent transfer.
dma_mapbp(D3)	dmamap.h & types.h & sema.h	N	Load DMA mapping registers for an imminent transfer.
dma_mapaddr(D3)	dmamap.h & types.h & sema.h	N	Return the "bus virtual" address for a given map and address.
dma_mapalloc(D3)	dmamap.h & types.h & sema.h	Y	Allocate a DMA map.
dma_mapfree(D3)	dmamap.h & types.h & sema.h	N	Free a DMA map.
vme_adapter(D3)	vmereg.h & types.h	N	Determine VME adapter that corresponds to a given memory address.

A device driver allocates a DMA map using **dma_mapalloc()**. This is typically done in the *pfx***edtinit()** entry point, provided that the maximum I/O size is known at that time (see "Entry Point edtinit()" on page 144). The important argument to **dma_mapalloc()** is the maximum number of pages (I/O pages, the unit is IO_NBPP declared in *sys/immu.h*) to be mapped at one time.

Note: In the Challenge and Onyx systems, a limit of 64 MB of mapped DMA space per VME adapater is imposed by the hardware. Some few megabytes of this are taken early by system drivers. Owing to a bug in IRIX 5.3 and 6.1, a request for 64 MB or more is not rejected, but waits forever. However, in any release, a call to **dma_mapalloc()** that requests a single map close to the 64 MB limit is likely to wait indefinitely for enough map space to become available.

DMA maps created by a loadable driver should be released in the *pfx***unload()** entry point (see “Entry Point unload()” on page 167 and “Unloading” on page 240).

Using a DMA Map

A DMA map is used prior to a DMA transfer into or out of a buffer in kernel virtual space. The function **dma_map()** takes a DMA map, a buffer address, and a length. It assigns a span of contiguous VME addresses of the specified length, and sets up a mapping between that range of VME addresses and the physical addresses that represent the specified buffer.

When the buffer spans two or more physical pages (IO_NBPP units), **dma_map()** sets up a scatter/gather operation, so that the VME bus controller will place the data in the appropriate page frames.

It is possible that **dma_map()** cannot map the entire size of the buffer. This can occur only when the buffer spans two or more pages, and is caused by a shortage of mapping registers in the bus adapter. The function maps as much of the buffer as it can, and returns the length of the mapped data.

You must always anticipate that **dma_map()** might map less than the requested number of bytes, so that the DMA transfer has to be done in two or more operations.

Following the call to **dma_map()**, you call **dma_mapaddr()** to get the bus virtual address that represents the first byte of the buffer. This is the address you program into the bus master device (using a PIO store), in order to set its starting transfer address. Then you initiate the DMA transfer (again by storing a command into a device register using PIO).

Allocating an Interrupt Vector Dynamically

When a VME device generates an interrupt, the Silicon Graphics VME controller initiates an interrupt acknowledge (IACK) cycle on the VME bus. During this cycle, the interrupting device presents a data value that characterizes the interrupt. This is the *interrupt vector*, in VME terminology.

According to the VME standard, the interrupt vector can be a data item of 8, 16, or 32 bits. However, Silicon Graphics systems accept only an 8-bit vector, and its value must fall in the range 1-254 inclusive. (0x00 and 0xFF are excluded because they could be generated by a hardware fault.)

The interrupt vector returned by some VME devices is hard-wired or configured into the board with switches or jumpers. When this is the case, the vector number should be written as the *vector* parameter in the VECTOR statement that describes the device (see "Configuring the System Files" on page 311).

Some VME devices are programmed with a vector number at runtime. For these devices, you omit the *vector* parameter, or give its value as an asterisk. In the device driver, you use the functions in Table 14-3 to choose a vector number.

Table 14-3 Functions to Manage Interrupt Vector Values

Function	Header Files	Can Sleep	Purpose
vme_ivec_alloc(D3)	vmereg.h & types.h	N	Allocate a VME bus interrupt vector.
vme_ivec_free(D3)	vmereg.h & types.h	N	Free a VME bus interrupt vector.
vme_ivec_set(D3)	vmereg.h & types.h	N	Register a VME bus interrupt vector.

Allocating a Vector

In the *pfx***edtinit()** entry point, the device driver selects a vector number for the device to use. The best way to select a number is to call **vme_ivec_alloc()**, which returns a number that has not been registered for that bus, either dynamically or in a VECTOR line.

The driver then uses **vme_ivec_set()** to register the chosen vector number. This function takes parameters that specify

- The vector number
- The bus number to which it applies
- The address of the interrupt handler for this vector—typically but not necessarily the name of the *pfx***intr()** entry point of the same driver
- An integer value to be passed to the interrupt entry point—typically but not necessarily the vector number

The **vme_ivec_set()** function simply registers the number in the kernel, with the following two effects:

- The **vme_ivec_alloc()** function does not return the same number to another call until the number is released.
- The specified handler is called when any device presents this vector number on an interrupt.

Multiple devices can present the identical vector, provided that the interrupt handler has some way of distinguishing one device from another.

Note: If you are working with both the VME and EISA interfaces, it is worth noting that the number and types of arguments of **vme_ivec_set()** differ from the similar EISA support function **eisa_ivec_set()**.

Releasing a Vector

There is a limit of 254 vector numbers per bus, so it is a good idea for a loadable driver, in its *pfx***unload()** entry point, to release a vector by calling **vme_ivec_free()** (see "Entry Point unload()" on page 167 and "Unloading" on page 240).

Vector Errors

A common problem with programmable vectors in the Challenge or Onyx systems is the appearance of the following warning in the SYSLOG file:

```
Warning: Stray VME interrupt: vector =0xff
```

One possible cause of this error is that the board is emitting the wrong interrupt vector; another is that the board is emitting the correct vector but with the wrong timing, so that the VME bus adapter samples all-binary-1 instead. Both these conditions can be verified with a VME bus analyzer. In the Challenge or Onyx hardware design, the most likely cause is the presence of empty slots in the VME card cage. All empty slots must be properly jumpered in order to pass interrupts correctly.

Supporting Early IO4 Cache Problems

VME drivers that support DMA to buffers that are not cache-aligned multiples of a cache-line need to take special precautions in a Challenge system; see Appendix B, "Challenge DMA with Multiple IO4 Boards."

Sample VME Device Driver

The source module displayed in Example 14-2 contains a complete character device driver for a hypothetical VME device. Although it is a character driver, it contains a strategy routine (the **cdev_strategy()** function). Both the *pfx***read()** and *pfx***write()** entry points call the strategy routine to perform the actual I/O. As a result, this driver could be installed as either a block device driver or a character driver, or as both.

The driver is multiprocessor-aware, so its *pfx***devflag** global contains D_MP. It uses two locks. A basic lock (*board.cd_lock*) is used for short-term mutual exclusion, to block a potential race between the strategy routine and the interrupt routine. A semaphore (*board.cd_rwsema*) is used for long-term mutual exclusion to make sure that only one process uses the device for reading or writing at any time.

Example 14-2 Example VME Character Driver

```
/*********************************************************************\
*       File:           cdev.c                                        *
*                                                                     *
*       The following is an example of how a device driver for a VME    *
*       character device might be written.  The sample driver          *
*       illustrates how to write code which performs DMA into both      *
*       kernel and user address space, as well as how a sample          *
*       driver's registers would be mapped into user address space.     *
*                                                                     *
\*********************************************************************/
#include <sys/types.h>          /* Contains basic kernel typedefs */
#include <sys/param.h>
#include <sys/immu.h>           /* Contains VM-specific definitions (map) */
#include <sys/region.h>         /* Contains VM data structure defs (map) */
#include <sys/conf.h>           /* Contains cdevsw and driver flag defs */
#include <sys/vmereg.h>         /* Contains VME bus-specific definitions */
#include <sys/edt.h>            /* Contains definition of edt structs */
#include <sys/dmamap.h>         /* Definitions for dma structs and routines */
#include <sys/pio.h>            /* Definitions for pio structs and routines */
#include <sys/cmn_err.h>        /* Definitions for cmn_err constants */
#include <sys/errno.h>          /* Define classic error numbers */
#include <sys/open.h>           /* Define open types used in otyp open parm */
#include <sys/cred.h>           /* Contains credential structure declaration */
#include <sys/ksynch.h>         /* Define ddi-compliant synch primitives */
#include <sys/sema.h>           /* Include semaphore prototypes */
#include <sys/ddi.h>            /* Include the ddi-compliant stuff */
/* Some constants used throughout the driver */
#define CDEV_MAX_XFERSIZE 65536
#define VALID_DEVICE 0x0acedeed
/* The following structure is provided so that we can memory map the
 * device's control registers.  For purposes of illustration, we
 * provide a couple of generic registers; a real device would have
 * completely different mappings.
 */
#define CMD_READ        0x1
#define CMD_WRITE       0x2
#define CMD_CLEAR_INTR  0x4
#define CMD_RESET       0x8
typedef struct deviceregs_s {
    volatile unsigned short  cr_status; /* The device's status register */
    volatile unsigned short  cr_cmd;    /* The device's command register */
    volatile unsigned int    cr_dmaaddr;/* The DMA address */
    volatile unsigned int    cr_count;  /* The number of bytes to xfer */
    volatile unsigned int    cr_devid;  /* The device ID register */
    volatile unsigned int    cr_parm;   /* A device parameter */
} deviceregs_t;
/* The cdevboard structure contains about a device which the
 * driver needs to maintain.  In general, each instance of a
 * device in the system has an associated cdevboard structure
 * which contains driver-specific information about that board.
 */
#define STATUS_PRESENT          0x1
#define STATUS_OPEN             0x2
#define STATUS_INTRPENDING      0x4
#define STATUS_TIMEOUT          0x8
#define FLAG_SET(_x, _y)        (((_x)->cd_status) |= (_y))
#define FLAG_CLEAR(_x, _y)      (((_x)->cd_status) &= (~(_y)))
#define FLAG_TEST(_x, _y)       (((_x)->cd_status) & (_y))
typedef struct cdevboard_s {
    lock_t             cd_lock;        /* Used for mutual exclusion */
    sema_t             cd_rwsema;      /* Prevents simult. read & write */
    volatile deviceregs_t *cd_regs;    /* Memory-mapped control regs */
    dmamap_t           *cd_map;        /* DMA Map for this device */
    unsigned int       cd_ctlr;        /* The controller # of this device */
    unsigned int       cd_status;      /* The board’s status. */
    unsigned int       cd_strayintr;   /* Counts stray interrupts */
    struct buf        *cd_buf;         /* Pointer to buffer */
    unsigned int       cd_count;       /* Count of bytes being transferred */
    toid_t             cd_tout;        /* Timeout handle */
} cdevboard_t;
/* We need to tell the kernel what kind of interface this driver
 * expects.  For a simple, non-MP driver, the devflag can be set to
 * 0.  Since we’re going to be a little more ambitious, we’ll tell
 * the kernel that we are capable of running MP.
 */
int cdev_devflag = D_MP;
/* Forward declarations of general driver functions */
int  cdev_intr(int board);
int  cdev_strategy(struct buf *bp);
void cdev_timeout(cdevboard_t *board);
/* Driver global data structures; to minimize memory use, we create
 * an array of pointers to audioboard structures and only allocate the
 * actual structure if the corresponding board is configured.
 */
#define CDEV_MAX_BOARDS 4
static cdevboard_t *CDevBoards[CDEV_MAX_BOARDS + 1];
#if DEBUG
#define DPRINTF(_x)     debug_printf _x
```

```
void debug_printf(char *fmt, ...)
{
        va_list ap;
        extern void icmn_err();
        va_start(ap, fmt);
        icmn_err(CE_NOTE, fmt, ap);
        va_end(ap);
}
#else
#define DPRINTF(_x)
#endif

/************************************************************************
 * edtinit is the first routine all VME drivers need to provide.
 * This function is called early during kernel initialization, and
 * drivers generally use it to set up driver-global data structures
 * and device mappings for any devices which exist.  The kernel calls
 * it once for each VECTOR line in the appropriate .sm file.
 */
void
cdev_edtinit(struct edt *e)
{
    piomap_t *piomap;           /* Control register mapping descriptor */
    dmamap_t *dmamap;           /* DMA mapping for read/write buffers */
    volatile deviceregs_t *base;   /* Base address of device's control regs */
    vme_intrs_t *intrs;         /* Pointer to VME interrupt information */
    int intr_vec;               /* Actual vector to use */
    int ctlr;                   /* Board number to be configured */
    cdevboard_t *board;         /* New board data structure */
    /* Make sure that the the controller number is within range */
    ctlr = e->e_ctlr;
    if (ctlr < 0 || ctlr > CDEV_MAX_BOARDS) {
        cmn_err(CE_WARN, "cdev%d: controller number is invalid", ctlr);
        return;
    }
    /* Allocate a programmed I/O mapping structure for the particular
     * device.  The kernel uses the data in the e_space field to figure
     * out both the VME base address and the total size of the register area.
     */
    piomap = pio_mapalloc(e->e_bus_type, e->e_adap, e->e_space,
                      PIOMAP_FIXED, "cdev");
    /* XXX Check for the success of piomap allocation */
    if (piomap == (piomap_t *)NULL){
        cmn_err(CE_WARN, "cdev%d: Could not allocate piomap", ctlr);
        return;
```

```
}
/* Now that the map is allocated, we position it so that it overlays
 * the device’s hardware registers.  Since this is a fixed map, we
 * just pass in the base address of the control register range.
 * iobase comes from the VECTOR line in the .sm file.
 */
base = (volatile deviceregs_t*) pio_mapaddr(piomap, e->e_iobase);
/* We’re going to need to DMA map the user’s buffer during read and
 * write requests, so we preallocate a fixed number of dma mapping
 * entries based on the constant CDEV_MAX_XFERSIZE.  If we allowed
 * multiple users to perform reads and writes simultaneously we’d
 * probably want to allocate one map for reads and one for writes.
 * Since we only allow one operation to occur at any given time,
 * though, we can get away with only one.
 *
 * IMPORTANT NOTE: There are only a limited number of dma mapping
 * registers available in a system; you should be somewhat conservative
 * in your use of them.  It is reasonable to consume up to 100 per
 * device (you can use more if you expect that only a couple devices
 * will be attached for each driver.  If, for example, this driver
 * will never control more than two devices, you could probably use
 * up to 512 mapping registers for each device.  If however, you’d expect
 * to see hundreds of devices, you’d need to be more conservative.
 */
dmamap = dma_mapalloc(DMA_A24VME, e->e_adap,
                      io_btoc(CDEV_MAX_XFERSIZE) + 1, 0);
if (dmamap == (dmamap_t*) NULL) {
    cmn_err(CE_WARN, “cdev%d: Could not allocate dmamaps”, ctlr);
    pio_mapfree(piomap);
    return;
}
/* The next step would be to probe the device to determine whether
 * it is actually present.  To do this, we attempt to read some
 * registers which behave in a manner unique to this particular
 * hardware.  We need to protect ourselves in the event that the
 * device isn’t actually present, however, so we use the badaddr
 * and wbadaddr routines.  For our example, we assume that the
 * device is present if it’s device
 */
if ((badaddr(&(base->cr_devid), 4) == 0) &&
    (base->cr_devid == VALID_DEVICE)) {
    DPRINTF((“cdev%d: found valid device”, ctlr));
} else {
    /* It doesn’t look like the device is there. */
    cmn_err(CE_WARN, “cdev%d: cannot find actual device”, ctlr);
    pio_mapfree(piomap);
    dma_mapfree(dmamap);
    return;
}
/* Now we set up the interrupt for this device.
 * It is possible to specify a vector and priority level on the
 * VECTOR line in the .sm file, so we check to see if such was the case.
 */
intrs = (vme_intrs_t*) e->e_bus_info;
intr_vec = intrs->v_vec;
/* If intr_vec is non-zero, user specified specific vec in .sm file.
 * If the interrupt was specified on the VECTOR line, the kernel has
 * already established a vector for us, so we don’t need to do it
 * ourselves.
 */
if (intr_vec == 0) {
    intr_vec = vme_ivec_alloc(e->e_adap);
    /* Make sure that we got a good interrupt vector */
    if (intr_vec == -1) {
        cmn_err(CE_WARN, “cdev%d: could not allocate intr vector\n”, ctlr);
        pio_mapfree(piomap);
        dma_mapfree(dmamap);
        return;
    }
    /* Associate this driver’s interrupt routine with the acquired vec */
vme_ivec_set(e->e_adap, intr_vec, cdev_intr, 0);
}
/* Initialize the board structure for this board */
board = (cdevboard_t*) kmem_alloc(sizeof(cdevboard_t));
if (board == (void*) 0) {
    cmn_err(CE_WARN, “cdev%d: kmem_alloc failed”, ctlr);
    pio_mapfree(piomap);
    dma_mapfree(dmamap);
    /* XXX Need to check whether it is allocated?? */
    vme_ivec_free(e->e_adap, intr_vec);
    return;
}
board = CDevBoards[ctlr];
board->cd_regs = base;
board->cd_ctlr = ctlr;
board->cd_status = STATUS_PRESENT;
board->cd_strayintr = 0;
board->cd_map = dmamap;
initnsema(&board->cd_rwsema, 1, “CDevRWM”);
/* Finally, call any one-time-only device initialization routines;
```

```
     * this particular device doesn’t have any.
     */
    return;
}
/************************************************************************
 * cdev_open -- When opening a device, we need to check for mutual
 * exclusion (if desired) and then set up an additional data structures
 * if this is the first time the device has been opened.  Remember that
 * the OS usually doesn’t call close until all users close the device,
 * so you can’t count on being able to set up unique data for each user
 * of the device unless you either disallow multiple opens at the same time
 * or mark the device as being a layered (otype = O_LYR) device.
 */
int
cdev_open(dev_t *dev, int flag, int otyp, cred_t *cred)
{
    minor_t            ctlr;           /* Controller # of cdev being opened */
    cdevboard_t        *board;         /* per-board data for opened cdev*/
    int                s;              /* Opaque lock value */
    /* We assume that the minor number encodes the ctlr number, so
     * we just go ahead and use it to index the CDevBoards array once
     * we’ve validated it.
     */
    ctlr = geteminor(*dev);
    if (ctlr > CDEV_MAX_BOARDS) {
        DPRINTF((“cdev%d: open: minor number out of range”, ctlr));
        return ENXIO;
    }
    board = CDevBoards[ctlr];
    if (FLAG_TEST(board, STATUS_PRESENT) || (board->cd_ctlr != ctlr)) {
        DPRINTF((“cdev%d: open: device not found”, ctlr));
        return ENXIO;
    }
    /* If exclusiveness is desired, we now need to atomically insure that
     * we are the owners of the device.
     */
    s = LOCK(&board->cd_lock, splhi);
    if (FLAG_TEST(board, STATUS_OPEN)) {
        UNLOCK(&board->cd_lock, s);
        return EBUSY;
    } else {
        ASSERT(board->cd_status == STATUS_PRESENT);
        FLAG_SET(board, STATUS_OPEN);
    }
    UNLOCK(&board->cd_lock, s);
    return 0;
}

/*************************************************************************
 * cdev_close -- Called when the open reference count drops to zero.
 *      Cleans up any leftover data structure and marks the device as
 *      available.
 */
int
cdev_close(dev_t dev, int flag, int otyp, cred_t *cred)
{
    int                 ctlr;           /* Controller # of dev being closed */
    cdevboard_t         *board;         /* per-board data structure */
    ctlr = geteminor(dev);
    ASSERT(ctlr <= CDEV_MAX_BOARDS);
    board = CDevBoards[ctlr];
    ASSERT(board && FLAG_TEST(board, STATUS_OPEN|STATUS_PRESENT));
    /* Do any cleanup required here */
    /* Reset the board's status flags (to clear the OPEN flag) */
    FLAG_CLEAR(board, STATUS_OPEN);
    return 0;
}
/*************************************************************************
 * cdev_intr -- Called when an interrupt occurs.  We check to see if
 *      a process was waiting for an I/O operation to complete and
 *      re-activate that process if such is the case.
 */
#ifdef EVEREST /* IO4 fix for Challenge */
extern int io4_flush_cache(caddr_t piomap);
#endif
int
cdev_intr(int ctlr)
{
    cdevboard_t         *board;         /* per-board data structure pointer */
        int s;  /* lock return value */
    /* Make sure that the controller value is legitimate */
    ASSERT(ctlr <= CDEV_MAX_BOARDS);
    board = CDevBoards[ctlr];
    ASSERT(board && FLAG_TEST(board, STATUS_PRESENT));
#ifdef EVEREST /* flush IO4 cache */
    (void)io4_flush_cache((caddr_t)board->cd_regs);
#endif
        /*
         * Get exclusive use of the board. This ensures that the strategy
         * routine is completely finished setting STATUS_INTRPENDING before
         * we examine it.
         */
        s = LOCK(&board->cd_lock, splhi);
    /* It’s possible that we could get a stray interrupt if the hardware
     * is flaky, so we keep a count of bogus interrupts and ignore them.
     */
    if (FLAG_TEST(board, STATUS_OPEN|STATUS_INTRPENDING)) {
        board->cd_strayintr++;
        return 0;
    }
    /* Acknowledge the interrupt from the device */
    board->cd_regs->cr_cmd = CMD_CLEAR_INTR;
    FLAG_CLEAR(board, STATUS_INTRPENDING);
    /* Remove the timeout request */
    untimeout(board->cd_tout);
    /* Update the buffer’s parameters */
    ASSERT(board->cd_buf->b_bcount > 0);
    board->cd_buf->b_bcount     -= board->cd_count;
    board->cd_buf->b_dmaaddr    += board->cd_count;
    /* Release the mutual exclusion on the board. */
    UNLOCK(&board->cd_lock,s);
    /* If the transfer count is 0, then we’ve transferred all of the
     * bytes in the request, so we call iodone to awaken the user process.
     * Otherwise, we call cdev_strat to initiate another transfer.
     */
    if (board->cd_buf->b_bcount == 0)
        iodone(board->cd_buf);
    else
        cdev_strategy(board->cd_buf);
    return 0;
}
/*************************************************************************
 * cdev_read -- reads data from the device.  We employ the uiophysio
 *      routine to perform all the requisite mapping of the buffer
 *      for us and then call the cdev_strat routine.  The big advantage
 *      of uiophysio() is that it sets up memory such that the device can
 *      DMA directly into the user address space.  The strategy routine
 *      is responsible for actually setting up and initiating the transfer.
 *      The process will block in uiophysio until the interrupt handler
 *      calls iodone() on buffer pointer.
 */
int
cdev_read(dev_t dev, uio_t *uio, cred_t *cred)
{
    int         ctlr;
    cdevboard_t *board;
    int         error = 0;
    ASSERT(ctlr >= 0 && ctlr <= CDEV_MAX_BOARDS);
    ctlr = geteminor(ctlr);
    ASSERT(board && FLAG_TEST(board, STATUS_OPEN|STATUS_PRESENT));
    board = CDevBoards[ctlr];
    /* Since we allocated only a single DMA buffer, we need to block
     * if a previous transfer hasn't completed.
     */
    psema(&board->cd_rwsema, PZERO+1);
    error = uiophysio(cdev_strat, NULL, dev, B_READ, uio);
    /* Check to see if the transfer timed out */
    if (FLAG_TEST(board, STATUS_TIMEOUT)) {
        FLAG_CLEAR(board, STATUS_TIMEOUT);
        error = EIO;
    }
    vsema(&board->cd_rwsema);
    return error;
}
/************************************************************************
 * cdev_write -- writes data from a user buffer to the device.
 *      We employ the uiophysio routine to set up the mappings for us.
 *      Once the mappings are established, uiophysio will call the
 *      given strategy routine (cdev_strat) with a buffer pointer.
 *      The strategy routine is then responsible for kicking off the
 *      transfer.  The process will block in uiophysio until the
 *      interrupt handler calls iodone() on the buffer pointer.
 */
int
cdev_write(dev_t dev, uio_t *uio, cred_t *cred)
{
    int         ctlr;
    cdevboard_t *board;
    int         error = 0;
    ASSERT(ctlr >= 0 && ctlr <= CDEV_MAX_BOARDS);
    ctlr = geteminor(ctlr);
    ASSERT(board && FLAG_TEST(board, STATUS_OPEN|STATUS_PRESENT));
    board = CDevBoards[ctlr];
    psema(&board->cd_rwsema, PZERO+1);
    error = uiophysio(cdev_strat, NULL, dev, B_WRITE, uio);
    /* Check to see if the transfer timed out */
    if (FLAG_TEST(board, STATUS_TIMEOUT)) {
        FLAG_CLEAR(board, STATUS_TIMEOUT);
        error = EIO;
    }
    vsema(&board->cd_rwsema);
    return error;
}
/*************************************************************************
 * cdev_strat -- Called by uiophysio, cdev_strat actually performs all
 *      the device-specific actions needed to initiate the transfer,
 *      such as establishing the DMA mapping of the transfer buffer and
 *      actually programming the device.  There is an implicit assumption
 *      that the device will interrupt at some later point when the I/O
 *      operation is complete.
 */
int
cdev_strategy(struct buf *bp)
{
    int ctlr;                   /* Controller # being accessed */
    cdevboard_t *board;         /* Board data structure */
    int mapcount;               /* Count */
    int s;                      /* opaque lock value */

    /* Get a reference to the actual board structure */
    ctlr = geteminor(bp->b_edev);
    ASSERT(ctlr >= 0 && ctlr <= CDEV_MAX_BOARDS);
    board = CDevBoards[ctlr];
    ASSERT(board && FLAG_TEST(board, STATUS_OPEN|STATUS_PRESENT));
    /* We start by mapping the appropriate region into VME address space.
     * Because of the mapping registers we don't have to worry about the
     * fact that the physical pages backing the data regions may be
     * physically discontinuous; in effect, the DMA mapping is taking the
     * place of scatter/gather hardware.  Nonetheless, in order to avoid
     * consuming an excessive number of translation entries we limit the
     * size of the transfer to CDEV_MAX_XFERSIZE.
     */
    mapcount = MIN(bp->b_bcount, CDEV_MAX_XFERSIZE);
    mapcount = dma_map(board->cd_map, bp->b_dmaaddr, mapcount);
    ASSERT(mapcount > 0);
    /* Before starting the I/O, get exclusive use of the board struct.
     * This ensures that, if this CPU is interrupted and we are slow to
     * set STATUS_INTRPENDING, cdev_intr() will be locked out until we do.
     */
    s = LOCK(&board->cd_lock, splhi);
    /* Now we start the transfer by writing into memory-mapped registers */
    board->cd_regs->cr_dmaaddr = dma_mapaddr(board->cd_map, bp->b_dmaaddr);
    board->cd_regs->cr_count = mapcount;
    board->cd_regs->cr_cmd = ((bp->b_flags & B_WRITE) ? CMD_WRITE : CMD_READ);
    /* Schedule a timeout, just in case the device decides to hang forever */
    itimeout(cdev_timeout, board, 2000, splhi);
    /* Finally, we update some of the board data structures */
    board->cd_buf   = bp;
    board->cd_count = mapcount;
    FLAG_SET(board, STATUS_INTRPENDING);
    /* Release the board struct, so the interrupt handler can use it. */
    UNLOCK(&board->cd_lock, s);
    /* Upon returning, uiophysio will block until cdev_intr calls iodone() */
    return 0;
}
/*************************************************************************
 * cdev_ioctl -- Not too exciting.  We'll assume that the device has
 *      one controllable parameter which can be both written and received.
 *      To help users avoid errors, we use unusual constants for the ioctl
 *      values.  In a real driver, the CDIOC definitions would go into a
 *      header file.
 */
#define CDIOC_SETPARM   0xcd01
#define CDIOC_GETPARM   0xcd02
int
cdev_ioctl(dev_t dev, int cmd, int arg, int mode, cred_t *cred)
{
    int                 ctlr;           /* Controller number */
    cdevboard_t        *board;          /* Per-controller data */
    int                 error = 0;      /* Error return value */
    ctlr = geteminor(dev);
    ASSERT(ctlr >= 0 && ctlr <= CDEV_MAX_BOARDS);
    board = CDevBoards[ctlr];
    ASSERT(board && FLAG_TEST(board, STATUS_OPEN|STATUS_PRESENT));
    switch (cmd) {
      case CDIOC_SETPARM:
        board->cd_regs->cr_parm = arg;
        break;
      case CDIOC_GETPARM:
        {
            int value;
            value = board->cd_regs->cr_parm;
            if (copyout(&value, (void*) arg, sizeof(int)))
                error = EFAULT;
        }
        break;
      default:
        error = EINVAL;
        break;
    }
    return error;
}
/************************************************************************
 * cdev_timeout -- If an I/O request takes a really long time to complete
 *      for some reason (if, for example, someone takes the device offline),
 *      it is better to warn the user than to simply hang.  This timeout
 *      routine will cancel any pending I/O requests and display a message.
 *      A more sophisticated routine might try resetting the device and
 *      re-executing the operation.
 */
void
cdev_timeout(cdevboard_t *board)
{
    /* Clear the pending request from the device.  This operation
     * is extremely dependent on the actual device.  This driver
     * pretends that we simply can use the reset command.
     */
    board->cd_regs->cr_cmd = CMD_RESET;
    /* Make a note that the operation timed out */
    FLAG_SET(board, STATUS_TIMEOUT);
    /* Display a warning */
    cmn_err(CE_WARN, “cdev%d: device timed out”, board->cd_ctlr);
    /* Notify the user process that the operation has “finished”. */
    iodone(board->cd_buf);
}
/************************************************************************
 * cdev_map -- For illustrative purposes, we show how one would go about
 *      mapping the device’s control registers.
 */
int
cdev_map(dev_t dev, vhandl_t *vt, off_t off, int len, int prot)
{
    int                ctlr;          /* Controller number */
    cdevboard_t       *board;         /* Per-controller data */
    ctlr = geteminor(dev);
    ASSERT(ctlr >= 0 && ctlr <= CDEV_MAX_BOARDS);
    board = CDevBoards[ctlr];
    ASSERT(board && FLAGS_TEST(board, STATUS_OPEN|STATUS_PRESENT));
    if (v_mapphys(vt, (void*) board->cd_regs, len))
        return ENOMEM;
    else
        return 0;
}
/************************************************************************
 * cdev_unmap -- Called when a region is unmapped.  We don’t actually
 *      need to do anything.
 */
int
cdev_unmap(dev_t dev, vhandl_t *vt)
{
    /* No need to do anything here; unmapping is handled by upper levels
     * of the kernel.
     */
    return 0;
}
```

PART FIVE

SCSI Device Drivers

Chapter 15: SCSI Device Drivers
Actual control of the SCSI bus is managed by one or more Host Adapter drivers. This chapter tells how SCSI device drivers use these facilities.

Chapter 15

SCSI Device Drivers

All Silicon Graphics systems support the Small Computer Systems Interface (SCSI) bus for the attachment of disks, tapes, and other devices. This chapter details the kernel-level support for SCSI device drivers.

If your aim is to control a SCSI device from a user-level process, this chapter contains some useful background information to supplement Chapter 5, "User-Level Access to SCSI Devices." If you are designing a kernel-level SCSI driver, this chapter contains essential information on kernel support. The major topics in this chapter are as follows:

- "SCSI Support in Silicon Graphics Systems" on page 352 gives an overview of the hardware and software support for SCSI.
- "Host Adapter Facilities" on page 354 documents the use of the host adapter driver to access a SCSI device.
- "Designing a SCSI Driver" on page 369 notes design differences from other driver types, and includes an example driver skeleton.
- "Example SCSI Device Driver" on page 370 lists a skeleton driver to illustrate the use of the interface.
- "Designing a Host Adapter Driver" on page 375 documents the facility for creating and installing customized host adapter drivers.
- "SCSI Reference Data" on page 377 tabulates SCSI codes and messages for reference.

In addition, you may want to review the following additional sources:

intro(7) reference page	Documents the naming conventions for disk and tape device special files.
dksc(7) reference page	Documents the Silicon Graphics disk volume partition layout and the ioctl support in the base-level SCSI drivers.
ANSI *X3.131-1986* and *X3T9.2/85-52 Rev 4B*.	SCSI standards documents.
http://www.abekrd.co.uk/SCSI2/	Web page containing the complete SCSI-2 standard in HTML form.

SCSI Support in Silicon Graphics Systems

All current Silicon Graphics systems rely on the SCSI bus as the primary attachment for disks and tapes. The IRIX kernel provides extensive support for OEM drivers for SCSI devices.

Note: As used here, the term "adapter" means a SCSI controller such as the Western Digital W93 chip, which attaches a unique chain of SCSI devices. In this sense, a SCSI adapter and a SCSI bus are the same. "Adapter number" is used instead of "bus number."

SCSI Hardware Support

The Silicon Graphics computer systems supported by IRIX 6.2 can attach multiple SCSI adapters, as follows:

- The Indy workstation has at least one SCSI adapter on its motherboard, and can have up to two additional adapters on a GIO option board.
- The Indigo^2 series supports two SCSI adapters on the motherboard.
- The Challenge S system has two SCSI adapters on the motherboard, and can have one or two additional on each of one or two additional GIO option boards, for a maximum of sixadapters.

- The Challenge M system supports one SCSI adapter on the CPU board and can have up to two additional adapters on a GIO option board.
- The POWERchannel (IO3) boards used in the Crimson line support two SCSI adapters per board.
- The Power Channel-2™ (IO4) boards used in the Challenge and Onyx series support two SCSI adapters, plus many as six additional SCSI adapters on mezzanine cards, for a maximum of eight adapters per IO4. In addition, VME-SCSI adapters (*Jag* units) can be installed on the VME bus in these systems.

In all systems, DMA mapping hardware allows a SCSI adapter to treat discontiguous memory locations as if they were a contiguous buffer, providing scatter/gather support.

IRIX Kernel SCSI Support

The IRIX kernel contains two levels of SCSI support. An inner SCSI driver, the *host adapter driver*, manages all communication with SCSI hardware adapters. The kernel-level SCSI device drivers for particular devices prepare SCSI commands and call on the host adapter driver to execute them. This design centralizes the management of SCSI adapters. Centralization is necessary because the use of the SCSI bus is multiplexed across many devices, while recovery and error-handling need central handling. In addition, use of the host adapter driver makes it simpler to write a SCSI device driver.

Host Adapter Drivers

Different host adapter drivers are loaded, depending on the hardware in the system. Some examples of host adapter drivers are *wd93*, *wd95*, and *jag*.

The host adapter drivers support all levels of the SCSI standard: SCSI-1, the Common Command Set (CCS, superceded by SCSI-2), and SCSI-2. Not all optional features of the standard are supported. Different systems support different feature combinations (such as synchronous, fast, and wide SCSI), depending on the available hardware.

The host adapter drivers handle the low-level communication over the SCSI interface, such as programming the SCSI interface chip or board, negotiating synchronous or wide mode, and handling disconnect/reconnect.

A host adapter driver is not, strictly speaking, a proper device driver because it does not support all the entry points documented in Chapter 8, "Structure of a Kernel-Level Driver." You can think of it as a specialized library module for SCSI-bus management or as a device driver, whichever you prefer. The software interface to the host adapter driver is documented under "Host Adapter Facilities" on page 354.

Caution: Connect/disconnect strategy is enabled on any SCSI bus by default (the option is controlled by a constant defined in the host adapter driver descriptive file in */var/sysgen/master.d*). When disconnect is enabled on a bus, and a device on that bus refuses to disconnect, it can cause timeouts on other devices.

SCSI Device Drivers

SCSI device drivers handle high-level device management, primarily by setting up SCSI commands for the host adapter driver to execute, and by interpreting returned sense data. Examples of device drivers are *dksc*, *tpsc*, and *smfd*.

Host Adapter Facilities

The principal difference between a SCSI driver and other kernel-level drivers is that, while other kinds of drivers are expected to control devices directly using PIO and DMA, a SCSI driver operates its devices indirectly, by making function calls to the host adapter driver. This section documents the functional interface to the host adapter driver.

Purpose of the Host Adapter Driver

The reason that IRIX uses host adapter drivers is that the SCSI bus is shared among multiple devices of different types, each type controlled by a different driver. A disk, a tape, a CDROM, and a scanner could all be cabled from the same SCSI adapter. Each device has a different driver, but each driver needs to use the adapter, a single chip-set, to communicate with its device.

If IRIX allowed multiple drivers to operate the host adapter, there would be confusion and errors from the conflicting uses. IRIX puts the management of each host adapter under the control of a host adapter driver, whose job is to issue commands on its bus and report the results. The host adapter is tailored to the hardware of the particular host adapter and to the architecture of the host system.

The interface to the host adapter driver is the same no matter what type of hardware the adapter uses. This insulates the individual device drivers from details of the adapter hardware.

The driver for each type of device is responsible for preparing the SCSI command bytes for its device, for passing the command requests to the correct host adapter driver, and for interpreting sense and status data as it comes back.

Host Adapter Concepts

IRIX 6.2 permits a total of 10 unique host adapter drivers—five supplied by Silicon Graphics and up to five from other vendors. Each host adapter is customized to manage one type of adapter hardware. Each adapter driver has an adapter type number that is declared in *sys/scsi.h*. The constant names, are listed in Table 15-1.

Table 15-1 Host Adapter Driver Classes

Driver Constant	**Driver Description**
SCSIDRIVER_NULL	No driver exists; invalid adapter number or nonexistent adapter.
SCSIDRIVER_WD93	The *wd93* driver, for adapters based on the Western Digital WD93 chip set.
SCSIDRIVER_JAG	The *jag* driver, for adapters based on the VME-SCSI bridge used in the Challenge and Onyx systems.
SCSIDRIVER_WD95	The *wd95* driver, for adapters based on the Western Digital WD95 chip set.
SCSIDRIVER_SCIP	The *scip* driver, for adapters based on the augmented WD95 chip set used in Challenge and Onyx systems.
SCSIDRIVER_QL	The *ql* driver, for adapters based on the QLogic chip set.
SCSIDRIVER_3RD_PARTY_START	First number available for OEM host adapter drivers.
SCSIDRIVER_3RD_PARTY_END	Last number available for OEM host adapter drivers.

Caution: The constant names listed in Table 15-1 compile to different values in different hardware systems. For this reason, you should avoid using these names in your driver; if you use one, your driver object file has to be recompiled for each CPU type.

The *lboot* command loads a host adapter driver for each unique type of adapter in the system. *lboot* is directed by VECTOR statements in the */var/sysgen/system/irix.sm* file (see "Configuring a Kernel" on page 236).

You can examine VECTOR lines in */var/sysgen/system/irix.sm* to see how many adapters your system has, and which of the host adapter drivers listed in Table 15-1 is loaded for each one.

The adapter number, the target number, and the logical unit number are important parameters to all the functions of the host adapter driver.

Target Numbers

The purpose of a host adapter driver is to carry communications between a device driver and a *target*. A target is a device on the SCSI chain that responds to SCSI commands. A target can be a single device, or it can be a controller that in turn manages other devices.

A target is identified by a number between 0 and 15. Normally this number is configured into the device with switches or jumpers. The SCSI controller, usually target number 0 but 7 for the jag controller, cannot be used as a target.

The target number must be conveyed to the device driver somehow. The target numbers of Silicon Graphics disk and tape devices are passed in the device minor number.

Not all adapters support the range of 0-15 targets. The Jaguar VME-SCSI unit contains two independent adapters, each supporting target numbers 0-7.

Logical Unit Numbers (LUNs)

When the target is a controller, it manages one or more sub-devices, each one a *logical unit* of that target. The logical unit being addressed is identified by a logical unit number (LUN). When the target is a single device, its LUN is 0.

Overview of Host Adapter Functions

IRIX 6.2 permits a total of 10 unique host adapter drivers, but each of the ten must provide the same functional interface, which is based on simple concepts. The interface to host adapter drivers is declared in *sys/scsi.h*. Each adapter driver must provide the functions listed in Table 15-2.

Table 15-2 Host Adapter Function Summary

Function	Header Files	Can Sleep?	Purpose
scsi_info(D3)	scsi.h	Y	Issue the SCSI Inquiry command and return the results.
scsi_alloc(D3)	scsi.h	Y	Open a connection between a driver and a target device.
scsi_free(D3)	scsi.h	Y	Release connection to target device.
scsi_command(D3)	scsi.h	Y?	Transmit a SCSI command on the bus and return results.
scsi_abort()	scsi.h	Y?	Transmit a SCSI ABORT command (see caution).
scsi_reset()	scsi.h	Y?	Reset the SCSI adapter or bus (see caution).

The normal sequence of operations is as follows:

1. In the *pfx***open()** entry point (or, rarely, in an initialization entry point), the device driver calls **scsi_info()** to test the device characteristics. The results verify that the target device exists and is of the expected type.
2. In the *pfx***open()** entry point, the device driver calls **scsi_alloc()** to set up communications with the target device. This allocates resources in the host adapter driver.
3. In the *pfx***strategy()** or *pfx***ioctl()** entry points, the device driver constructs SCSI command strings and calls **scsi_command()** to have them executed.
4. In the *pfx***close()** entry point, the device driver calls **scsi_free()** to release any held resources related to this device.

Caution: The program interface to the **scsi_abort()** and **scsi_reset()** functions is likely to change in a near-future release. If you need these powerful functions, it is a good idea to code the function calls as macros that can easily be changed.

How the Host Adapter Functions Are Found

A SCSI device driver can be asked to manage devices on different adapters. But different adapters can use different hardware, and be managed by different host adapter drivers. When opening one device, the device driver might need to call **scsi_alloc()** as provided by the *wd93* driver. When opening a different device, the driver might need the **scsi_alloc()** function from the *jag* driver. How can a driver locate the correct host adapter function for a given device?

The answer is provided by a set of function vector tables that are indexed by adapter number, and that yield the address of the appropriate function for that adapter.

Using the Function Vector Tables

The function vector tables are maintained by the *scsi* driver module and filled in by each host adapter driver as it is initialized. The four vector tables are declared in *sys/scsi.h*. The declaration of table *scsi_command* is as follows:

```
extern void (*scsi_command[])(struct scsi_request *req);
```

This declaration states that *scsi_command* is an array of pointers to functions. Each function has the prototype

```
void function(struct scsi_request *req);
```

The four tables *scsi_command*, *scsi_alloc*, *scsi_free*, and *scsi_info* are similar. Each is an array of pointers to functions. Each array is indexed by the adapter type number. If *iAdapT* is an integer variable containing the adapter type number for a device, the following statements are valid calls to the four host adapter functions (the function arguments are examined in detail in the following topics):

```
#include <sys/scsi.h>
pTargInfo = (*scsi_info[iAdapT])(iAdap,iTarg,iLun);
iAllocRet = (*scsi_alloc[iAdapT])(iAdap,iTarg,iLun,0,NULL);
(void) (*scsi_command[iAdapT])(&request);
(void) (*scsi_free[iAdapT])(iAdap,iTarg,iLun,NULL);
```

Each statement is a function call, but in each case, the name of the function is replaced by an expression that indexes the appropriate table.

Learning the Adapter Type Number

Clearly, a SCSI driver needs to know the adapter type number for each device that it manages. Otherwise it cannot call the host adapter functions to manage that device.

The adapter type number for each adapter in the system is stored in an array maintained by the *scsi* driver. The array is declared as follows in *sys/scsi.h*:

```
extern u_char scsi_driver_table[];
```

When indexed by the number of the adapter in use, this table returns the adapter type number of the host adapter driver for that adapter.

Learning the Adapter Number

Now all that remains is for the device driver to learn the adapter number with each device that it manages. There are two simple ways to do this.

One method is to get the number in the *edt_t* structure. When a device is configured using a VECTOR line, the VECTOR should contain an *adapter=n* parameter. This number is stored in the *e_adap* field of the *edt_t* structure that is passed to the *pfx***edtinit()** entry point. Code to retrieve it in a hypothetical driver is shown in Example 15-1.

Example 15-1 Storing the Adapter Type Number in pfxedtinit()

```
#include <sys/scsi.h>
typedef struct devVital_s {
   uchar devAdapNum;
   uchar devAdapType;
...} devVital_t;
void hypo_edtinit(edt_t *edt)
{
   devVital_t *pVitals;
   ...
   pVitals->devAdapNum = edt->e_adap;
   pVitals->devAdapType = scsi_driver_table[edt->e_adap];
...
}
```

A second method is to get it from the device minor number. For all Silicon Graphics disk and tape devices, the adapter number is encoded into both the visible name and the minor number of the device special file. You can use the bits of the minor number of any device in a similar way (see “Minor Device Number” on page 35).

Under the second plan, **geteminor()** is used to extract the minor number is extracted from the *dev_t* value passed to each entry point (see "Device Number Functions" on page 181). The adapter number is calculated by shifting and masking the minor number. Hypothetical example code is shown in Example 15-2. The code of Example 15-2 can be extended to macros for the logical unit and control unit in obvious ways.

Example 15-2 Extracting an Adapter Number From a Minor Device Number

```
/* Hypothetical minor bits: 00 aaaaaaaa ccccuuuu */
#define MINOR_ADAP_SHIFT 8
#define MINOR_ADAP_MASK 0x00ff
#define MINOR_ADAP(devt) (MINOR_ADAP_MASK & \
                  (geteminor(devt) >> MINOR_ADAP_SHIFT))
```

When the adapter number is known, the expression to call a host adapter function can be converted to a macro as well, possibly making the code more readable. The macro in Example 15-3 encapsulates a call to **scsi_alloc()**. This code takes advantage of the fact that the adapter number is an argument to the function in any case.

Example 15-3 Macro to Encapsulate a Call to scsi_alloc()

```
#define SCSI_ALLOC(adap,targ,lun,opt,func) \
   (*scsi_alloc[scsi_driver_table[adap]]) \
   (adap,targ,lun,opt,func)
```

It could be argued that the double indexing in Example 15-3 imposes needless overhead. An approach with minimum overhead is to reserve space in the device-information structure for four function addresses, and to store the addresses of the host adapter functions with the other unique device information when the device is initialized.

Using scsi_info()

Before a SCSI driver tries to access a device, it must call the host adapter **scsi_info()** function. This function issues an Inquiry command to the specified adapter, target, and logical unit. If the Inquiry is not successful—or if the adapter, target, or LUN is invalid—the return value is NULL. Otherwise, the return value is a pointer to a *scsi_target_info* structure.

The SCSI driver can learn the following things from a call to **scsi_info()**:

- If the return is NULL, there is a serious problem with the device or the information about it. Write a descriptive log message with **cmn_err()** and return ENODEV.
- The *si_inq* field points to the Inquiry bytes returned by the device. Examine them for device-dependent information.
- Note the value in *si_maxq* for the largest number of pending SCSI commands that can be queued to this host adapter driver.
- Test the bits in *si_ha_status* for information about the capabilities and error status of the host adapter itself. The possible bits are declared in *sys/scsi.h*. For example, SRH_NOADAPSYNC indicates that the specified target, or possibly the host adapter itself, does not support synchronous transfer. Not all bits are supported by all host adapter drivers.

You can also call **scsi_info()** at other times; some of the returned information can be useful in error recovery. However, be aware that **scsi_info()** for some host adapters is slow, and can use serialized access to hardware.

Using scsi_alloc()

Depending on its particular design, the host adapter driver may need to allocate memory for data structures, DMA maps, or buffers for sense and inquiry data, before it is ready to execute commands to a particular target. The call to **scsi_alloc()** gives the host adapter driver the opportunity to prepare in these ways.

Because the host adapter driver may allocate virtual memory, it may sleep. Some host adapter drivers allocate all the resources they need on the first call to **scsi_alloc()** and do little or nothing on subsequent calls. However, you cannot predict whether your driver will make the first call or not.

A SCSI device driver will typically call the **scsi_alloc()** function from the *pfx***open()** entry point. However, if the driver needs to issue commands to the device at initialization time, it would call **scsi_alloc()**, use **scsi_command()**, and call **scsi_free()**, all within the *pfx***init()** or *pfx***edtinit()** entry point.

A call to **scsi_alloc()** specifies these parameters:

adap, *targ*, *lun*	Numbers that identify the device on the bus.
option	An integer comprising two parameters, a flag and a count.
callback	Address of a function to be called whenever sense data is gotten from the device.

The option parameter may include the SCSIALLOC_EXCLUSIVE flag to request exclusive use of the target. For example, a tape device driver would typically use exclusive access, while a disk device driver would not. If another driver has allocated a path to the same device, **scsi_alloc()** return EBUSY.

The option parameter may include SCSIALLOC_NOSYNC to specify that this device should not, or cannot, use synchronous transfer mode. That setting can be overridden for single commands by a flag to **scsi_command()** (see Table 15-4 on page 364).

The option parameter can also include a small integer value indicating the number of SCSI commands the driver would like to start before any have completed. The call to **scsi_info()** returns the maximum number of commands that can be queued in this way; a larger number is ignored.

The callback function address can be specified as NULL. The specified callback function is called only when sense data is gotten from the allocated device. Only one driver that allocates a path to a device can specify a callback function. If the path is not held exclusively, any other drivers must specify a null address for their callback functions.

Using scsi_free()

A SCSI driver typically calls **scsi_free()** from the *pfx***close()** entry point. That is the time when the driver knows that no processes have the device open, so the host adapter should be allowed to release any resources it is holding just for this device.

In addition, **scsi_free()** releases the device for use by other drivers, if the driver had allocated it for exclusive use.

Using scsi_command()

A SCSI device driver sends SCSI commands to its device by storing information in a *scsi_request* structure and passing the structure to the **scsi_command()** function for the adapter. The host adapter driver schedules the command on the SCSI bus that it manages, and returns to the caller. When the command completes, a callback function is invoked.

Tip: When debugging a driver using a debugging kernel (see "Preparing the System for Debugging" on page 243), you can display the contents of a *scsi_request* structure using *symmon* or *idbg* (see "Commands to Display I/O Status" on page 267).

Input to scsi_command()

The device driver prepares the *scsi_request* fields shown in Table 15-3.

Table 15-3 Input Fields of the scsi_request Structure

Field Name	Contents
sr_ctlr	The adapter number.
sr_target	The target number.
sr_lun	The logical unit number.
sr_tag	If this target supports the SCSI-2 tagged-queue feature, and this command is directed to a queue, this field contains the queue tag message. Constant names for queue messages are in *sys/scsi.h*: SC_TAG_SIMPLE and two others.
sr_command	Address of the bytes of the SCSI command to issue.
sr_cmdlen	The length of the string at **sr_command*. Constants for the common lengths are in *sys/scsi.h*: SC_CLASS0_SZ (6), SC_CLASS1_SZ (10), and SC_CLASS2_SZ (12).
sr_flags	Flags for data direction and DMA mapping, see Table 15-4.
sr_timeout	Number of ticks (HZ units) to wait for a response before timing out. The host adapter driver supplies a minimum value if this field is zero or too small.
sr_buffer	Address of first byte of data. Must be zero when *sr_bp* is supplied and SRF_MAPBP is specified in *sr_flags*.
sr_buflen	Length of data or buffer space.
sr_sense	Address of space for sense data, in case the command ends in a check condition.

Table 15-3 (continued) Input Fields of the scsi_request Structure

Field Name	Contents
sr_senselen	Length of the sense area.
sr_notify	Address of the callback function, called when the command is complete. A callback address is required on all commands.
sr_bp	Address of a *buf_t* object, when the command is called from a block driver's *pfx***strategy()** entry point and buffer mapping is requested in *sr_flags*.
sr_dev	Address of additional information that could be useful in the callback routine **sr_notify*.

The callback function address in *sr_notify* must be specified. (Device drivers for versions of IRIX previous to 5.x may set a NULL in this field; that is no longer permitted.)

The possible flag bits that can be set in *sr_flags* are listed in Table 15-4.

Table 15-4 Values for the sr_flags Field of a scsi_request

Flag Constant	Purpose
SRF_DIR_IN	Data will be received in memory. If this flag is absent, the command sends data from memory to the device.
SRF_FLUSH	The data cache for the buffer area should be flushed (for output) or marked invalid (for input) prior to the command. This flag should be used whenever the buffer is local to the driver, not mapped by a *buf_t* object. It causes no extra overhead in systems that do not require cache flushing.
SRF_MAPUSER	Set this flag when doing I/O based on a *buf_t* and the B_MAPUSER bit is set in *b_flags*.
SRF_MAP	Set this flag when doing I/O based on a *buf_t* and the BP_ISMAPPED macro returns nonzero.
SRF_MAPBP	The *sr_bp* field points to a *buf_t* in which BP_ISMAPPED returns false. The host adapter driver maps in the buffer.
SRF_AEN_ACK	This request is an acknowledgment of an AEN (Asynchronous Event Notification) message from the target. Following an AEN, any command without this flag is rejected with status SC_ATTN.

Table 15-4 (continued) Values for the sr_flags Field of a scsi_request

Flag Constant	Purpose
SRF_NEG_SYNC	Attempt to negotiate synchronous transfer mode for this command. Ignored by some host adapter drivers. Overrides SCSIALLOC_NOSYNC (see "Using scsi_alloc()" on page 361).
SRF_NEG_ASYNC	Attempt to negotiate asynchronous mode for this command. Ignored unless the device is currently using synchronous mode.

When none of the three flag values beginning SR_MAP are supplied, the sr_buffer address must be a physical memory address. The SR_MAPUSER and SR_MAPBP flags are normally used when the command is issued from a *pfx***strategy()** entry point in order to read or write a buffer controlled from a *buf_t* object.

Command Execution

The host adapter driver validates the contents of the *scsi_request* structure. If the contents are valid, it queues the command for transmission on the adapter and returns. If they are invalid, it sets a status flag (see Table 15-6), calls the *sr_notify* function, and returns.

In any event, the *sr_notify* function is called when the command is complete. This function can be called from the host adapter interrupt handler, so it must assume that it is called in interrupt state.

The device driver should wait for the notify function to be called. The usual way is to share a semaphore (see "Semaphores" on page 222), as follows:

- Prior to calling **scsi_command()**, initialize the semaphore to 0 (the semaphore is being used to wait for an event).
- Immediately after the call to **scsi_command()**, call **psema()** for the semaphore.
- In the notify function, call **vsema()** for the function.

If the request is valid, the device driver will sleep in the **psema()** call until the command completes. If the request is invalid, the semaphore will already have been posted when **psema()** is called.

When the device driver holds an exclusive lock prior to issuing the command, a synchronization variable provides an appropriate way to wait for command completion (see "Using Synchronization Variables" on page 220).

Values Returned in a scsi_request Structure

The host adapter driver sets the results of the request in the *scsi_request* structure. The sr_notify function is the first to inspect the values summarized in Table 15-5.

Table 15-5 Values Returned From a SCSI Command

Field Name	Purpose
sr_status	Software status flags, see Table 15-6.
sr_scsi_status	SCSI status byte, see Table 15-7.
sr_ha_flags	Host adapter status flags, see Table 15-8.
sr_sensegotten	When no sense command was issued, 0. When a sense command was issued following an error, the number of bytes of sense data received. When an error occurred during a sense command, -1
sr_resid	The difference between *sr_buflen* and the number of bytes actually transferred.

The *sr_status* field should be tested first. It contains an integer value; the possible values are summarized in Table 15-6.

Table 15-6 Software Status Values From a SCSI Request

Constant Name	Meaning
SC_GOOD	The request was valid and the command was executed. The command might still have failed; see *sr_scsi_status*.
SC_TIMEOUT	The device did not respond to selection within 250 milliseconds.
SC_HARDERR	A hardware error occurred; inspect *sr_senselen* to see how much sense data was received, if any.
SC_PARITY	SCSI bus parity error detected.
SC_MEMERR	System memory parity or ECC error detected.
SC_CMDTIME	The device responded to selection but the command did not complete before *sr_timeout* expired.
SC_ALIGN	The buffer address was not aligned as required by the adapter hardware. Most Silicon Graphics adapters require word (4-byte) alignment.

Table 15-6 (continued) Software Status Values From a SCSI Request

Constant Name	Meaning
SC_ATTN	Either a unit attention was received, or this command follows an AEN and did not contain the SR_AEN_ACK flag (see Table 15-4).
SC_REQUEST	An error was detected in the input values; the command was not attempted. The error could be that **scsi_alloc()** has not been called; or it could be due to missing or incorrect values.

One or more bits are set in the *sc_scsi_status* field. This field represents the status following the requested command, when the requested command executes correctly. When the requested command ends with Check Condition status, a sense command is issued and the SCSI status following the sense is placed in *sc_scsi_status*. In other words, the true indication of successful execution of the requested command is a zero in *sr_sensegotten*, because this indicates that no sense command was attempted.

Possible values of *sc_scsi_status* are summarized in Table 15-7.

Table 15-7 SCSI Status Bytes

Constant Name	Meaning
ST_GOOD	The target has successfully completed the SCSI command. If a check condition was returned, a sense command was issued. The *sr_sensegotten* field is nonzero when this was the case.
ST_CHECK	This bit is only set for the special case when a check condition occurred on a sense command following a check condition on the requested command. The *sr_sensegotten* field contains -1.
ST_COND_MET	Search condition was met.
ST_BUSY	The target is busy. The driver will normally delay and then request the command again.
ST_INT_GOOD	This status is reported for every command in a series of linked commands. Linked commands are not supported by Silicon Graphics host adapters.
ST_RES_CONF	A conflict with a reserved logical unit or reserved extent.

One or more bits can be set in *sr_ha_flags* to document a host adapter state or problem. These flags are summarized in Table 15-8.

Table 15-8 Host Adapter Status After a SCSI Request

Constant Name	Meaning
SRH_CANTSYNC	Unable to negotiate synchronous mode.
SRH_SYNCXFR	Synchronous mode was used. If not set, asynchronous mode was used.
SRH_TRIEDSYNC	Synchronous mode negotiation was attempted; see the SHR_CANTSYNC bit for the result.
SRH_BADSYNC	When SRH_CANTSYNC is set, this bit indicates that the negotiation failed because the device cannot negotiate.
SRH_NOADAPSYNC	When SRH_CANTSYNC is set, this bit indicates that the host adapter does not support synchronous negotiation, or that the system has been configured to not use synchronous mode for this device.
SRH_WIDE	This adapter supports Wide mode.
SRH_DISC	This adapter supports Disconnect mode and is configured to use it.
SRH_TAGQ	This adapter supports tagged queueing and is configured to use it.
SRH_MAPUSER	This host adapter driver can map user addresses.

Using scsi_abort()

The purpose of the **scsi_abort()** function is to issue a SCSI ABORT command to a specified target and logical unit. The prototype of the function is:

```
int (*scsi_abort[adapter-type])(struct scsi_request *req);
```

The only fields of the scsi_request that are input to this function are those that identify the device: *sr_ctlr*, *sr_target*, and *sr_lun*. The ABORT command is issued on the bus as soon as possible but there could be a delay if the bus is busy. Status is returned in *sr_status*. The function returns a nonzero value when the ABORT command is issued successfully, and a zero when the ABORT command fails (which probably indicates a serious bus problem).

Using scsi_reset()

The purpose of **scsi_reset()** is to reset the adapter hardware and possibly the attached bus, for example by asserting the reset line on the bus for at least 25 microseconds. The prototype of the function is

```
int (*scsi_reset[adapter-type])(uchar adap);
```

The adapter number is reset and a nonzero value is returned. If the host adapter driver does not support this function, or if it is unable to reset the hardware, it returns 0.

Designing a SCSI Driver

A kernel-level SCSI device driver has the driver architecture described in Chapter 8, “Structure of a Kernel-Level Driver,” and it uses the basic system services documented in Chapter 9, “Device Driver/Kernel Interface.” You prepare a SCSI driver and configure it into the kernel as described in Chapter 10, “Building and Installing a Driver.”

However, a SCSI driver uses additional services, including those of the host adapter driver, and its configuration is slightly different from other drivers.

SCSI Driver Initialization

A SCSI driver can be included in the kernel through a VECTOR, INCLUDE, or USE line in the system file in */var/sysgen/system* (see “Configuring a Kernel” on page 236). When included through a VECTOR line, the *pfx***edtinit()** entry point is called for each VECTOR line given. The VECTOR can describe a logical unit or a control unit, according to your design choice. However, a VECTOR line for a high-level SCSI driver can not include a *probe* or *exprobe* operand, because the hardware is owned by the host adapter driver, not by the SCSI device driver.

When included through a USE line, a SCSI driver is initialized at its *pfx***init()** entry point. In this case, the driver must obtain the adapter number by some other means. (See “Initialization Entry Points” on page 143.)

Opening a SCSI Device

When the *pfx***open()** entry point is called, the SCSI driver uses the appropriate **scsi_info()** function to verify the device and get hardware dependent Inquiry data from it. If the device is operational, the driver calls **scsi_alloc()** to open a communications path to it.

Accessing a SCSI Device

In general, it is simplest to put all access to a device within a *pfx***strategy()** entry point, even in a character device driver. When the *pfx***read()**, *pfx***write()**, or *pfx***ioctl()** entry point needs to read or write data, it can prepare a *uio_t* to describe the data, and call **uiophysio()** to direct the operation through the single *pfx***strategy()** entry point.

The notify routine passed in the *sr_notify* field plays the same role as the *pfx***intr()** entry point in other device drivers. It is called asynchronously, when the SCSI command completes. It may not call a kernel function that can sleep. However, it does not have to be named *pfx***intr()**, and a SCSI driver does not have to provide a *pfx***intr()** entry point.

Configuring a SCSI Driver

A SCSI driver can be either a block or a character driver, or it can support both interfaces. When preparing the descriptive file for */var/sysgen/master.d*, you must use the *s* flag, specifying a software-only driver, and list *scsi* as a dependency, in the descriptive line in */var/sysgen/master.d*. See “Describing the Driver in /var/sysgen/master.d” on page 233.

Example SCSI Device Driver

The following example shows how a driver can communicate with a direct access SCSI device, such as a disk. This driver is simplified and does not do as much error checking as a real driver would do. Also, this example uses a single, global SCSI request structure that does not work in real drivers, since multiple reads or writes would overwrite a command in progress.

Tip: A more complete sample SCSI driver is available on the Developer’s Toolbox CD. See information about the Developer Program under “Developer Program” on page xxxvii.

```
#include "sys/param.h"
#include "sys/types.h"
#include "sys/user.h"
#include "sys/buf.h"
#include "sys/errno.h"
#include "sys/cmn_err.h"
#include "sys/cred.h"
#include "sys/ddi.h"
#include "sys/systm.h"
#include "sys/scsi.h"

int sdk_devflag = 0; /* not old, not _MP either */

#define ADAPT    0     /* SCSI host adapter */
#define TARGET   7     /* the disk will have target ID #7 */
#define LU       0     /* and logical unit  #0 */
#define TIMEOUT (30*HZ)/* wait 30 secs for SCSI device to
                          respond */
#define DIRECTACCESS 0 /* First byte of inqry cmnd */

unchar scsi_read[]    = {0x28, 0, 0, 0, 0, 0, 0, 0, 0, 0};
unchar scsi_write[]   = {0x2a, 0, 0, 0, 0, 0, 0, 0, 0, 0};
int    sdk_inuse = 0;
int    sdk_driver;
struct scsi_target_info *sdk_info;
struct scsi_request sdk_req;
u_char sdk_sensebuf[SCSI_SENSE_LEN];  /* SCSI_SENSE_LEN
                                         from scsi.h */
/* forward definitions*/
int sdk_strategy(struct buf *bp);
void sdk_notify(struct scsi_request *req);
/*
 * sdk_open - Open the SCSI device exclusively.
 *
 * Issue a SCSI inquiry command upon device and ensure
 * it is a direct access device.
 */
int
sdk_open(dev_t *devp, int flag, int otyp, cred_t *crp)
{
   if (sdk_inuse)
      return EBUSY;
   /* Get driver number */
   sdk_driver = scsi_driver_table[ADAPT];
   /*
```

```
    * Call through scsi_info to get inquiry data and to
    * find out if a device is at the address we want.
    */
   sdk_info = (*scsi_info[sdk_driver])(ADAPT, TARGET, LU);
   if (sdk_info == NULL)
      return ENODEV;
   /*
    * Is it a direct access device?  We could check the
    * entire inquiry buffer to ensure it is actually the
    * correct device.
    */
   if (sdk_info->si_inq[0] != DIRECTACCESS)
      return ENXIO;
   /*
    * It's a direct access device (disk drive).  Initialize
    * the connection to the host adapter driver.
    */
   if ((*scsi_alloc[sdk_driver])
      (ADAPT, TARGET, LU, 1, NULL) == 0)
      return EBUSY;
   /*
    * We have successfully allocated a connection between
    * sdk and the host adapter driver.  Initialize the
    * scsi_request structure, and mark the driver as being
    * in use.
    */
   sdk_inuse = 1;
   bzero(&sdk_req, sizeof(sdk_req));
   sdk_req.sr_ctlr = ADAPT;
   sdk_req.sr_target = TARGET;
   sdk_req.sr_lun = LU;
   sdk_req.sr_timeout = TIMEOUT;
   sdk_req.sr_sense = sdk_sensebuf;
   sdk_req.sr_senselen = sizeof(sdk_sensebuf);
   sdk_req.sr_notify = sdk_notify;

   return 0;
}
/* sdk_close - close the device and free the subchannel. */
int
sdk_close(dev_t dev, int flag, int otyp, cred_t *crp)
{
   (*scsi_free[sdk_driver])(ADAPT, TARGET, LU, NULL);
   sdk_inuse = 0;
   return 0;
```

```
}
/*
 * sdk_read - read from the SCSI device, ensuring an even
 * block count and a word-aligned address.
 */
sdk_read(dev_t dev, uio_t *uiop, cred_t *crp)

/*
 * sdk_write - write to the SCSI device, ensuring an even
 * block count and a word-aligned address.
 */
sdk_write(dev_t dev, uio_t *uiop, cred_t *crp)

/*
 * sdk_strategy - do the dirty work of the I/O.
 * Use either the SCSI read or write command as
 * appropriate.  Modify the block number and block counts
 * within the command buffer. Simply return here;
 * physio( ) will wait for an iodone( ).
 */
int
sdk_strategy(struct buf *bp)
{
   int blkno, blkcount;
   /* Prime the subchannel communication block. */
   blkno = bp->b_blkno;
   blkcount = BTOBB(bp->b_bcount);
   sdk_req.sr_command = bp->b_flags & B_READ ?
                     scsi_read : scsi_write;
   sdk_req.sr_command[2] = (char)(blkno>>24);
   sdk_req.sr_command[3] = (char)(blkno>>16);
   sdk_req.sr_command[4] = (char)(blkno>>8);
   sdk_req.sr_command[5] = (char) blkno;
   sdk_req.sr_command[7] = (char)(blkcount>>8);
   sdk_req.sr_command[8] = (char) blkcount;

   sdk_req.sr_cmdlen = SC_CLASS1_SZ;
   sdk_req.sr_flags = bp->b_flags & B_READ ? SRF_DIR_IN : 0;
   if (BP_ISMAPPED(bp)) {
      sdk_req.sr_buffer = bp->b_dmaaddr;
      sdk_req.sr_buflen = bp->b_bcount;
      sdk_req.sr_flags |= SRF_MAP;
   }
   else {
      sdk_req.sr_buffer = NULL;
```

```
        sdk_req.sr_buflen = bp->b_bcount;
        sdk_req.sr_flags = SRF_MAPBP;
    }
    sdk_req.sr_bp = bp;   /* required for SRF_MAPBP, but a
                           * convenience in all cases */
    /* Perform the SCSI operation. */
    (*scsi_command[sdk_driver])(&sdk_req);
}

/*
 * sdk_notify - SCSI command completion notification routine
 *
 * Simply check for errors and wake up physio( ) with
 * an iodone( ) on the buffer.
 * Note that a more robust driver would be more thorough
 * about error handling by retrying errors, giving more
 * information about error types, etc.
 */
void
sdk_notify(struct scsi_request *req)
{
    register struct buf *bp = req->sr_bp;
    if ((req->sr_status != SC_GOOD) ||
        (req->sr_scsi_status != ST_GOOD) ||
        (req->sr_sensegotten < 0))
    {
        cmn_err(CE_NOTE,
            "sdk: Error: driver stat 0x%x, scsi stat 0x%x"
            " sensegotten %d\n", req->sr_status,
            req->sr_scsi_status, req->sr_sensegotten);
        bioerror(bp, EIO);
            }
    else if (req->sr_sensegotten > 0) {
        cmn_err(CE_NOTE, "sdk: Error: sensekey 0x%x\n",
            sdk_sensebuf[2] & 0x0F);
        bioerror(bp, EIO);
    }
    bp->b_resid = req->sr_resid;
    biodone(bp);
}
```

Designing a Host Adapter Driver

IRIX 6.2 provides the ability to load and install third-party host adapter drivers. This section documents the special features of this type of driver.

Overview of Host Adapter Driver Architecture

A host adapter driver is a low-level driver for a SCSI bus adapter. A host adapter driver is similar to other device drivers described in this book in many ways:

- Like other device drivers, it uses the kernel facilities described in Chapter 9, "Device Driver/Kernel Interface."
- It is compiled and linked like other drivers (see Chapter 10, "Building and Installing a Driver").
- It is configured to the system using files in */var/sysgen/master.d*, and loaded by *lboot*. A host adapter driver should not be loadable; if it is loadable, it should not unload.
- Like other drivers, it can have entry points *pfx***start()** or *pfx***edtinit()** for initialization, *pfx***intr()** for interrupt handling, and *pfx***halt()** for shutdown.

Unlike other drivers, a host adapter driver does not provide any entry points for serving the needs of system functions, such as *pfx***read()**, *pfx***poll()**, or *pfx***strategy()**. Instead, it supplies the entry points used by SCSI device drivers.

Host Adapter Initialization

In its initialization, the host adapter driver does three things:

- initializes the adapter hardware it supports
- acquires an adapter type number
- stores pointers to its functions in the function pointer arrays

Initializing the Hardware

If it is called at its *pfx***edtinit()** entry point, the host adapter driver receives adapter hardware information in an *edt_t* structure. Integration of the driver in this way, using a VECTOR line, is preferred. It removes the need to hard-code device addresses; and it allows the use of an *exprobe* operand to load the driver only when its adapter hardware is present.

If it is not loaded by a VECTOR line, the driver must be loaded with a USE line in the system database (see "Configuring a Kernel" on page 236) and it must find out the address of its adapter hardware by other means.

The driver may also include a *pfx***start()** entry point for general initialization, including the two steps of acquiring a type number and setting up its entry point addresses.

Acquiring a Type Number

Every host adapter driver must have a unique adapter type number. The type numbers for Silicon Graphics drivers are declared in *sys/scsi.h*. An OEM driver acquires a number dynamically, by calling the kernel function **get_driver_number()**. The prototype of this function is

```
uchar get_driver_number(void);
```

If successful, the function returns a number between SCSIDRIVER_3RD_PARTY_START and SCSIDRIVER_3RD_PARTY_END, inclusive (see Table 15-1 on page 355). If unsuccessful, it returns -1, and the driver cannot initialize.

Storing Entry Point Addresses

After it has its type number, the driver can store the address of each of its functional entry points in the arrays used by its callers. For example it stores the address of its command execution function in the *scsi_command* array, indexed by its type number.

The driver must support the functions summarized in Table 15-2 on page page 357. For each function there is a corresponding array of function pointers, in which the driver stores the address of its function, indexed by its driver type number.

SCSI Reference Data

This section contains reference material in the following categories:

- "SCSI Error Messages" on page 377 describes the general form of messages written by host adapter drivers into the system log.
- "Adapter Error Codes (Table scsi_adaperrs_tab)" on page 378 lists the possible adapter error codes and their message strings.
- "SCSI Sense Codes (Table scsi_key_msgtab)" on page 379 lists the primary sense codes and the corresponding message strings.
- "Additional Sense Codes (Table scsi_addit_msgtab)" on page 380 lists the possible additional sense codes (ASCs) and their message strings.

SCSI Error Messages

The host adapter drivers such as *wd93*, *wd95*, and *jag* send error messages to the system log using the **cmn_err()** function (see "Producing Diagnostic Displays" on page 249).

These messages almost always contain the adapter number (sometimes called the bus number or controller number). They sometimes contain the number of the target device, and sometimes add the number of the logical unit that was addressed.

Messages from the *wd93* driver specify the adapter number as `Bus=`*n*. The target device is shown as `ID=`*n* and the logical unit as `LUN=`*n*.

Messages from the *wd95* and *jag* drivers contain one, two, or three or more decimal numbers. In all cases, the first number is the adapter number, the second is the target ID, and the third (when present) is the logical unit number.

When error messages list a sense code, refer to "SCSI Sense Codes (Table scsi_key_msgtab)" on page 379 and to "Additional Sense Codes (Table scsi_addit_msgtab)" on page 380.

When the error message reports an error from the adapter itself, refer to "Adapter Error Codes (Table scsi_adaperrs_tab)" on page 378.

SCSI Error Message Tables

The *scsi* module contains a set of error message tables that you can use to generate error messages based on SCSI sense codes and other data. The contents of these tables is documented here for reference, and to assist in decoding messages from SCSI drivers.

Each table is an array of pointers to strings; for example the *scsi_key_msgtab* table is defined beginning as follows:

```
char *scsi_key_msgtab[SC_NUMSENSE] = {
   "No sense",          /* 0x0 */
   "Recovered Error",   /* 0x1 */
...};
```

Each of the tables is declared as extern in *sys/scsi.h*.

Adapter Error Codes (Table scsi_adaperrs_tab)

The table with the external name *scsi_adaperrs_tab* contains message strings to document the adapter error codes that can be returned in the *scsirequest.sr_status* field (see Table 15-6). The *scsi_adaperrs_tab* table contains NUM_ADAP_ERRS entries (9, defined in *sys/scsi.h*). The first entry (index 0x0) contains a pointer to a null string. The other entries are documented in Table 15-9.

Table 15-9 Adapter Error Codes

Adapter Error Code	Constant Name	Message Text
0x1	SC_TIMEOUT	Device does not respond to selection .
0x2	SC_HARDERR	Controller protocol error or SCSI bus reset.
0x3	SC_PARITY	SCSI bus parity error.
0x4	SC_MEMERR	Parity/ECC error in system memory during DMA.
0x5	SC_CMDTIME	Command timed out.
0x6	SC_ALIGN	Buffer not correctly aligned in memory.
0x7	SC_ATTN	Unit attention received on another command causes retry.
0x8	SC_REQUEST	Driver protocol error.

SCSI Sense Codes (Table scsi_key_msgtab)

The table with the external name *scsi_key_msgtab* is indexed by the primary sense code. Its contents are listed in Table 15-10. The table contains SC_NUMADDSENSE entries (16, defined in *sys/scsi.h*), of which the last two should not occur.

Table 15-10 Primary Sense Key Error Table

Sense Key	Message	Most Common Cause
0x0	`No sense`	No error information available.
0x1	`Recovered error`	The device recovered by itself.
0x2	`Device not ready`	No media, or drive not spun up.
0x3	`Media error`	An actual media problem.
0x4	`Device hardware error`	Usually a device hardware error.
0x5	`Illegal request`	Invalid command or data issued.
0x6	`Unit attention`	Device was reset or power-cycled.
0x7	`Data protect error`	Usually device is write-protected.
0x8	`Unexpected blank media`	Tried to read at end of a tape.
0x9	`Vendor unique error`	Varies.
0xA	`Copy aborted`	Copy command aborted by host (not used).
0xB	`Aborted command`	Target device aborted command.
0xC	`Search data successful`	Search data command OK (not used).
0xD	`Volume overflow`	Tried to write past EOT on tape.
0xE	`Reserved (0xE)`	0xE should not be seen.
0xF	`Reserved (0xF)`	0xF should not be seen.

Additional Sense Codes (Table scsi_addit_msgtab)

The table with the external name *scsi_addit_msgtab* is indexed by the Additional Sense Code (ASC) value, when one is present. The table contains SC_NUMADDSENSE entries (0x71, defined in *sys/scsi.h*). Some values have no standard definition; for these, the table contains a NULL value. Therefore you should always test the table value for a valid pointer before using it to format a message.

Table 15-11 lists the contents of this message table. Undefined (NULL) table entries are omitted.

Table 15-11 Additional Sense Code Table

ASC Value	Corresponding Message String
0x01	`No index/sector signal`
0x02	`No seek complete`
0x03	`Write fault`
0x04	`Not ready to perform command`
0x05	`Unit does not respond to selection`
0x06	`No reference position`
0x07	`Multiple drives selected`
0x08	`LUN communication error`
0x09	`Track error`
0x0a	`Error log overflow`
0x0c	`Write error`
0x10	`ID CRC or ECC error`
0x11	`Unrecovered data block read error`
0x12	`No address mark found in ID field`
0x13	`No address mark found in Data field`
0x14	`No record found`
0x15	`Seek position error`

Table 15-11 (continued) Additional Sense Code Table

ASC Value	Corresponding Message String
0x16	`Data sync mark error`
0x17	`Read data recovered with retries`
0x18	`Read data recovered with ECC`
0x19	`Defect list error`
0x1a	`Parameter overrun`
0x1b	`Synchronous transfer error`
0x1c	`Defect list not found`
0x1d	`Compare error`
0x1e	`Recovered ID with ECC`
0x20	`Invalid command code`
0x21	`Illegal logical block address`
0x22	`Illegal function`
0x24	`Illegal field in CDB`
0x25	`Invalid LUN`
0x26	`Invalid field in parameter list`
0x27	`Media write protected`
0x28	`Media change`
0x29	`Device reset`
0x2a	`Log parameters changed`
0x2b	`Copy requires disconnect`
0x2c	`Command sequence error`
0x2d	`Update in place error`
0x2f	`Tagged commands cleared`
0x30	`Incompatible media`

Table 15-11 (continued) Additional Sense Code Table

ASC Value	Corresponding Message String
0x31	`Media format corrupted`
0x32	`No defect spare location available`
0x33[a]	`Media length error`
0x36	`Toner/ink error`
0x37	`Parameter rounded`
0x39	`Saved parameters not supported`
0x3a	`Medium not present`
0x3b	`Forms error`
0x3d	`Invalid ID msg`
0x3e	`Self config in progress`
0x3f	`Device config has changed`
0x40	`RAM failure`
0x41	`Data path diagnostic failure`
0x42	`Power on diagnostic failure`
0x43	`Message reject error`
0x44	`Internal controller error`
0x45	`Select/reselect failed`
0x46	`Soft reset failure`
0x47	`SCSI interface parity error`
0x48	`Initiator detected error`
0x49	`Inappropriate/illegal message`
0x4a	`Command phase error`
0x4b	`Data phase error`
0x4c	`Failed self configuration`

Table 15-11 (continued) Additional Sense Code Table

ASC Value	Corresponding Message String
0x4e	`Overlapped commands attempted`
0x53	`Media load/unload failure`
0x57	`Unable to read table of contents`
0x58	`Generation (optical device) bad`
0x59	`Updated block read (optical device)`
0x5a	`Operator request or state change`
0x5b	`Logging exception`
0x5c	`RPL status change`
0x5d	`Self diagnostics predict unit will fail soon`
0x60	`Lamp failure`
0x61	`Video acquisition error/focus problem`
0x62	`Scan head positioning error`
0x63	`End of user area on track`
0x64	`Illegal mode for this track`
0x70[b]	`Decompression error`

a. Specified as tape only.

b. DAT only; may be in SCSI3.

WD93 States and Phases

Some of the SCSI states and phases that can be detected by the *wd93* host adapter driver are listed in Table 15-12 for reference, in case they appear in a debugging log message. These states and phases are declared in the */usr/include/sys/wd93.h* header file. The comments in the table have been extracted from that file and supplemented with additional information.

"Out" is from the CPU to the SCSI device in these descriptions, and "receive" and "send" are also from the SCSI device point of view, since the target controls all the bus phases except for initial selection.

Table 15-12 SCSI State Error Messages

State Message	Sense Key	Comments
`ST_RESET`	0x00	SCSI chip reset by reset command or power-up.
`ST_SELECT`	0x11	Selection of target complete (after C93SELATN).
`ST_SATOK`	0x16	Select-And-Transfer completed successfully, that is, all phases have completed in a normal manner.
`ST_TR_DATAOUT`	0x18	Transfer command done, target requesting data.
`ST_TR_DATAIN`	0x19	Transfer command done, target sending data.
`ST_TR_STATIN`	0x1b	Target is sending status in.
`ST_TR_MSGIN`	0x1f	Transfer command done, target sending message.
`ST_TRANPAUSE`	0x20	Transfer command has paused with ACK.
`ST_SAVEDP`	0x21	Save Data Pointers message during SAT normal state when device is disconnecting from the bus.
`ST_A_RESELECT`	0x27	Reselected after disconnect (93A).
`ST_UNEXPDISC`	0x41	Device disconnected without sending a disconnect message. This sometimes happens when devices with removable media have had the media removed during a transfer.
`ST_PARITY`	0x43	Command terminated due to parity error on the SCSI bus.

Table 15-12 (continued) SCSI State Error Messages

State Message	Sense Key	Comments
`ST_PARITY_ATN`	0x44	Command terminated due to parity error (ATN is asserted so that host can send a message to device; the transfer is just aborted).
`ST_TIMEOUT`	0x42	Time-out during Select or Reselect, that is, the device never responded to an attempt to select it; normally seen only during hardware inventory probing, but sometimes happens after a SCSI bus reset if device takes a long time to recover from the reset or is powered off.
`ST_INCORR_DATA`	0x47	Incorrect message or status byte.
`ST_UNEX_RDATA`	0x48	Unexpected receive data phase device tried to send more data than the SCSI chip is programmed to expect. This can be OK, as when a high-level request is made to transfer more data than the DMA hardware can map on a single request. In this case, simply reprogram the DMA hardware for the next chunk of data and restart the transfer (but don’t send a new SCSI command to the device). When printed as part of an error message, it can sometimes be caused by a SCSI cabling problem, or (particularly with devscsi user drivers) by a mismatch in the byte count given to the driver and the byte count implied by the SCSI command sent to the device.
`ST_UNEX_SDATA`	0x49	Unexpected send-data phase (same as above, but device is asking for more data).
`ST_UNEX_CMDPH`	0x4a	Unexpected command phase
`ST_UNEX_SSTATUS`	0x4b	Unexpected send status phases occur at the end of SCSI command (that is, byte count remaining is 0); if they happen at other times, the chip interrupts. This can happen when you ask a device for more data than it can give you, and in this case, you just return a short I/O count to the caller. When printed as part of an error message, it usually implies a cabling or termination problem.
`ST_UNEX_RMESGOUT`	0x4e	Unexpected request-message-out phase; usually indicates a SCSI cabling problem.

Table 15-12 (continued) SCSI State Error Messages

State Message	Sense Key	Comments
`ST_UNEX_SMESGIN`	0x4f	Unexpected send-message-in phase. Usually indicates a SCSI cabling problem; also happens when device sends an unsolicited disconnect message when preparing to disconnect from the bus.
`ST_RESELECT`	0x80	WD33C93 has been reselected.
`ST_93A_RESEL`	0x81	Reselected while idle (93A).
`ST_DISCONNECT`	0x85	Disconnect has occurred.
`ST_NEEDCMD`	0x8a	Target is ready for a command.
`ST_REQ_SMESGOUT`	0x8e	REQ signal for send message out.
`ST_REQ_SMESGIN`	0x8f	REQ signal for send message in above 3 usually seen only during sync negotiations.

PART SIX

Network Drivers

Chapter 16: Network Device Drivers
Network device drivers are special in that they interface a device to the *ifnet* interface of the TCP/IP protocol stack.

Chapter 16

Network Device Drivers

A network device driver is a kernel-level driver that connects a communications device to the IRIX TCP/IP protocol stack using the *ifnet* interface established by BSD UNIX. This chapter contains these major topics:

- "Overview of Network Drivers" on page 390 gives an overview of the IRIX networking subsystem and the role of an ifnet driver in it.
- "Network Driver Interfaces" on page 392 summarizes the unique interfaces used by an *ifnet* driver.
- "Example ifnet Driver" on page 401 displays the code of a network driver, omitting all device-specific features.

Note: If your interest is in creating a network application based on sockets, TLI, or streams, this chapter offers little but background information. Refer to the *IRIX Network Programming Guide*, document Number 007-0810-050, for a complete review of all application-level services.

Even if your interest is in creating a kernel-level network driver, you should be familiar with the facilities documented in the *IRIX Network Programming Guide*. This chapter assumes that your are familiar with them.

Overview of Network Drivers

A network driver is a kernel-level driver module that connects a communications device such as an Ethernet board to the IRIX implementation of TCP/IP. An overview of the IRIX networking subsystem is shown in Figure 16-1.

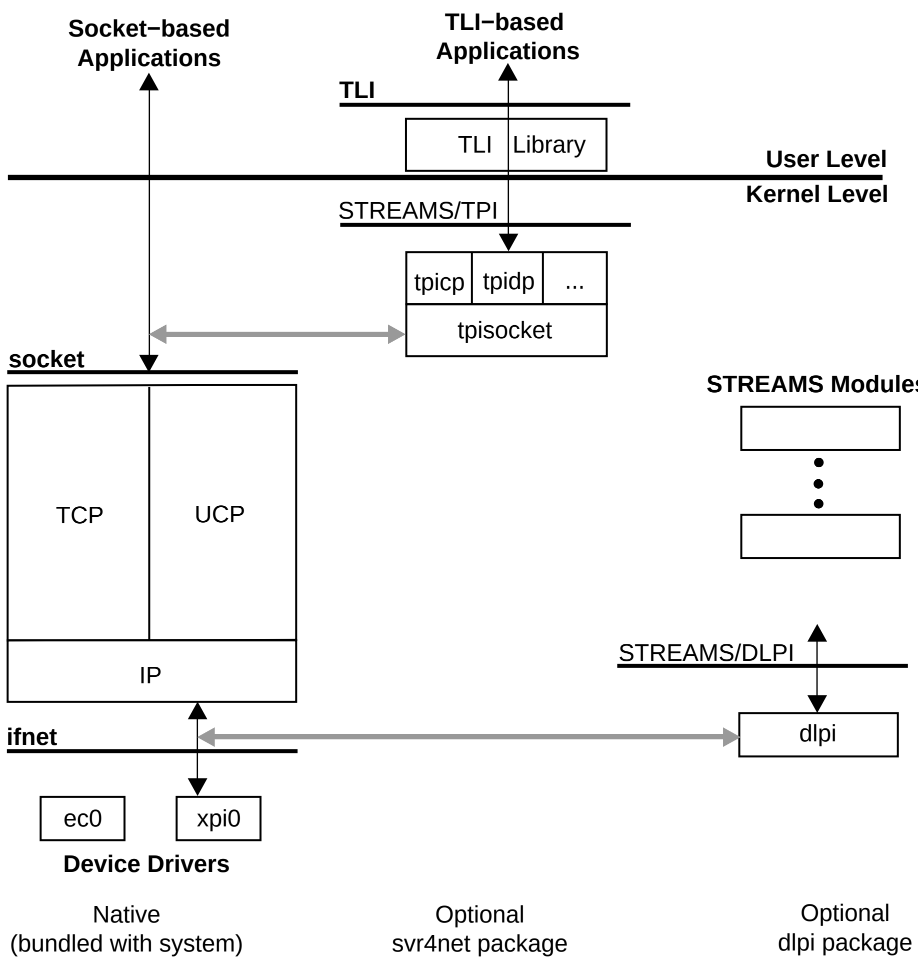

Figure 16-1 Overview of Network Architecture

Application Interfaces

User-level processes access the network in one of three ways:

- using the BSD socket interface (top left of Figure 16-1)
- using the SVR4 TLI interface through compatibility libraries that convert TLI operations into socket operations (top center of Figure 16-1)
- using a STREAMS interface to a STREAMS-based protocol stack (top right of Figure 16-1)

These three interfaces are documented in the *IRIX Network Programming Guide*

The native socket-based TCP/IP protocol code, the socket layer, and a number of *ifnet*-based device drivers are bundled in the basic IRIX system. Socket-based applications such as *rlogin*, *rcp*, NFS client and server, and the socket-based RPC library operate directly over this native networking framework.

Compatibility support is included for applications written to the STREAMS Transport Layer Interface (TLI). *tpisocket* is a kernel library module used by protocol-specific STREAMS pseudo-drivers, such as tpitcp, tpiudp, and so on, providing a TPI interface above the native kernel sockets-based network protocol stack.

A STREAMS pseudo-driver that supports the Data Link Provider Interface (DLPI) for STREAMS-based kernel protocol stacks is delivered in the optional dlpi package.

Protocol Stack Interfaces

A *protocol stack* is the software subsystem that manages data traffic according to the rules of a particular communications protocol. There are two ways in which a protocol stack can be integrated into the IRIX kernel. The TCP/IP stack creates and uses the *ifnet* interface to drivers (bottom left of Figure 16-1) and the socket interface to applications (top left of Figure 16-1).

Alternatively, a stack written to the DLPI architecture can communicate with STREAMS drivers (bottom right of Figure 16-1).

Device Driver Interfaces

A network driver uses the methods and facilities of other kernel-level device drivers, as described in Part III, “Kernel-Level Drivers” of this book. A network driver is compiled and linked like other drivers, configured using the same configuration files, and loaded into the kernel by *lboot* like other drivers.

However, other device drivers support the UNIX filesystem, transferring data in response to calls to their *pfx***read()**, *pfx***write()**, or *pfx***strategy()** entry points. This is not the case with a network driver; it supports protocol stacks, and it transfers data in response to calls from the *ifnet* interface.

Note: If you are working on a network device driver that uses DMA and that can be used in a Challenge or Onyx system, you should read Appendix B, “Challenge DMA with Multiple IO4 Boards.”

Network Driver Interfaces

The IRIX kernel networking design is based on the kernel networking framework in 4.3BSD. If you are familiar with the 4.3BSD kernel networking design, then you are already familiar with the IRIX kernel networking design because they are basically the same.

The IRIX networking design is based on the socket interface: *mbuf* objects are used to exchange messages within the kernel, and device drivers support the TCP/IP internet protocol suite by supporting the *ifnet* interface.

Since the BSD-based networking framework and the implementation of the TCP/IP protocol suite have changed little from previous releases of IRIX, porting your *ifnet* device driver to this release of IRIX should be straightforward.

Note: Although the general kernel facilities documented in Chapter 9, “Device Driver/Kernel Interface” are standardized and stable, this is not the case with network interfaces. *The ifnet and other interfaces summarized in this topic are subject to change without notice*.

Kernel Facilities

A network driver is structured like any kernel-level device driver, much as described in Chapter 8, “Structure of a Kernel-Level Driver,” but with the following similarities and differences:

- A network driver is loaded by *lboot* in response to either a USE or VECTOR line in a file in */var/sysgen/system* (see “Configuring a Nonloadable Driver” on page 232).
- A network driver is initialized by a call to either its *pfx***init()** or *pfx***edtinit()** entry point when it is loaded.
- A network driver does not need to provide any other entry points (see “Entry Point Summary” on page 138).
- A network driver does not need to provide a driver flag constant *pfx***devflag** because a network driver is always assumed to be multiprocessor-aware (see “Driver Flag Constant” on page 140).
- Although a network driver can use the kernel functions for synchronization and locking (see “Waiting and Mutual Exclusion” on page 204), it normally does not because the *ifnet* interface includes special-purpose locking facilities that are more convenient (see “Multiprocessor Considerations” on page 396).

Principal ifnet Header Files

The software interface to network facilities is declared in the following important header files:

net/if.h	Basic ifnet facilities and data structures, including the *ifnet* structure, the basic driver interface object.
net/if_types.h	Constants for interface types, used in decoding address headers.
sys/mbuf.h	The *mbuf* structure with related constants and macros, and declarations of functions to allocate, manipulate, and free *mbuf* objects.
net/netisr.h	Declarations related to software interrupts, including **schednetisr()** to schedule an interrupt, and the IP input queue *ipintrq*.
net/multi.h	Routines defining a generic filter for use by drivers whose devices cannot perfectly filter multicast packets.

net/soioctl.h	Socket **ioctl()** function numbers, some of which reach a driver for action.
net/raw.h	The interface to the raw protocol family members *snoop* and *drain*.
net/if_arp.h	Generic ARP declarations.
netinet/if_ether.h	Essential declarations for Ethernet drivers, including ARP protocol for Ethernet.
sys/dlsap_register.h	DLPI interface declarations.

Debugging Facilities

When your driver is operating under a debugging kernel, you can use the facilities of symmon and idbg to display a variety of network-related data structures. See “Preparing the System for Debugging” on page 243, and see “Commands to Display Network-Related Structures” on page 269.

Information Sources

Aside from comments in header files, the complete *ifnet* interface and related interfaces have never been documented. In prior years, most people working on *ifnet* drivers have had access to the Berkeley UNIX source distribution and have been able to answer questions by referring to the code.

Referring to the code is an even more common option today, thanks to the release of 4.4BSD-Lite, a software distribution of BSD UNIX that does not require a source license, now widely available at a reasonable price. To obtain a copy, order the following:

- *4.4BSD-Lite Berkely Software Distribution CD-ROM Companion*, published by USENIX and O’Reilly & Associates; ISBN 1-56592-081-3 (US domestic) or ISBN 1-56592-092-9 (non-US).

The *ifnet* source code in this software is functionally compatible with IRIX *ifnet*, although some protocols (for example, *snoop* and *drain*) are not implemented in BSD-Lite.

In many respects, the *ifnet* interfaces and the logic of device drivers is dictated by Internet standards that are published as Requests for Comment (RFCs). The text of all RFCs can be obtained via FTP or with a WWW browser at *ftp://ds.internic.net/rfc/*.

Finally, the IRIX reference pages contain a wealth of detail regarding network interfaces. Some reference pages that are related to the interests of driver designers are listed in Table 16-1.

Table 16-1 Important Reference Pages Related to Network Drivers

Reference Page	Contents
arp(7)	Operation of the ARP protocol, with details of **ioctl()** functions.
drain(7)	Operation of the drain driver, which receives unwanted packets, with details of its **ioctl()** functions.
ethernet(7)	Overview of the IRIX Ethernet drivers, including error messages and the use of VECTOR lines to configure them.
fddi(7)	Cursory overview of IRIX FDDI drivers, with naming conventions.
ifconfig(1)	Management program used to enable and disable network interfaces (drivers) and change their runtime parameters.
netintro(7)	Overview of network facilities; mentions the role of the network interface (driver); has extensive detail on routing **ioctl()** calls.
network(1)	Documents the network initialization script that runs when the system is booted up.
raw(7)	Overview of the Raw protocol family whose members are snoop and drain.
routed(1)	Documents operation of the routing daemon, including **ioctl()** use.
snoop(7)	Operation of the snoop driver, which allows inspection of packets, with details of its **ioctl()** features.
ticlts(7)	Operation and use of the ticlts, ticots, and ticotsord loopback drivers.
tokenring(7)	Overview of the IRIX token-ring drivers, including packet formats.

Network Inventory Entries

When a device driver is initialized, it can install an entry in the system hardware inventory (see "Creating an Inventory Entry" on page 32). It is a good idea for a network driver to do this, because the *netsnoop* program relies on hardware inventory entries. After successfully initializing, a network device driver should call **add_to_inventory()** with the following five parameters (see *sys/invent.h*):

class	INV_NETWORK
type	The packet type, for example INV_NET_ETHER. See *sys/invent.h* for the possible "types for class network" list.
controller	The kind of network controller from the "controllers for network types" list in *sys/invent.h*.
unit	Any distinguishing number for this device. The *hinv* command does not decode this field.
state	Any characteristic number for this device. The *hinv* command does not decode this field.

Multiprocessor Considerations

Prior to IRIX 5.3, the kernel BSD framework code and TCP/IP protocol stack executed under a single kernel lock, creating a single-threaded implementation. Beginning with IRIX 5.3, the BSD framework and TCP/IP protocol suite have been multi-threaded to support symmetric multiprocessing. The code uses different kernel locks to protect different critical sections.

IRIX now supports multiple, concurrent threads of execution within the TCP/UDP/IP protocol suite, and the kernel socket layer. In addition, network device drivers run on any available CPU, concurrently with the network software, applications, and other drivers.

This means that any ifnet-based network driver must be prepared to run asynchronously and concurrently with other drivers and with the protocol stack.

Ineffective spl() Functions

The **spl*()** functions were the traditional UNIX method of gaining exclusive use of data. In single-threaded ifnet drivers, the **splimp()** or **splnet()** functions were used to get exclusive use of the ifnet structure.

In a multiprocessor, **spl*()** functions like **splimp()** or **splnet()** do block interrupts on the local CPU, but they do not prevent interrupts from occurring on other processors in the system, nor do they prevent other processes on other CPUs from executing code that refers to the same data.

If you are porting a driver from a uniprocessor environment, search for any use of an **spl*()** function and plan to replace it with effective mutual exclusion locking macros.

Multiprocessor Locking Macros

Under BSD networking, drivers interface with the protocol stacks by queueing incoming packets on a per-protocol input queue. In a multiprocessor, each protocol input queue must be protected by the locking macros defined in the file *net/if.h*.

All the locking macros that protect the input queue are assumed to be called at the proper processor interrupt masking level, **splimp**. All input queue locking macros also take an input parameter *ifq*, which is a pointer to the protocol input queue that must be defined as a *struct ifqueue*.

Compilation Flags for MP TCP/IP

The _MP_NETLOCKS and MP compiler variables must be defined in order to enable the macros necessary to run under multi-threaded TCP/IP. Make sure the following compiler options are used:

```
-D_MP_NETLOCKS -DMP
```

Mutual Exclusion Macros

The macros for mutual exclusion defined in *net/if.h* are listed in Table 16-2.

Table 16-2 Mutual Exclusion Macros for ifnet Drivers

Macro Prototype	**Purpose**
IFNET_LOCK(*ifp*, *s*)	Get exclusive use of the structure **ifp*. **splimp()** is called to raise the interrupt level if necessary, and the returned value is saved in *s*.
IFNET_UNLOCK(*ifp*,*s*)	Release use of **ifp*, and return to interrupt level *s*.
IFNET_LOCKNOSPL(*ifp*)	Get exclusive use of the structure **ifp*, but do not call **splimp()** (the driver knows it is already at the appropriate level.)
IFNET_UNLOCKNOSPL(*ifp*)	Release use of **ifp* after use of IFNET_LOCKNOSPL.
IFNET_ISLOCKED(*ifp*)	Test whether **ifp* is locked.
IFQ_LOCK(*ifq*)	Get exclusive use of an input queue **ifq*.
IFQ_UNLOCK(*ifq*)	Release use of **ifq*.
IF_ENQUEUE(*ifq*, *mp*)	Lock the queue **ifq*; post the mbuf **mp*; release the queue.
IF_ENQUEUE_NOLOCK(*ifq*,*mp*)	Post the mbuf **mp* without locking.

The variables used in Table 16-2 are as follows:

ifp	Address of a *struct ifnet* to be used exclusively.
s	Integer variable to store the current interrupt mask level.
ifq	Address of a *struct ifqueue* to be posted.
mp	Address of a *struct mbuf* to be posted.

Macro Use

The TCP/IP protocol stack automatically acquires the ifnet structure before calling a network driver routine through that structure. Thus the driver's **init()**, **stop()**, **start()**, **output()**, and **ioctl()** functions do not need to use IFNET_LOCK or IFNET_UNLOCK. Look for expressions

```
ASSERT(IFNET_ISLOCKED(ifp));
```

in the example driver ("Example ifnet Driver" on page 401) to see places where this is the case. Explicit use of IFNET_LOCK is needed in the interrupt handler.

Input Queueing Example

Example 16-1 displays a code fragment of an interrupt handler that queues an input packet pointed to by *m* onto the IP input queue. The function **schednetisr()** is called to schedule processing of that packet. The code is assumed to be already at **splimp()**.

Example 16-1 Input Queueing Using Locking Macros

```
{
...
    ifq = &ipintrq; /* the ip protocol queue */
 /*
 * If queue is full, we drop the packet.
 */
    IFQ_LOCK(ifq);
    if (IF_QFULL(ifq)) {
       m_freem(m);
       IF_DROP(ifq);
       IFQ_UNLOCK(ifq);
       return(-1);
    }
    IF_ENQUEUE_NOLOCK(ifq, m);
    schednetisr(NETISR_IP); /* schedule ip interrupt */
    IFQ_UNLOCK(ifq);
    return(0);
}
```

Interrupt Handler Example

Example 16-2 displays the skeleton of an Ethernet interrupt handler.

Example 16-2 Interrupt Handling Using Locking Macros

```
/*
 * Ethernet interface interrupt.
 */
if_etintr(int unit)
{
 ETIO io;
 struct et_info *ei;
 register int s = splimp(); /* get the spin lock */

 ASSERT(unit == 0);
 ei = &et_info;
 io = ei->ei_io;

 if (io == 0) { /* ignore early interrupts */
     printf("et0: early interrupt\n");
     splx(s);
     return 1;
 }
 IFNET_LOCKNOSPL(&ei->ei_if);
 et_poll(ei);
 IFNET_UNLOCKNOSPL(&ei->ei_if);
 splx(s);
}
```

Example ifnet Driver

The code in Example 16-3 represents the skeleton of an ifnet driver, showing its entry points, data structures, required **ioctl**() functions, address format conventions, and its use of kernel utility routines and locking primitives.

A comment beginning “MISSING:” represents a point at which a complete driver would contain code related to the device or bus it manages.

Example 16-3 Skeleton ifnet Driver

```
/*
 * Locking strategy:
 * IFNET_LOCK() and IFNET_UNLOCK() acquire/release the
 * lock on a given ifnet structure. IFQ_LOCK() and
 * IFQ_UNLOCK() acquire/release the lock on a given ifqueue
 * structure. The ifnet or ifqueue lock must be held while
 * modifying any fields within the associated data
 * structure. The ifnet lock is also held to singlethread
 * portions of the device driver. The driver xxinit,
 * xxreset, xxoutput, xxwatchdog, and xxioctl entry points
 * are called with IFNET_LOCK() already acquired thus only
 * a single thread of execution is allowed in these
 * portions of the driver for each interface. It is the
 * driver’s responsibility to call IFNET_LOCK() within its
 * xxintr() and other private routines to singlethread any
 * other critical sections.  It is also the driver’s
 * responsibility to acquire the ifq lock by calling
 * IFQ_LOCK() before attempting to enqueue onto the IP
 * input queue "ipintrq".
 *
 * Notes:
 * - don’t forget appropriate machine-specific cache flushing operations
 *    (see “Managing Memory for Cache Coherency” on page 200)
 * - declare pointers to device registers as "volatile"
 * - compile on multiprocessor systems with "-D_MP_NETLOCKS -DMP"
 *
 * Copyright 1994 Silicon Graphics, Inc.  All rights reserved.
 */
#ident "$Revision: 1.0$"

#include <sys/types.h>
#include <sys/param.h>
#include <sys/systm.h>
#include <sys/sysmacros.h>
```

```
#include <sys/cmn_err.h>
#include <sys/debug.h>
#include <sys/edt.h>
#include <sys/errno.h>
#include <sys/tcp-param.h>
#include <sys/mbuf.h>
#include <sys/immu.h>
#include <sys/sbd.h>
#include <sys/ddi.h>
#include <sys/cpu.h>
#include <sys/invent.h>
#include <net/if.h>
#include <net/if_types.h>
#include <net/netisr.h>
#include <netinet/if_ether.h>
#include <net/raw.h>
#include <net/multi.h>
#include <netinet/in_var.h>
#include <net/soioctl.h>
#include <sys/dlsap_register.h>
/* MISSING: local includes and defines */
#define    WORDALIGNED(p)    (p & (sizeof(int)-1) == 0)
#define    SK_MAX_UNITS    8
#define    SK_MTU       4096
#define    SK_DOG       (2*IFNET_SLOWHZ) /* watchdog duration in seconds */
#define    SK_IFT       (IFT_FDDI)    /* refer to <net/if_types.h> */
#define    SK_INV       (INV_NET_FDDI)    /* refer to <sys/invent.h> */
#define    INV_FDDI_SK    (23)       /* refer to <sys/invent.h> */
#define    IFF_ALIVE       (IFF_UP|IFF_RUNNING)
#define    iff_alive(flags)    (((flags) & IFF_ALIVE) == IFF_ALIVE)
#define iff_dead(flags)        (((flags) & IFF_ALIVE) != IFF_ALIVE)
#define    SK_ISBROAD(addr)    (!bcmp((addr), &skbroadcastaddr, SKADDRLEN))
#define    SK_ISGROUP(addr)    ((addr)[0] & 01)
/* MISSING: media-specific definitions of address size and header format */
#define    SKADDRLEN    (6)
#define    SKHEADERLEN    (sizeof (struct skheader))
/*
 * Our hypothetical medium has an IEEE 802-like header.
 */
struct skaddr {
    u_int8_t sk_vec[SKADDRLEN];
};
struct skheader {
    struct skaddr sh_dhost;
    struct skaddr sh_shost;
    u_int16_t sh_type;
};
struct skaddr skbroadcastaddr = {
    0xff, 0xff, 0xff, 0xff, 0xff, 0xff
};
/*
 * Each interface is represented by a private
 * network interface data structure that maintains
 * the device hardware resource addresses, pointers
 * to device registers, allocated dma_alloc maps,
 * lists of mbufs pending transmit or reception, etc, etc.
 * This hypothetical driver uses ARP and have an 802 address.
 */
struct sk_info {
    struct arpcom si_ac;        /* common ifnet and arp */
    struct skaddr si_ouraddr;   /* our individual media address */
    struct mfilter si_filter;   /* AF_RAW sw snoop filter */
    struct rawif si_rawif;      /* raw snoop interface */
    int si_unit;
    int si_flags;
    int si_initdone;
    /* MISSING: other per-interface fields */
};
#define    si_if    si_ac.ac_if
#define    sktoifp(si) (&(si)->si_ac.ac_if)
#define ifptosk(ifp)((struct sk_info *)ifp)
struct sk_info sk_info[SK_MAX_UNITS];
/* MISSING: other device-dependent structures */
/*
 * The start of an mbuf containing an input frame
 */
struct sk_ibuf {
    struct ifheader sib_ifh;
    struct snoopheader sib_snoop;
    struct skheader sib_skh;
};
#define SK_IBUFSZ (sizeof (struct sk_ibuf))
/*
 * Multicast filter request for SIOCADDMULTI/SIOCDELMULTI .
 */
struct mfreq {
    union mkey *mfr_key;    /* pointer to socket ioctl arg */
    mval_t    mfr_value;    /* associated value */
};
/*
```

```
 * forward declarations of internal subfunctions.
 */
static void skedtinit(struct edt *e);
static int sk_init(int unit);
static void sk_reset(struct sk_info *si);
static void sk_intr(int unit);
static int sk_output(struct ifnet *ifp, struct mbuf *m, struct sockaddr *dst);
static void sk_input(struct sk_info *si, struct mbuf *m, int totlen);
static int sk_ioctl(struct ifnet *ifp, int cmd, void *data);
static void sk_watchdog(int unit);
static void sk_stop(struct sk_info *si);
static int sk_start(struct sk_info *si, int flags);
static int sk_add_da(struct sk_info *si, union mkey *key, int ismulti);
static int sk_del_da(struct sk_info *si, union mkey *key, int ismulti);
static int sk_dahash(char *addr);
static int sk_dlp(struct sk_info *si, int port, int encap, struct mbuf *m, int
len);
/*
 * global ifnet structures
 */
extern struct ifqueue ipintrq;    /* ip input queue */
extern struct ifnet loif;         /* loopback driver if */
/*
 * EDT initialization routine, called by lboot(1) when processing a VECTOR
 * statement from /var/sysgen/system/*.sm.
 */
static void
skedtinit(struct edt *e)
{
    struct sk_info *si;
    struct ifnet *ifp;
    int    unit;
    /* MISSING: hardware-dependent local variables. */
    /*
     * Depending on the bus attachment, VME, GIO, or EISA,
     * refer to the appropriate part of this book for information on
     * bus-dependent support for probing and initializing a device,
     * allocating interrupt vectors, and registering the interrupt handler.
     */
    /* MISSING: bus- and device-dependent code to initialize the device. */
    /*
     * Other driver-specific actions that might go here:
     * - allocate an unused unit number and initialize
     *   that sk_info structure.
     * - call sk_reset to disable the device
     * - allocate shared host/device memory
     * - allocate DMA and PIO maps
     */
    if (showconfig)
        cmn_err(CE_NOTE,"sk%d: hardware MAC address %s\n",
            si->si_unit,
            sk_sprintf(si->si_ouraddr));
    /*
     * MISSING: address translation protocol goes here.
     * Save a copy of the MAC address in the arpcom structure.
     */
    bcopy((caddr_t)&si->si_ouraddr, (caddr_t)si->si_ac.ac_enaddr,
        SKADDRLEN);
    /*
     * Initialize ifnet structure with our name, type, mtu size,
     * supported flags, pointers to our entry points,
     * and attach to the available ifnet drivers list.
     */
    ifp = sktoifp(si);
    ifp->if_name = "sk";
    ifp->if_unit = unit;
    ifp->if_type = SK_IFT;
    ifp->if_mtu = SK_MTU;
    ifp->if_flags = IFF_BROADCAST | IFF_MULTICAST | IFF_NOTRAILERS;
    ifp->if_init = (int (*)(int))sk_init;
    ifp->if_output = sk_output;
    ifp->if_ioctl = (int (*)(struct ifnet*, int, void*))sk_ioctl;
    ifp->if_watchdog = sk_watchdog;
    if_attach(ifp);
    /*
     * Allocate a multicast filter table with an initial
     * size of 10.  See <net/multi.h> for a description
     * of the support for generic sw multicast filtering.
     * Use of these mf routines is purely optional -
     * if you’re not supporting multicast addresses or
     * your device does perfect filtering or you think
     * you can roll your own better, feel free.
     */
    if (!mfnew(&si->si_filter, 10))
        cmn_err(CE_PANIC, "sk_edtinit: no memory for frame filter\n");
    /*
     * Initialize the raw socket interface.  See <net/raw.h>
     * and the man pages for descriptions of the SNOOP
     * and DRAIN raw protocols.
     */
```

```
    rawif_attach(&si->si_rawif, &si->si_if,
        (caddr_t) &si->si_ouraddr,
        (caddr_t) &skbroadcastaddr,
        SKADDRLEN,
        SKHEADERLEN,
        structoff(skheader, sh_shost),
        structoff(skheader, sh_dhost));
    /*
     * Add the initialized network interface to the hardware inventory.
     */
    add_to_inventory(INV_NETWORK, SK_INV, INV_FDDI_SK, unit, 0);
}
/*
 * The sk_init() entry point, called ??? */
static int
sk_init(int unit)
{
    struct sk_info *si;
    struct ifnet *ifp;
    /* MISSING: device-dependent local variables */
    si = &sk_info[unit];
    ifp = sktoifp(si);
    ASSERT(IFNET_ISLOCKED(ifp));
    /*
     * Reset the device first, ask questions later..
     */
    sk_reset(si);
    /*
     * - free or reuse any pending xmit/recv mbufs
     * - initialize device configuration registers, etc.
     * - allocate and post receive buffers
     *
     * See such topics in Chapter 9, “Device Driver/Kernel Interface”
     * as “Setting Up a DMA Transfer” on page 198, and see the
     * appropriate bus-related section (VME, GIO, EISA) for methods
     * of mapping DMA and PIO addresses.
     */
    /*
     * enable if_flags device behavior (IFF_DEBUG on/off, etc.)
     */
    /* MISSING: general device reset and initialize. */
    ifp->if_timer = SK_DOG;    /* turn on watchdog */
    /* turn device "on" now */
    /* MISSING: open the device for business. */
    return 0;
```

```
}
/*
 * Reset the interface.
 */
static void
sk_reset(struct sk_info *si)
{
    struct ifnet *ifp = sktoifp(si);
    ifp->if_timer = 0;    /* turn off watchdog */
    /* MISSING:
     * - reset device
     * - reset device receive descriptor ring
     * - free any enqueued transmit mbufs
     * - create device xmit descriptor ring
     */
}
/*
 * Interrupt handler.
 */
static void
sk_intr(int unit)
{
    register struct sk_info *si;
    struct ifnet *ifp;
    struct mbuf *m;
    struct ifqueue *ifq;
    int totlen;
    int s;
    int error;
    int port;
    si = &sk_info[unit];
    ifp = sktoifp(si);
    /*
     * Ignore early interrupts.
     */
    if ((si->si_initdone == 0) || iff_dead(ifp->if_flags)) {
        sk_stop(si);
        return;
    }
    /*
     * acquire interface lock
     */
    IFNET_LOCK(ifp, s);
    /*
     * MISSING: disable device and return if early interrupt
     */
    /*
     * MISSING: test and clear device interrupt pending register.
     */
    /*
     * process any received packets.
     */
    while (/* MISSING: received packets available */) {
        /*
         * MISSING: device-specific receive processing here.
         * Allocate and post a replacement receive buffer.
         */
        sk_input(si, m, totlen);
    }
    while (/* MISSING: mbuf has completed transmission */) {
        /*
         * Reclaim a completed device transmit resource
         * freeing completed mbufs, checking for errors,
         * and maintaining if_opackets, if_oerrors,
         * if_collisions, etc.
         */
    }
    IFNET_UNLOCK(ifp, s);
}
/*
 * Transmit packet.  If the destination is this system or
 * broadcast, send the packet to the loop-back device if
 * we cannot hear ourself transmit.  Return 0 or errno.
 */
static int
sk_output(
    struct ifnet    *ifp,
    struct mbuf *m0,
    struct sockaddr *dst)
{
    struct    sk_info    *si = ifptosk(ifp);
    struct skaddr *sdst, *ssrc;
    struct skheader *sh;
    struct mbuf *m, *m1, *m2;
    struct mbuf *mloop;
    int error;
    u_int16_t type;
    /* MISSING: other local variables. */
    ASSERT(IFNET_ISLOCKED(ifp));
    mloop = NULL;
```

```
if (iff_dead(ifp->if_flags)) {
    error = EHOSTDOWN;
    goto bad;
}
/*
 * If send queue full, try reclaiming some completed
 * mbufs.  If it’s still full, then just drop the
 * packet and return ENOBUFS.
 */
if (IF_QFULL(&si->si_if.if_snd)) {
    while (/* MISSING: transmits done */) {
        /*
         * Reclaim completed transmit descriptors.
         */
        IF_DEQUEUE_NOLOCK(&si->si_if.if_snd, m);
        m_freem(m);
    }
    if (IF_QFULL(&si->si_if.if_snd)) {
        m_freem(m0);
        si->si_if.if_odrops++;
        IF_DROP(&si->si_if.if_snd);
        return (ENOBUFS);
    }
}
switch (dst->sa_family) {
case AF_INET: {
    /*
     * Get room for media header,
     * use this mbuf if possible.
     */
    if (!M_HASCL(m0)
        && m0->m_off >= MMINOFF+sizeof(*sh)
        && (sh = mtod(m0, struct skheader*))
        && WORDALIGNED((u_long)sh)) {
        ASSERT(m0->m_off <= MSIZE);
        m1 = 0;
        --sh;
    } else {
        m1 = m_get(M_DONTWAIT, MT_DATA);
        if (m1 == NULL) {
            m_freem(m0);
            si->si_if.if_odrops++;
            IF_DROP(&si->si_if.if_snd);
            return (ENOBUFS);
        }
```

```
                sh = mtod(m1, struct skheader*);
                m1->m_len = sizeof (*sh);
            }
            bcopy(&si->si_ouraddr, &sh->sh_shost, SKADDRLEN);
            /*
             * translate dst IP address to media address.
             */
            if (!ip_arpresolve(&si->si_ac, m0,
                &((struct sockaddr_in *)dst)->sin_addr,
                (u_char*)&sh->sh_dhost)) {
                m_freem(m1);
                return (0);    /* just wait if not yet resolved */
            }
            if (m1 == 0) {
                m0->m_off -= sizeof (*sh);
                m0->m_len += sizeof (*sh);
            } else {
                m1->m_next = m0;
                m0 = m1;
            }
            /*
             * Listen to ourself, if we are supposed to.
             */
            if (SK_ISBROAD(&sh->sh_shost)) {
                mloop = m_copy(m0, sizeof (*sh), M_COPYALL);
                if (mloop == NULL) {
                    m_freem(m0);
                    si->si_if.if_odrops++;
                    IF_DROP(&si->si_if.if_snd);
                    return (ENOBUFS);
                }
            }
            break;
        }

        case AF_UNSPEC:
    #define EP ((struct ether_header *)&dst->sa_data[0])
            /*
             * Translate an ARP packet using RFC-1042.
             * Require the entire ARP packet be in the first mbuf.
             */
            sh = mtod(m0, struct skheader*);
            if (M_HASCL(m0)
                || !WORDALIGNED((u_long)sh)
                || m0->m_len < sizeof(struct ether_arp)
                || m0->m_off < MMINOFF+sizeof(*sh)
                || EP->ether_type != ETHERTYPE_ARP) {
                printf("sk_output: bad ARP output\n");
                m_freem(m0);
                si->si_if.if_oerrors++;
                IF_DROP(&si->si_if.if_snd);
                return (EAFNOSUPPORT);
            }
            ASSERT(m0->m_off <= MSIZE);
            m0->m_len += sizeof(*sh);
            m0->m_off -= sizeof(*sh);
            --sh;
            bcopy(&si->si_ouraddr, &sh->sh_shost, SKADDRLEN);
            bcopy(&EP->ether_dhost[0], &sh->sh_dhost, SKADDRLEN);
            sh->sh_type = EP->ether_type;
# undef EP
            break;

        case AF_RAW:
            /* The mbuf chain contains the raw frame incl header.
             */
            sh = mtod(m0, struct skheader*);
            if (M_HASCL(m0)
                || m0->m_len < sizeof(*sh)
                || !WORDALIGNED((u_long)sh)) {
                m0 = m_pullup(m0, SKHEADERLEN);
                if (m0 == NULL) {
                    si->si_if.if_odrops++;
                    IF_DROP(&si->si_if.if_snd);
                    return (ENOBUFS);
                };
                sh = mtod(m0, struct skheader*);
            }
            break;

        case AF_SDL:
#define SCKTP ((struct sockaddr_sdl *)dst)
            /*
             * Send an 802 packet for DLPI.
             * mbuf chain should already have everything
             * but MAC header. First, sanity-check the MAC address.
             */
            if (SCKTP->ssdl_addr_len != SKADDRLEN) {
                m_freem(m0);
                return (EAFNOSUPPORT);
            }
            sh = mtod(m0, struct skheader*);
            if (!M_HASCL(m0)
                && m1->m_off >= MMINOFF+SCKTP_HLEN
                && WORDALIGNED(sh)) {
                ASSERT(m0->m_off <= MSIZE);
                m0->m_len += SCKTP_HLEN;
                m0->m_off -= SCKTP_HLEN;
            } else {
                m1 = m_get(M_DONTWAIT,MT_DATA);
                if (!m1) {
                    m_freem(m0);
                    si->si_if.if_odrops++;
                    IF_DROP(&si->si_if.if_snd);
                    return (ENOBUFS);
                }
                m1->m_len = SCKTP_HLEN;
                m1->m_next = m0;
                m0 = m1;
                sh = mtod(m0, struct skheader*);
            }
            sh->sh_type = htons(ETHERTYPE_IP);
            bcopy(&si->si_ouraddr, &sh->sh_shost, SKADDRLEN);
            bcopy(SCKTP->ssdl_addr, &sh->sh_dhost, SKADDRLEN);
            break;
# undef SCKTP

        default:
            printf("sk_output:  bad af %u\n", dst->sa_family);
            m_freem(m0);
            return (EAFNOSUPPORT);
        } /* switch (dst->sa_family) */

        /*
         * Check whether snoopers want to copy this packet.
         */
        if (RAWIF_SNOOPING(&si->si_rawif)
            && snoop_match(&si->si_rawif, (caddr_t)sh, m0->m_len)) {
            struct mbuf *ms, *mt;
            int len;        /* m0 bytes to copy */
            int lenoff;
            int curlen;
            len = m_length(m0);
            lenoff = 0;
            curlen = len + SK_IBUFSZ;
            if (curlen > MCLBYTES)
                curlen = MCLBYTES;
            ms = m_vget(M_DONTWAIT, MAX(curlen, SK_IBUFSZ), MT_DATA);
            if (ms) {
                IF_INITHEADER(mtod(ms,caddr_t), &si->si_if, SK_IBUFSZ);
                curlen = m_datacopy(m0, lenoff, curlen - SK_IBUFSZ,
                    mtod(ms,caddr_t) + SK_IBUFSZ);
                mt = ms;
                for (;;) {
                    lenoff += curlen;
                    len -= curlen;
                    if (len <= 0)
                        break;
                    curlen = MIN(len, MCLBYTES);
                    m1 = m_vget(M_DONTWAIT, curlen, MT_DATA);
                    if (0 == m1) {
                        m_freem(ms);
                        ms = 0;
                        break;
                    }
                    mt->m_next = m1;
                    mt = m1;
                    curlen = m_datacopy(m0, lenoff, curlen,
                                mtod(m1, caddr_t));
                }
            }
            if (ms == NULL) {
                snoop_drop(&si->si_rawif, SN_PROMISC,
                       mtod(m0,caddr_t), m0->m_len);
            } else {
                (void)snoop_input(&si->si_rawif, SN_PROMISC,
                          mtod(m0, caddr_t),
                          ms,
                          (lenoff > SKHEADERLEN)?
                          (lenoff - SKHEADERLEN) : 0);
            }
        }
        /*
         * Save a copy of the mbuf chain to free later.
         */
        IF_ENQUEUE_NOLOCK(&si->si_if.if_snd, m0);
        /*
         * MISSING: Start DMA on the msg.
         * - allocate device-specific xmit resources  (need max
         *   of twice the number of mbufs in the mbuf chain
         *   if we’re using physical memory addresses for
         *   GIO assuming worst case that each mbuf crosses
         *   a page boundary.
         */
        if (error)
            goto bad;
        ifp->if_opackets++;
        if (mloop) {
            si->si_if.if_omcasts++;
            (void) looutput(&loif, mloop, dst);
        } else if (SK_ISGROUP(sh->sh_dhost.sk_vec))
            si->si_if.if_omcasts++;
        return (0);

    bad:
        ifp->if_oerrors++;
        m_freem(m);
        m_freem(mloop);
        return (error);
    }
    /*
     * deal with a complete input frame in a string of mbufs.
     * mbuf points at a (struct sk_ibuf), totlen is #bytes
     * in user data portion of the mbuf.
     */
    static void
    sk_input(struct sk_info *si,
        struct mbuf *m,
        int totlen)
    {
        struct sk_ibuf *sib;
        struct ifqueue *ifq;
        int snoopflags = 0;
        uint port;
        /*
         * MISSING: set ‘snoopflags’ and ‘if_ierrors’ as appropriate
         */
        ifq = NULL;
        sib = mtod(m, struct sk_ibuf*);
        IF_INITHEADER(sib, &si->si_if, SK_IBUFSZ);
        si->si_if.if_ibytes += totlen;
        si->si_if.if_ipackets++;
        /*
         * If it is a broadcast or multicast frame,
         * get rid of imperfectly filtered multicasts.
     */
    if (SK_ISGROUP(sib->sib_skh.sh_dhost.sk_vec)) {
        if (SK_ISBROAD(sib->sib_skh.sh_dhost.sk_vec))
            m->m_flags |= M_BCAST;
        else {
            if (((si->si_ac.ac_if.if_flags & IFF_ALLMULTI) == 0)
            && !mfethermatch(&si->si_filter,
                sib->sib_skh.sh_dhost.sk_vec, 0)) {
                if (RAWIF_SNOOPING(&si->si_rawif)
                && snoop_match(&si->si_rawif,
                    (caddr_t) &sib->sib_skh, totlen))
                    snoopflags = SN_PROMISC;
                else {
                    m_freem(m);
                    return;
                }
                m->m_flags |= M_MCAST;
            }
        }
        si->si_if.if_imcasts++;
    } else {
        if (RAWIF_SNOOPING(&si->si_rawif)
            && snoop_match(&si->si_rawif,
                (caddr_t) &sib->sib_skh,
                totlen))
            snoopflags = SN_PROMISC;
        else {
            m_freem(m);
            return;
        }
    }

    /*
     *  Set ‘port’ .  For this example, just sh_type.
     */
    port = ntohs(sib->sib_skh.sh_type);
    /*
     * do raw snooping.
     */
    if (RAWIF_SNOOPING(&si->si_rawif)) {
        if (!snoop_input(&si->si_rawif, snoopflags,
                 (caddr_t)&sib->sib_skh,
                 m,
                 (totlen>sizeof(struct skheader)
                  ? totlen-sizeof(struct skheader) : 0))) {
        }
        if (snoopflags)
            return;
    } else if (snoopflags) {
        goto drop;    /* if bad, count and skip it */
    }
    /*
     * If it is a frame we understand, then give it to the
     * correct protocol code.
     */
    switch (port) {
    case ETHERTYPE_IP:
        ifq = &ipintrq;
        break;
    case ETHERTYPE_ARP:
        arpinput(&si->si_ac, m);
        return;
    default:
        if (sk_dlp(si, port, DL_ETHER_ENCAP, m, totlen))
            return;
        break;
    }
    /*
     * if we cannot find a protocol queue, then flush it down the
     * drain, if it is open.
     */
    if (ifq == NULL) {
        if (RAWIF_DRAINING(&si->si_rawif)) {
            drain_input(&si->si_rawif,
                    port,
                    (caddr_t)&sib->sib_skh.sh_dhost.sk_vec,
                    m);
        } else
            m_freem(m);
        return;
    }
    /*
     * Put it on the IP protocol queue.
     */
    if (IF_QFULL(ifq)) {
        si->si_if.if_iqdrops++;
        si->si_if.if_ierrors++;
        IF_DROP(ifq);
        goto drop;
    }
```

```
    IF_ENQUEUE(ifq, m);
    schednetisr(NETISR_IP);
    return;
drop:
    m_freem(m);
    if (RAWIF_SNOOPING(&si->si_rawif))
        snoop_drop(&si->si_rawif, snoopflags,
               (caddr_t)&sib->sib_skh, totlen);
    if (RAWIF_DRAINING(&si->si_rawif))
        drain_drop(&si->si_rawif, port);
}
/*
 * See if a DLPI function wants a frame.
 */
static int
sk_dlp(struct sk_info *si,
    int port,
    int encap,
    struct mbuf *m,
    int len)
{
    dlsap_family_t *dlp;
    struct mbuf *m2;
    struct sk_ibuf *sib;
    if ((dlp = dlsap_find(port, encap)) == NULL)
        return (0);
    /*
     * The DLPI code wants the entire MAC and LLC headers.
     * It needs the total length of the mbuf chain to reflect
     * the actual data length, not to be extended to contain
     * a fake, zeroed LLC header which keeps the snoop code from
     * crashing.
     */
    if ((m2 = m_copy(m, 0, len+sizeof(struct skheader))) == NULL)
        return (0);
    if (M_HASCL(m2)) {
        m2 = m_pullup(m2, SK_IBUFSZ);
        if (m2 == NULL)
            return (0);
    }
    sib = mtod(m2, struct sk_ibuf*);
    /*
     * The DLPI code wants the MAC address in canonical bit order.
     * MISSING: Convert here if necessary.
     */
    /*
     * The DLPI code wants the LLC header, if present,
     * not to be hidden with the MAC header.  Decrement
     * LLC header size from ifh_hdrlen if necessary.
     */
    if ((*dlp->dl_infunc)(dlp, &si->si_if, m2, &sib->sib_skh)) {
        m_freem(m);
        return (1);
    }
    m_freem(m2);
    return (0);
}
/*
 * Process an ioctl request. Return 0 or errno.
 */
static int
sk_ioctl(
    struct ifnet *ifp,
    int cmd,
    void *data)
{
    struct sk_info *si;
    int error = 0;
    int flags;
    /* MISSING: device-dependent local variables */
    ASSERT(IFNET_ISLOCKED(ifp));
    si = ifptosk(ifp);
    switch (cmd) {
    case SIOCSIFADDR:
    {
        struct ifaddr *ifa = (struct ifaddr *)data;
        switch (ifa->ifa_addr.sa_family) {
        case AF_INET:
            sk_stop(si);
            si->si_ac.ac_ipaddr = IA_SIN(ifa)->sin_addr;
            sk_start(si, ifp->if_flags);
            break;
        case AF_RAW:
            /*
             * Not safe to change addr while the
             * board is alive.
             */
            if (!iff_dead(ifp->if_flags))
                error = EINVAL;
            else {
                bcopy(ifa->ifa_addr.sa_data,
                    si->si_ac.ac_enaddr, SKADDRLEN);
                error = sk_start(si, ifp->if_flags);
            }
            break;
        default:
            error = EINVAL;
            break;
        } /* inner switch (ifa->ifa_addr.sa_family) */
        break;
    } /* case SIOCSIFADDR */
    case SIOCSIFFLAGS:
    {
        flags = ((struct ifreq *)data)->ifr_flags;
        if (((struct ifreq*)data)->ifr_flags & IFF_UP)
            error = sk_start(si, flags);
        else
            sk_stop(si);
        break;
    }
    case SIOCADDMULTI:
    case SIOCDELMULTI:
    {
#define MKEY ((union mkey*)data)
        int allmulti;
        /*
         * Convert an internet multicast socket address
         * into an 802-type address.
         */
        error = ether_cvtmulti((struct sockaddr *)data, &allmulti);
        if (0 == error) {
            if (allmulti) {
                if (SIOCADDMULTI == cmd)
                    si->si_if.if_flags |= IFF_ALLMULTI;
                else
                    si->si_if.if_flags &= ~IFF_ALLMULTI;
                /* MISSING: enable hw all multicast addrs */
            } else {
                bitswapcopy(MKEY->mk_dhost, MKEY->mk_dhost,
                    sizeof (MKEY->mk_dhost));
                if (SIOCADDMULTI == cmd)
                    error = sk_add_da(si, MKEY, 1);
                else
                    error = sk_del_da(si, MKEY, 1);
            }
        }
        break;
#undef MKEY
    } /* case SIOCADDMULTI, SIOCDELMULTI */
    case SIOCADDSNOOP:
    case SIOCDELSNOOP:
    {
#define SF(nm) ((struct skheader*)&(((struct snoopfilter *)data)->nm))
        /*
         * raw protocol snoop filter.  See <net/raw.h>
         * and <net/multi.h> and the snoop(7P) man page.
         */
        u_char *a;
        union mkey key;
        a = &SF(sf_mask[0])->sh_dhost.sk_vec[0];
        if (!SK_ISBROAD(a)) {
            /*
             * cannot filter on device unless mask is trivial.
             */
            error = EINVAL;
        } else {
            /*
             * Filter individual destination addresses.
             * Use a different address family to avoid
             * damaging an ordinary multi-cast filter.
             * MISSING: You’ll have to invent your own
             * multicast filter routines if this doesn’t
             * fit your address size or needs.
             */
            a = &SF(sf_match[0])->sh_dhost.sk_vec[0];
            key.mk_family = AF_RAW;
            bcopy(a, key.mk_dhost, sizeof (key.mk_dhost));
            if (cmd == SIOCADDSNOOP)
                error = sk_add_da(si, &key, SK_ISGROUP(a));
            else
                error = sk_del_da(si, &key, SK_ISGROUP(a));
        }
        break;
    } /* case SIOCADD/DELSNOOP */
    /*
     * MISSING: driver-specific ioctl cases here.
     */
    default:
        error = EINVAL;
    }
    return (error);
}
/*
 * Add a destination address.
 * Add address to the sw multicast filter table and to
 * our hw device address (if applicable).
 */
static int
sk_add_da(
    struct sk_info *si,
    union mkey *key,
    int ismulti)
{
    struct mfreq mfr;
    /*
     * mfmatchcnt() looks up key in our multicast filter
     * and, if found, just increments its refcnt and
     * returns true.
     */
    if (mfmatchcnt(&si->si_filter, 1, key, 0))
        return (0);

    mfr.mfr_key = key;
    mfr.mfr_value = (mval_t) sk_dahash(key->mk_dhost);
    if (!mfadd(&si->si_filter, key, mfr.mfr_value))
        return (ENOMEM);

    /* MISSING: poke this hash into device’s hw address filter */
    return (0);
}
/*
 * Delete an address filter. If key is unassociated, do nothing.
 * Otherwise delete software filter first, then hardware filter.
 */
sk_del_da(
    struct sk_info *si,
    union mkey *key,
    int ismulti)
{
    struct mfreq mfr;
    /*
     * Decrement refcnt of this address in our multicast filter
     * and reclaim the entry if refcnt == 0.
     */
    if (mfmatchcnt(&si->si_filter, -1, key, &mfr.mfr_value))
```

```
        return (0);
    mfdel(&si->si_filter, key);
    /* MISSING: disable this hash value from the device if necessary */
    return (0);
}
/*
 * compute a hash value for destination addr
 */
static int
sk_dahash(char *addr)
{
    int    hv;
    hv = addr[0] ^ addr[1] ^ addr[2] ^ addr[3] ^ addr[4] ^ addr[5];
    return (hv & 0xff);
}
/*
 * Periodically poll the device for input packets
 * in case an interrupt gets lost or the device
 * somehow gets wedged.  Reset if necessary.
 */
static void
sk_watchdog(int unit)
{
    struct sk_info *si;
    struct ifnet *ifp;
    int s;

    si = &sk_info[unit];
    ifp = sktoifp(si);
    ASSERT(IFNET_ISLOCKED(ifp));
    /* MISSING: poll the device, handle accordingly */
}
/*
 * Disable the interface.
 */
static void
sk_stop(struct sk_info *si)
{
    struct ifnet *ifp = sktoifp(si);
    ASSERT(IFNET_ISLOCKED(ifp));
    ifp->if_flags &= ~IFF_ALIVE;
    /*
     * Mark an interface down and notify protocols
     * of the transition.
     */
    if_down(ifp);
    sk_reset(si);
}
/*
 * Enable the interface.
 */
static int
sk_start(
    struct sk_info *si,
    int flags)
{
    struct ifnet *ifp = sktoifp(si);
    int    error;
    ASSERT(IFNET_ISLOCKED(ifp));
    error = sk_init(si->si_unit);
    if (error || (ifp->if_addrlist == NULL))
        return error;
    ifp->if_flags = flags | IFF_ALIVE;
    /*
     * Broadcast an ARP packet, asking who has addr
     * on interface ac.
     */
    arpwhohas(&si->si_ac, &si->si_ac.ac_ipaddr);
    return (0);
}
```

PART SEVEN

EISA Drivers

Chapter 17: EISA Device Drivers
Overview of the architecture of the EISA bus attachment and the services offered by the kernel to EISA device drivers.

Chapter 17

EISA Device Drivers

The EISA (Extended Industry Standard Architecture) bus is supported by the Silicon Graphics Indigo^2, POWER Indigo^2™, and Indigo^2 Maximum IMPACT™ systems, and the Challenge M. This chapter contains the following topics related to support for the EISA bus:

- "The EISA Bus in Silicon Graphics Systems" on page 428 gives an overview of the EISA bus features and implementation.
- "Kernel Functions for EISA Support" on page 439 discusses the kernel functions that are specifically used by EISA device drivers.
- "Sample EISA Driver Code" on page 447 displays a complete character driver for an EISA device.

Note: Often it is most practical to control an EISA device through programmed I/O from a user-level process. For information on PIO, turn to "EISA Programmed I/O" on page 67 after reading "The EISA Bus in Silicon Graphics Systems" on page 428.

For information on the general architecture of a kernel-level device driver, see Part III, "Kernel-Level Drivers."

The EISA Bus in Silicon Graphics Systems

The EISA (Extended Industry Standard Architecture) bus is an enhancement of the ISA (Industry Standard Architecture) bus standard originally developed by IBM.

EISA Bus Overview

EISA is backward compatible with ISA, but expands the ISA data bus from 16 bits to 32 bits, and provides 23 more address lines and 16 more indicator and control lines.

The EISA bus supports the following features:

- all ISA transfers
- bus master devices
- burst-mode DMA transfers
- 32-bit data and address paths
- peer-to-peer card communication

For detailed information on EISA-bus protocols, electrical specifications, and operation, see the standards documents ("Standards Documents" on page xxxviii). Figure 17-1 shows the high-level design of the EISA attachment in the Indigo^2 architecture.

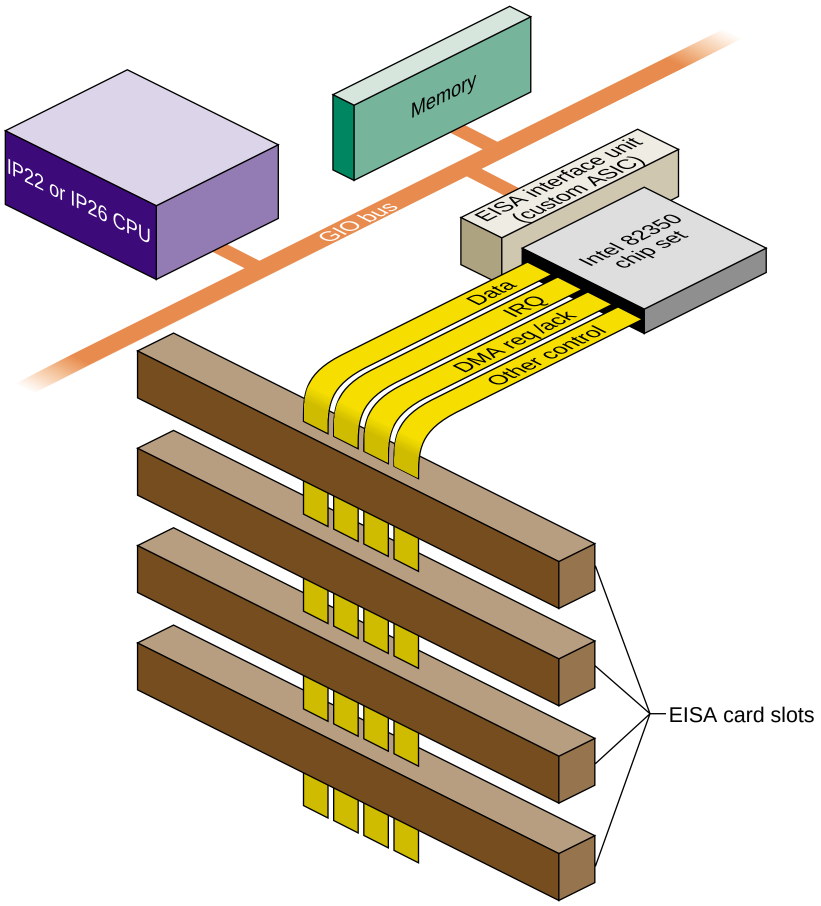

Figure 17-1 High-Level Overview of EISA Bus in Indigo^2

EISA Request Arbitration

EISA provides server DMA channels arranged into two channel groups (channels 0-3 and channels 5-7) for priority resolution. Silicon Graphics uses the rotating scheme described in the EISA specification. Although the channels rotate in this scheme, channels 5-7 receive more cycles, in general, than channels 0-3.

EISA Interrupts

The EISA bus supports 11 edge-triggerable or level-triggerable interrupts. IRQ0–IRQ2, IRQ8, and IRQ13 are reserved for internal functions and are not available to EISA cards. The remaining 11 interrupt lines (IRQ3–IRQ7, IRQ9–IRQ12, IRQ14, IRQ15) can be generated by EISA cards. Multiple cards can use one IRQ level, so long as they use the same triggering method.

All EISA-generated interrupts are transmitted to a single interrupt level on the Silicon Graphics CPU (see “Interrupt Priority Scheduling” on page 435.)

EISA Data Transfers

The EISA bus supports 8-bit, 16-bit, and 32-bit data transfers through direct CPU access (PIO) as well as DMA initiated by a bus-master card or the on-board DMA hardware.

EISA Address Spaces

The EISA-bus address space is divided into I/O address space and memory address space. On the EISA bus, accesses to memory and to I/O are distinguished by having different bus cycle protocols. The MIPS architecture has only one type of memory access, so in the Silicon Graphics systems, EISA I/O space and memory space are assigned separate ranges of physical addresses. The EISA Interface Unit (see Figure 17-1 on page 429) decodes the address ranges and causes the Intel 82350 bus control to issue the appropriate bus cycle type, I/O or memory.

The I/O address space comprises a sequence of 4 KB page, one for each bus slot. The first page, slot 0, corresponds to the registers of the Intel 82350 chip set. The pages for slots 1-4 correspond to the four accessible slots in the Indigo^2 and Challenge M chassis (see “Available Card Slots” on page 434).

EISA Locked Cycles

The EISA bus architecture provides a signal, LOCK*, which allows a card (or the processor, in an Intel architecture system) to lock bus access so as to perform one or more atomic updates.

The Silicon Graphics hardware implementation of the EISA bus is bridged onto the GIO bus, which does not support a locked cycle. The general form of locked bus cycles is not supported in the Silicon Graphics implementation of EISA. An EISA card cannot lock the bus nor can software in the IRIX kernel lock the EISA bus.

A device driver in the IRIX kernel can perform a software-controlled read-modify-write cycle, as on a VME bus, using the **pio_*_rmw()** kernel functions. See ("Using the PIO Map in Functions" on page 441). This function ensures that no other software accesses the EISA bus during the read-modify-write operation.

EISA Byte Ordering

An important implementation detail of the EISA bus is that it uses the Intel convention of "little-endian" byte ordering, in which the least significant byte of a halfword or word is in the lowest address. The Silicon Graphics CPU uses "big-endian" ordering, with the most significant byte first. Hence data exchanged with the EISA bus often needs to be reordered before use.

EISA Product Identifier

EISA expansion boards, embedded devices, and system boards have a four-byte product identifier (ID) that can be read from I/O port addresses 0x*s*C80 through 0x*s*C83 in the card's I/O address space, where *s* is the offset of the card slot. For example, the slot 1 product ID can be read as a 4-byte value from I/O port addresses 0x1C80. This value can be tested in an exprobe parameter of the VECTOR line during system boot (see "Configuring IRIX" on page 436).

The first two bytes (0x*s*C80 and 0x*s*C81) contain a compressed representation of the manufacturer code. The manufacturer code is a three-character code (uppercase ASCII characters in the range of A to Z) chosen by the manufacturer and registered with the standard (see "Standards Documents" on page xxxviii). The manufacturer code "ISA" is used to indicate a generic ISA adapter.

The three-character manufacturer code is compressed into three 5-bit values so that it can be incorporated into the two I/O bytes at 0x*s*C80 and 0x*s*C81. The compression procedure is as follows:

1. Find the hexadecimal ASCII value for each letter:

   ASCII for "A"-"Z" : "A" = 0x41, "Z" = 0x5a

2. Subtract 0x40 from each ASCII value:

   Compressed "A" = 0x41-0x40 = 0x01 = 0000 0001
   Compressed "Z" = 0x5a-0x40 = 0x1A = 0001 1010

3. Discard leading 0-bits, retaining the five least significant bits of each letter:

   Compressed "A" = 00001. Compressed "Z" = 11010

4. Compressed code = concatenate "0" and the three 5-bit values:

   "AZA" = 0 00001 11010 00001

Figure 17-2 summarizes the contents of the EISA manufacturer ID value.

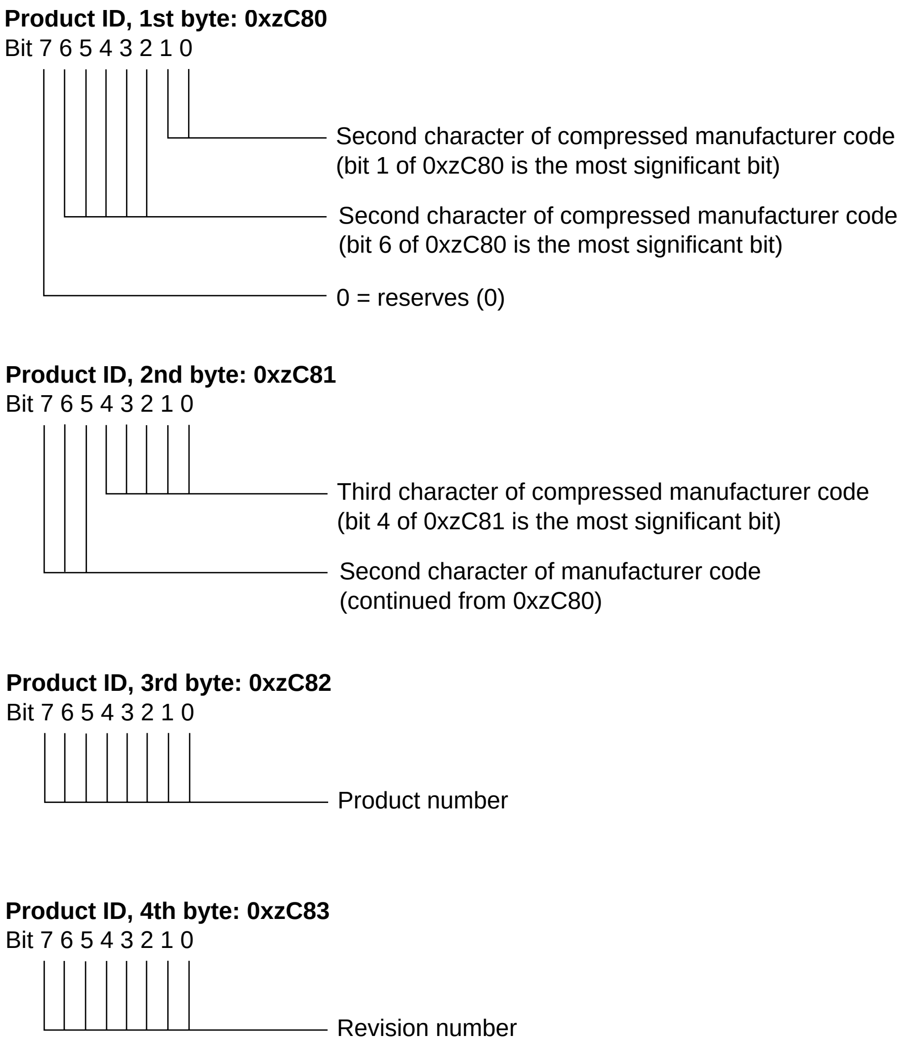
Product ID, 1st byte: 0xzC80
Bit 7 6 5 4 3 2 1 0
Second character of compressed manufacturer code (bit 1 of 0xzC80 is the most significant bit)
Second character of compressed manufacturer code (bit 6 of 0xzC80 is the most significant bit)
0 = reserves (0)
Product ID, 2nd byte: 0xzC81
Bit 7 6 5 4 3 2 1 0
Third character of compressed manufacturer code (bit 4 of 0xzC81 is the most significant bit)
Second character of manufacturer code (continued from 0xzC80)
Product ID, 3rd byte: 0xzC82
Bit 7 6 5 4 3 2 1 0
Product number
Product ID, 4th byte: 0xzC83
Bit 7 6 5 4 3 2 1 0
Revision number

Figure 17-2 Encoding of the EISA Manufacturer ID

EISA Support in Indigo^2 and Challenge M Series

One or more EISA cards can be plugged into an Indigo^2 series workstation, or into a Challenge M system (which uses the identical chassis). Any EISA-conforming card can be plugged into an available slot. EISA devices can be used as block devices or character devices, but they cannot be used as boot devices.

Available Card Slots

The Indigo^2 series has four peripheral card slots that accept graphics adapters, EISA cards, or GIO cards in any combination. Graphics cards are available that use one, two, or three slots, resulting in the following combinations:

- With Extreme graphics installed, one slot is available for use by an EISA card.
- With XZ graphics installed, two slots are used by the graphics, and two are available for EISA cards.
- The XL graphics uses only one slot, so up to three EISA cards can be accommodated.

The Challenge M system, having no graphics adapter, has four available slots.

EISA Address Mapping

The pages of EISA I/O address space are mapped to physical addresses 0x0001 0000 (slot 1) through 0x0004 0000 (slot 4). The 112 MB of EISA memory address space is mapped to physical addresses between 0x000A 0000 and 0x06FF FFFF.

Addresses in these ranges can be mapped into the kernel address space for PIO or for DMA (see “Kernel Functions for EISA Support” on page 439).

Interrupt Priority Scheduling

The EISA architecture associates interrupt priority with the IRQ level, from IRQ0 to IRQ15. In Silicon Graphics systems, all EISA interrupts are channeled into one CPU interrupt level. The priority of this CPU interrupt is below that of the clock and at the same level as on-board devices. When multiple EISA interrupts arrive, they are serviced in their EISA-bus priority order.

When the CPU receives an EISA-bus interrupt, it responds to each interrupt level in IRQ priority order (lower number first). For each interrupt level, the IRIX kernel calls one or more interrupt service functions that have been established by device drivers (see "Allocating IRQs and Channels" on page 442).

EISA Configuration

In order to integrate an EISA device into a Silicon Graphics system you must configure the EISA card itself, and then configure the system to recognize the card.

Configuring the Hardware

The I/O address space on an EISA card plugged into a card slot responds to the range of bus addresses for that slot. All EISA cards are identified by a manufacturer-specific device ID that the operating system uses to register the existence of each card. ISA cards, in contrast, are jumpered to respond to a specific address range that corresponds to the device's I/O registers.

Normally a kernel-level driver accesses registers in the I/O space using a PIO map (see "Mapping PIO Addresses" on page 439). For a card's memory space to be accessible, the card must be configured or jumpered to respond to the appropriate address range. The specified address range must be selected to avoid conflicts with other EISA/ISA devices.

Configuring IRIX

In the PC/DOS hardware and software environment, where the EISA bus is commonly found, device configuration is handled in part by use of a standalone ROM BIOS initialization program that stores device information in the nonvolatile RAM of the PC; and in part by saving device initialization information in configuration files that are read at boot time.

Neither of these facilities is available in the same way under IRIX. Each EISA device is configured to IRIX using a VECTOR line in a file stored in the directory */var/sysgen/system* (see "System Configuration Files" on page 39).

The syntax of a VECTOR line is documented in two places:

- The */var/sysgen/system/irix.sm* file itself contains descriptive comments on the syntax and meaning of the statement, as well as numerous examples.
- The system(4) reference page gives a more formal definition of the syntax.

In a Silicon Graphics system equipped with an EISA bus, the file */var/sysgen/system/irix.sm* contains a number of VECTOR lines describing the EISA devices supported by distributed code.

The important elements in a VECTOR line for EISA are as follows:

bustype	Specified as *EISA* for EISA devices. The VECTOR statement can be used for other types of buses as well.
module	The base name of a kernel-level device driver for this device, as used in the */var/sysgen/master.d* database (see "Master Configuration Database" on page 38 and "How Names Are Used in Configuration" on page 232).
adapter	The number of the EISA bus where the device is attached—always 0, or omitted, in current systems.
ctlr	The "controller" number is simply an integer parameter that is passed to the device driver at boot time. It can be used for example to specify a slot number.
iospace, *iospace2*, *iospace3*	Each *iospace* group specifies the address space, the starting address, and the size of a segment of address space used by this device.
probe or *exprobe*	Specifies a hardware test that can be applied at boot time to find out if the device exists.

The following is a typical VECTOR line for an EISA device (it must be a single physical line in the file):

```
VECTOR: bustype=EISA module=if_ec3 ctlr=1
iospace=(EISAIO,0x1000,0x1000)
exprobe_space=(r,EISAIO, 0x1c80,4,0x6010d425,0xffffffff)
```

Using the iospace Parameters

The *iospace*, *iospace2*, and *iospace3* parameters are used to pass ranges of device addresses to the device driver. Each parameter contains the following three items:

- A keyword for the address space, either *EISAIO* or *EISAMEM*.
- The starting address, which depends on the address space and the card itself, as follows:
  - For the I/O space of an EISA card, the starting address of I/O registers is 0x1000 multiplied by the slot number of the card (from 1 to 4), and extends for a length of 0x1000 (4096). For example, the manufacturer ID of the card in slot 2 is at address 0x1C80.
  - The I/O space of an ISA card is hard-wired or jumpered on the card, and falls in the range 0x0100 to 0x0400.
  - The EISAMEM space is card-dependent and falls in the range 0x000A 0000 through 0x06FF FFFF.
- The length of this bus address range.

The values in these parameters are passed to the device driver at its *pfx***edtinit()** entry point, provided that the probe shows the device is active.

Using the probe and exprobe Parameters

You use the *probe* or *exprobe* parameter to program a test for the existence of the device at boot time. When no test is specified, *lboot* assumes the device exists. Then it is up to the device driver to determine if the device is active and usable. When the device does not respond to a probe (because it is off-line or because it has been removed from the system), the *lboot* command will not invoke the device driver for this device.

An example exprobe parameter is as follows:

```
exprobe_space=(r,EISAIO, 0x1c80,4,0x6010d425,0xffffffff)
```

The exprobe parameter lists groups of six subparameters, as follows:

Sequence	One or more of *w* for write, *r* for read, or *rn* for read-negate.
Space	*EISAIO* or *EISAMEM*.
Address	The address of the byte, halfword, or word to test.
Length	The number of bytes to test: 1, 2, or 4.
Value	The value to write, or the test value for a read.
Mask	A number to be ANDed with the Value operand before a write or after a read. Specify 0xffffffff to nullify the AND operation.

You can use the *w* operation to prime a device. You can use the *r* operation to test for a specific value, and the *rn* operation to test that a specific value (or a specific bit, after masking) is *not* returned.

Typically, a simple *r* operation is used on an EISA card to test for the manufacturer's product identifier.

To test the existence of an ISA card, use a *wr* sequence to write a value to a register and read it back unchanged. Or read a value and verify that it does not come back all-binary-1, the value returned by a nonexistent device.

Using the module Parameter

The device driver specified by the *module* parameter is invoked at its *pfx***edtinit()** entry point, where it receives *ctlr* and *iospace* information specified in the VECTOR line (see "Entry Point edtinit()" on page 144). The device driver initializes the device at this time.

You use the *iospace* parameters to pass in the exact bus addresses that correspond to this device. Up to three address space ranges can be passed to the driver. This does not restrict the device—it can use other ranges of addresses, but the device driver has to deduce their addresses from other information. The device driver typically uses this data to set up PIO maps (see "Mapping PIO Addresses" on page 439).

Kernel Functions for EISA Support

The kernel provides services for mapping the EISA bus into the kernel virtual address space for PIO or DMA, and for transferring data using these maps. Two types of DMA are supported, Bus-master DMA and Slave DMA.

Mapping PIO Addresses

A PIO map is a system object that represents the mapping of a location in kernel virtual memory to some range of addresses on a VME or EISA bus. After creating a PIO map, a device driver can use it in the following ways:

- Extract a specific kernel virtual address that represents the device. This address can be used to load or store data, or it can be mapped that into user process space.
- Copy data between the device and memory without learning the specific kernel addresses involved.
- Perform bus read-modify-write cycles to apply Boolean operators to device data.

The functions used with PIO maps are summarized in Chapter 17, “EISA Device Drivers.”.

Table 17-1 Functions to Create and Use PIO Maps

Function	Header Files	Can Sleep	Purpose
pio_mapalloc(D3)	pio.h & types.h	Y	Allocate a PIO map.
pio_mapfree(D3)	pio.h & types.h	N	Free a PIO map.
pio_badaddr(D3)	pio.h & types.h	N	Check for bus error when reading an address.
pio_wbadaddr(D3)	pio.h & types.h	N	Check for bus error when writing to an address.
pio_mapaddr(D3)	pio.h & types.h	N	Convert a bus address to a virtual address.
pio_bcopyin(D3)	pio.h & types.h	Y	Copy data from a bus address to kernel’s virtual space.
pio_bcopyout(D3)	pio.h & types.h	Y	Copy data from kernel’s virtual space to a bus address.

Table 17-1 (continued) Functions to Create and Use PIO Maps

Function	**Header Files**	**Can Sleep**	**Purpose**
pio_andb_rmw(D3)	pio.h & types.h	N	Byte read-AND-write cycle.
pio_andh_rmw(D3)	pio.h & types.h	N	16-bit read-AND-write cycle.
pio_andw_rmw(D3)	pio.h & types.h	N	32-bit read-AND-write cycle.
pio_orb_rmw(D3)	pio.h & types.h	N	Byte read-OR-write cycle.
pio_orh_rmw(D3)	pio.h & types.h	N	16-bit read-OR-write cycle.
pio_orw_rmw(D3)	pio.h & types.h	N	32-bit read-OR-write cycle.

A kernel-level device driver creates a PIO map by calling **pio_mapalloc()**. This function performs memory allocation and so can sleep. PIO maps are typically created in the *pfx***edtinit()** entry point, where the driver first learns about the device addresses from the contents of the *edt_t* structure (see "Entry Point edtinit()" on page 144).

The parameters to **pio_mapalloc()** describe the range of addresses that can be mapped in terms of

- the bus type, in this case ADAP_EISA from *sys/edt.h*
- the bus number, when more than one bus is supported
- the address space, using constants such as PIOMAP_EISA_IO from *sys/pio.h*
- the starting bus address and a length

This call also specifies a "fixed" or "unfixed" map. This distinction applies only to VME maps. An EISA map is always a fixed map.

A call to **pio_mapfree()** releases a PIO map. PIO maps created by a loadable driver must be released in the *pfx***unload()** entry point (see "Entry Point unload()" on page 167 and "Unloading" on page 240).

Testing the PIO Map

The PIO map is created from the parameters that are passed. These are not validated by **pio_mapalloc()**. If there is any possibility that the mapped device is not installed, not active, or improperly configured, you should test the mapped address.

The **pio_baddr()** and **pio_wbaddr()** functions test the mapped address to see if it is usable.

Using the Mapped Address

From a fixed PIO map you can recover a kernel virtual address that corresponds to the first bus address in the map. The **pio_mapaddr()** function is used for this.

You can use this address to load or store data into device registers. In the *pfx***map()** entry point (see "Concepts and Use of mmap()" on page 158), you can use this address with the **v_mapphys()** function to map the range of device addresses into the address space of a user process.

Using the PIO Map in Functions

You can apply a variety of kernel functions to any PIO map, fixed or unfixed. The **pio_bcopyin()** and **pio_bcopyout()** functions copy a range of data between memory and a PIO map. There is no performance advantage to using these functions, as compared to loading or storing to the mapped addresses, but their use makes the device driver code simpler and more readable.

The series of functions **pio_andb_rmw()** and **pio_orb_rmw()** perform a read-modify-write cycle. You can use them to set or clear bits in device registers. Read-modify-write cycles on the EISA bus are atomic operations to software only (see "EISA Locked Cycles" on page 431).

Allocating IRQs and Channels

Before a kernel-level driver can field EISA interrupts, it must associate a handler with one of the IRQ levels. In order to perform DMA, the driver must allocate one of the DMA channels. The functions used for these purposes are summarized inTable 17-2

Table 17-2 Functions for IRQ and Channel Allocation

Function	Header Files	Can Sleep	Purpose
eisa_dmachan_alloc()	eisa.h & types.h	N	Allocate DMA channel.
eisa_ivec_alloc()	eisa.h & types.h	N	Allocate IRQ and set triggering.
eisa_ivec_set()	eisa.h & types.h	N	Associate handler to IRQ.

Note: There are no reference pages for the functions in Table 17-2.

Allocating and Programming an IRQ

The function **eisa_ivec_alloc()** allocates an available IRQ number from a set of acceptable numbers. Its prototype is

```
int eisa_ivec_alloc(uint_t adap,ushort_t mask,uchar_t trig);
```

The arguments are as follows:

adap	The adapter number, always 0 in current systems.
mask	A 16-bit mask containing a 1-bit for each IRQ level that is acceptable for this device. (For available IRQ levels, see "EISA Interrupts" on page 430.)
trig	The triggering method used by the card, either EISA_EDGE_IRQ or EISA_LEVEL_IRQ from *sys/eisa.h*.

ISA cards are usually hard-wired or jumpered to a particular IRQ, so that the *mask* argument contains a single bit. Some EISA cards can be programmed dynamically to use a selected IRQ; in that case *mask* contains a 1-bit for each IRQ the card can be programmed to use.

The function attempts to allocate an IRQ from the *mask* set that is not in use by any card. If all acceptable levels are in use, it allocates an IRQ that is already in use with the requested kind of triggering. In either case, it returns the number of the IRQ to be used.

In the event that all the IRQs requested are already in use with a conflicting type of triggering, the function returns -1.

After allocating an IRQ, the device driver programs the card (using PIO) to interrupt on that line.

The function **eisa_ivec_set()** associates a function in the device driver with an IRQ number. Its prototype is

```
int eisa_ivec_set(uint_t adap, int irq,
                  void (*e_intr)(long), long e_arg)
```

The parameters are as follows:

adap	The adapter number, always 0 in current systems.
irq	The IRQ level to be monitored.
e_intr	The address of the interrupt handling function to call.
e_arg	An argument to pass to the function when called.

When more than one device is allocated the same IRQ, the kernel calls all the interrupt functions associated with that IRQ. This means that an interrupt function must always verify, by testing device registers, that the interrupt was caused by its device.

The first call to **eisa_ivec_set()** for a given IRQ enables interrupts from that IRQ. Prior to the call, interrupts from that IRQ are ignored.

Note: If you are working with both the VME and EISA interfaces, it is worth noting that the number and type of arguments of **eisa_ivec_set()** differ from those of **vme_ivec_set()**.

Note: There is no way to retract the association of an interrupt function with an IRQ. This means that if an EISA driver handles interrupts and is loadable, it must not support the *pfx***unload()** entry point. An interrupt arriving after the driver had been unloaded would panic the system.

Allocating a DMA Channel

The function **eisa_dmachan_alloc()** allocates one of the seven available DMA channels (channel 4 is reserved by the hardware) from a set of acceptable channels. The function's prototype is

```
int eisa_dmachan_alloc(uint_t adap, uchar_t dma_mask)
```

The arguments are as follows:

adap	The adapter number, always 0 in current systems.
dma_mask	An 8-bit mask containing 1-bits for the DMA channels that can be used by this device.

The function allocates the channel in the requested set that is in use by the fewest devices. It is possible for a single channel to be requested by multiple devices. However, if the device can use any of several channels, it is likely that the device will be the only one using the channel whose number is returned. After allocating a channel number, the device driver programs the device to use that channel, if necessary.

Programming Bus-Master DMA

Bus-master DMA is performed by an EISA card that has bus-master logic. The card generates the DMA bus cycles, and provides the target memory address to store or retrieve data.

The device driver sets up Bus-master DMA by programming the card with a target physical address and length of data. Some cards support scatter/gather operations, in which the card is programmed with a list of memory pages and their lengths, and the card transfers a stream of data across all of the pages. However, programming an EISA bus master card is a highly hardware-dependent operation. The cards vary widely in their capabilities and programming methods.

The key programming issue for a device driver is locating the target memory buffers in system memory, so as to be able to program the EISA card with correct physical memory addresses.

The kernel provides functions for mapping memory for DMA. The functions that operate on EISA DMA maps are summarized in Table 17-3.

Table 17-3 Functions That Operate on DMA Maps

Function	Header Files	Can Sleep	Purpose
dma_map(D3)	dmamap.h & types.h & sema.h	N	Prepare DMA mapping.
dma_mapaddr(D3)	dmamap.h & types.h & sema.h	N	Return the target physical address for a given map and address.
dma_mapalloc(D3)	dmamap.h & types.h & sema.h	Y	Allocate a DMA map.
dma_mapfree(D3)	dmamap.h & types.h & sema.h	N	Free a DMA map.

A device driver allocates a DMA map using **dma_mapalloc()**. This is typically done in the *pfx***edtinit()** entry point, provided that the maximum I/O size is known at that time (see "Entry Point edtinit()" on page 144).

A DMA map is used prior to a DMA transfer into or out of a buffer in kernel virtual space. The function **dma_map()** takes a DMA map, a buffer address, and a length. It relates the buffer address to physical addresses for use in DMA, and returns the length mapped. The returned length is typically less than the length of the buffer. This is because, for EISA, the function does not support scatter/gather, so the mapping must stop at the first page boundary.

After calling **dma_map()**, the device driver calls **dma_mapaddr()** to get the physical address corresponding to the current map. This is the address that is programmed into the EISA bus master card as a target address for a segment of the transfer up to one page in size.

Repeated calls to **dma_map()** and **dma_mapaddr()** can be used to map successive pages, until the EISA card is loaded with as many transfer segment addresses as it supports.

Programming Slave DMA

In Slave DMA, an EISA card that does not have DMA logic is commanded by the EISA Interface Unit and 82350 chip set (see Figure 17-1 on page 429) to perform a series of transfers into memory.

The kernel supplies a unique set of functions for managing Slave DMA, unrelated to the DMA functions for Bus-master DMA. The functions that operate on EISA DMA maps are summarized in Table 17-4.

Table 17-4 Functions for EISA DMA

Function	Header Files	Can Sleep	Purpose
eisa_dma_disable(D3)	eisa.h & types.h	N	Disable recognition of hardware requests on a DMA channel.
eisa_dma_enable(D3)	eisa.h & types.h	N	Enable recognition of hardware requests on a DMA channel.
eisa_dma_free_buf(D3)	eisa.h & types.h	N	Free a previously allocated DMA buffer descriptor.
eisa_dma_free_cb(D3)	eisa.h & types.h	N	Free a previously allocated DMA command block.
eisa_dma_get_buf(D3)	eisa.h & types.h	Y	Allocate a DMA buffer descriptor.
eisa_dma_get_cb(D3)	eisa.h & types.h	Y	Allocate a DMA command block.
eisa_dma_prog(D3)	eisa.h & types.h	Y	Program a DMA operation for a subsequent software request.
eisa_dma_stop(D3)	eisa.h & types.h	N	Stop software-initiated DMA operation and release channel.
eisa_dma_swstart(D3)	eisa.h & types.h	Y	Initiate a DMA operation via software request.

The EISA attachment hardware has many options for performing Slave DMA, and most of these options are reflected in the contents of the *eisa_dma_cb* and *eisa_dma_buf* data structures (see the eisa_dma_buf(D4) and eisa_dma_cb(D4) reference pages, in addition to the reference pages listed in Table 17-4). By setting appropriate values declared in *sys/eisa.h* into these structures, you can program most varieties of Slave DMA.

Sample EISA Driver Code

Initialization Sketch

The code in Example 17-1 represents an outline of the *pfx***edtinit()** entry point for a hypothetical EISA device, showing the allocation of a PIO map, an IRQ, and a DMA channel. The driver supports as many as four identical devices. It keeps information about them in an array of structures, *einfo*. Each entry to *pfx***edtinit()** initializes one element of this array, as indexed by the *ctlr* value from the VECTOR statement.

An important point to note in this example is that most of the arguments to **pio_map_alloc()** can simply be passed as the values from the *edt_t* received by the entry point.

Example 17-1 Sketch of EISA Initialization

```
#include <sys/types.h>
#include <sys/edt.h>
#include <sys/pio.h>
#include <sys/eisa.h>
#include <sys/cmn_err.h>
#define MAX_DEVICE 4
/* Array of info structures about each device. A device
** that does not initialize OK ought to be marked, but
** no such logic is shown.
*/
struct edrv_info {
   caddr_t e_addr[NBASE];     /* pio mapped addr per space */
   int     e_dmachan;         /* dma chan in use */
} einfo[MAX_DEVICE];

#define CARD_ID         0x0163b30a   /* mfr. ID */
#define IRQ_MASK        0x0018       /* acceptable IRQs */
#define DMACHAN_MASK    0x7a         /* acceptable chans */
```

```
edrv_edtinit(edt_t *e)
{
   int iospace;         /* index over iospace array */
   int eirq;            /* allocated IRQ # */
   int edma_chan;       /* allocated chan # */
   struct edrv_info *einf; /* -> einfo[n] */
   piomap_t *pmap;

   if (e->e_ctlr < MAX_DEVICE)
      einf = &einfo[e->e_ctlr];
   else
   { /* unknown device, nowhere to put info */
      cmn_err(CE_WARN,"devno too large:%d",e->e_ctlr);
      return;
   }

/* for each nonempty iospace parameter,
** set up a PIO map and save the kv address.
*/
   for (iospace = 0; iospace < NBASE; iospace++) {
      if (!e->e_space[iospace].ios_iopaddr)
         einf->e_addr[iospace] = 0; /* note no addr */
      pmap = pio_mapalloc( /* make a PIO map */
            e->e_bus_type,   /* pass bus type given */
            e->e_adap, /* pass adapter # given */
            &e->e_space[iospace], /* given iospace too */
            PIOMAP_FIXED, /* always fixed for EISA */
            "edrv");
      einf->e_addr[iospace] = pio_mapaddr(pmap,
            e->e_space[iospace].ios_iopaddr);
   }
/* Set up an edge-triggered IRQ for this device.
** Associate it with our interrupt entry point.
** There is no need to remember the assigned IRQ.
*/
   eirq = eisa_ivec_alloc(e->e_adap,IRQ_MASK,EISA_EDGE_IRQ);
   if (eirq < 0) {
      cmn_err(CE_WARN,
         "edrv: ctlr %d could not allocate IRQ\n”,
         e->e_ctlr);
      /* should mark einfo unusable */
      return;
   }
   eisa_ivec_set(e->e_adap, eirq, edrv_intr, e->e_ctlr);
/* Allocate a DMA Channel for this device and note
** the number in the device info array.
*/
   edma_chan = eisa_dmachan_alloc(e->e_adap,DMACHAN_MASK);
   if (edma_chan < 0) {
      cmn_err(CE_WARN,
            "edrv: ctlr %d could not allocate DMA Chan\n",
            e->e_ctlr);
      /* should mark einfo unusable */
      return;
   }
   einf->e_dmachan = edma_chan;
}
```

Complete EISA Character Driver

The code in this section displays a complete character device driver for an EISA card, the Roland RAP-10 synthesizer. This inexpensive synthesizer card can be installed in an Indigo^2 and driven by a program through this device driver.

- Example 17-6 on page 452 displays the code of the driver itself.
- Example 17-2 on page 449 displays the descriptive file to be placed in */var/sysgen/master.d* to describe the driver.
- Example 17-3 on page 450 displays the configuration file to be placed in */var/sysgen/system* to enable loading the driver.
- Example 17-4 on page 450 displays a shell script to install the driver.
- Example 17-5 on page 450 contains a test program to operate the synthesizer.

Example 17-2 Master File /var/sysgen/rap for RAP-10 Driver

```
*
* rap  - Roland RAP-10 Musical Board
*
* $Revision: 1.0 $
*
*FLAG   PREFIX  SOFT    #DEV    DEPENDENCIES
c   rap     61      -

$$$
```

Example 17-3 Configuration File /var/sysgen/rap.sm for RAP-10 Driver

```
VECTOR: bustype=EISA module=rap ctlr=0 adapter=0
iospace=(EISAIO,0x330,16) probe_space=(EISAIO,0x330,1)
```

Example 17-4 Installation Script for RAP-10 Driver

```
#!/bin/csh

if [ `whoami` != “root” ]
then
  echo “You must be root to run this script.\n”
  exit 1
fi

echo “cp rap.o /var/sysgen/boot/rap.o\n”
cp rap.o /var/sysgen/boot/rap.o

echo “cp rap.master /var/sysgen/master.d/rap\n”
cp rap.master /var/sysgen/master.d/rap

echo “cp rap.sm /var/sysgen/system/rap.sm”
cp rap.sm /var/sysgen/system/rap.sm

echo “mknod /dev/rap c 62 0\n”
mknod /dev/rap c 62 0

echo “Make a new kernel anytime by typing: autoconfig -f -v\n”
```

Example 17-5 Program to Test RAP-10 Driver

```
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <signal.h>
#include “rap.h”
/*
 *  record.c
 *
 *  This program plays song from a previuosly recorded file
 *  using RAP-10 board.
 *
 */

#define    BUF_SIZE    4096
```

```
#define    FILE_HDR    “RAP-10 WAVE FILE”
#define    RAP_FILE    “/dev/rap”
#define    MAX_BUF     10
#define    FOREVER     for(;;)

uchar_t    buf[BUF_SIZE];
uchar_t    *fname;
void       endProg( int );

main (int argc, char **argv)
{
    register int    fd, rapfd, bytes;
    if ( argc <= 1 ) {
        printf (“play: Usage: play <file_name>\n”);
        exit(0);
    }
    fname = argv[1];
    printf (“play: opening file %s\n”, fname);
    fd = open (fname, O_RDONLY);
    if ( fd == -1 ) {
        printf (“play: Cannot create file, errno = %d\n”, errno);
        close(rapfd);
        exit(0);
    }
    printf (“play: Checking RAP-10 File ID\n”);
    if ( read(fd, buf, strlen(FILE_HDR)) <= 0 ) {
        printf (“play: Could not read the file ID, errno = %d\n”,
        errno);
        close(fd);
        exit(0);
    }
    if ( strcmp(buf, FILE_HDR) ) {
        printf (“play: File is not a RAP file\n”);
        close(fd);
        exit(0);
    }
    printf (“play: opening RAP card\n”);
    rapfd = open (RAP_FILE, O_WRONLY);
    if ( rapfd <= 0 ) {
        printf (“play: Cannot open RAP card, errno = %d\n”,
        errno);
        exit(0);
    }
    printf (“play: Playing ..please wait\n”);
    /*   ignore Interrupt   */
    sigset (SIGINT, SIG_IGN );
    FOREVER {
        bytes = read(fd, buf, BUF_SIZE);
        if ( bytes < 0 ) {
            printf (“play: error reading data, errno = %d”,
                errno);
            close(fd);
            close(rapfd);
            exit(0);
        }
        if ( bytes == 0 )
            break;
        bytes = write(rapfd, buf, BUF_SIZE);
        if ( bytes <= 0 ) {
            printf (“play: Cannot read from RAP, errno = %d\n”,
            errno);
            close (rapfd);
            close (fd);
            exit(0);
        }
    }
    printf (“play: waiting for Play to End\n”);
    if ( ioctl (rapfd, RAPIOCTL_END_PLAY) ) {
           printf (“play: Ioctl error %d”, errno );
    }
    else printf (“play: Song succesfully played\n”);
    close(rapfd);
    close (fd);
}
```

Example 17-6 Complete EISA Character Driver for RAP-10

```
/******************************************************************************
 *
 *          Roland RAP-10 Music Card Device Driver for Eisa Bus
 *          ---------------------------------------------------
 *
 *   INTRODUCTION:
 *   -------------
 *   This file contains the device driver for Roland RAP-10
 *   Music Card. Currently it contains necessary routines to Record and
 *   Playback a  Wave file. The MIDI Implementation is to be defined and
 *   implemented at later time.
 *
 *   DESIGN OVERVIEW:
 *   ----------------
 *   We will use DMA for wave data movements. At any given time, the card
 *   can be either playing or recording and both operations are not allowed.
 *   Also no more than one process at a time can access the card.
 *
 *   Circular Buffers:
 *   -----------------
 *   Since DMA operation is performed independently of the processor,
 *   we will buffer the user’s data and release the user’s process to
 *   do other things (i.e. preparing more data). Internally we use a
 *   circular queue (rwQue) to store the data to be played or recorded.
 *   Each entry in this queue is of the type rwBuf_t where the data will
 *   be stored.  Each entry can store up to RW_BUF_SIZE bytes of data.
 *   At the init time, we try to allocate two DMA channels for the card:
 *   Channel 5 and 6. If we can only allocate Channel 5, we will use the
 *   card in Mono mode, otherwise, we will use it as Stereo. DMA has two
 *   buffers of its own: dmaRigh[] and dmaLeft[] for each Channel. For
 *   Stereo play, the data user provides us is of the format:
 *
 *       <Left Byte><Right Byte><Left Byte><Right Byte>.....
 *
 *   So for playing, we have to move all Left_Bytes to dmaLeft buffer
 *   and all Right_Bytes to the dmaRight buffer (in Stereo mode only).
 *   In mono mode, we will use dmaLeft[] buffer and all the user’s data
 *   are moved to dmaLeft[].
 *
 *   The basic operation of the Card are as follow:
 *
 *   Playing:
 *   --------
 *   For playing wave data, the user must first open the card through
 *   open() system call.The call comes to us as rapopen(). This
 *   routine resets all global values, states and counters, prepares
 *   necessary DMA structures for each channel, disables RAP-10
 *   interrupts and establishes this process as the owner of the card.
 *
 *   The user provides us with the wave data by issuing write()
 *   system calls. This call comes to us as rapwrite(). We will
 *   move the data from user’s address space into an empty rwQue[]
 *   entry and will retrun so that the user can issue another call.
 *   If there is no DMA going, we will start one and the data will
 *   start to be moved to the Card to be played.
 *   The user can issue as many write() as necessary. The playing
 *   operation will be done by either closing the card or issuing
 *   an Ioctl call. Issuing Ioctl, will leave this process as owner
 *   still while closing the card will release the card.
 *
 *   Recording:
 *   ----------
 *   Assuming that the user has opened the card and is the current
 *   owner, user will issue read() system call. The call comes to
 *   us as rapread(). If no DMA Record is going on, we will start
 *   one. We will move data from rwQue[] entries (as they are filled)
 *   to user’s address space. The recording is done either by a
 *   close() or ioctl() call.
 *
 *   DMA Starting:
 *   -------------
 *   For Playing, we will start DMA when we have a full circular buffer.
 *   This is done so that we have enough data available for a fast DMA
 *   operation to be busy with. For recording, we will start DMA
 *   immediatly.
 *
 *   Interrupts:
 *   -----------
 *   For each DMA transfer, we will receive two interrupts: One when 1st
 *   half the buffer is transfered, one when 2nd half of the buffer is
 *   transfered.  We must fill the half that has just been transfered with
 *   fresh data.  Note that in Stereo mode, there are two DMA operation
 *   going. So when we receive Interrupt for one DMA, we must wait for the
 *   exact interrupt from the other DMA and service both DMA’s half buffers.
 *
 *   Card Address and IRQ
 *   --------------------
 *   We will use the default bus address of 0x330 and IRQ 5. Change in
 *   bus address should also be reflected in /var/sysgen/system/rap.sm
 *   file.  Changes in IRQ should be reflected in the source code and
 *   the program must be recomplied.
 *
 *   ISSUES:
 *   -------
 *   1. The DMA processing and transfer of data from/to user’s buffer
 *      are independent of each other. When we are servicing the
 *      one half of the dma buffer that just been transfered, there is
 *      no guarantee that we can fill that half of the buffer BEFORE
 *      dma is done with the other half. In this case, dma plays the
 *      fist half of buffer WHILE we are writing into it.
 *
 *   2. Currently eisa_dma_disable() routine does not actually
 *      releases the Dma channels. This is the reason why we access
 *      the Dma channel table (e_ch[]) ourselves and release the
 *      channel.
 *
 *   3. Somehow because of number 2, the Play program cannot be
 *      stopped with a Ctrl-C. In Play program this signal is
 *      explicitly ignored. Trapping a Ctrl-C causes a kernel panic.
 *      Once we have a workable eisa_dma_disable(), this problem will
 *      be resolved.
 *
 *   TECHNICAL REFERENCES:
 *   ---------------------
 *   Roland RAP-10 Technical Reference and Programmer’s Guide, Ver. 1.1
 *   IRIX Device Driver Programming Guide
 *   IRIX Device Driver Reference Pages.
 *   Intel 82357 Preliminary Reference, Section: 3.7.8 Mode Register (pp: 223)
 *
 ******************************************************************************
 ***                                                                        ***
 ***   Copyright 1994, Silicon Graphics Inc., Mountain View, CA.            ***
 ***                                                                        ***
 ******************************************************************************
 */
#include “sys/types.h”
#include “sys/file.h”
#include “sys/errno.h”
#include “sys/open.h”
#include “sys/conf.h”
#include “sys/cmn_err.h”
#include “sys/debug.h”
#include “sys/param.h”
#include “sys/edt.h”
#include “sys/pio.h”
#include “sys/uio.h”
#include “sys/proc.h”
#include “sys/user.h”
#include “sys/eisa.h”
#include “sys/sema.h”
#include “sys/buf.h”
#include “sys/cred.h”
#include “sys/kmem.h”
#include “sys/ddi.h”
#include “./rap.h”
/*
 *  Macros to Read/Write 8 and 16-bit values from an address
 */
#define OUTB(addr, b)  ( *(volatile uchar_t *)(addr) = (b) )
```

```
#define INPB(addr)     ( *(volatile uchar_t *)(addr) )
#define OUTW(addr, w)  ( *(volatile ushort_t *)(addr) = (w) )
#define INPW(addr)     ( *(volatile ushort_t *)(addr) )
/*
 *  Raising and lowering CPU interrupt
 */
#define LOCK()     spl5()
#define UNLOCK(s)  splx(s)
#define FROM_INTR  1
#define FROM_USR   0
#define User_pid   u.u_procp->p_pid
/*
 *   IRQ and DMA channels we need.
 *
 */
#define IRQ_MASK    0x0020
#define DMAC_CH5        0x20     /*  DMA Channel 5   */
#define DMAC_CH6        0x40     /*  DMA Channel 6   */
/*=======================================*
 *       MIDI and RAP Registers      *
 *=======================================*
 *
 *  The following is a description of RAP-10 registers. The same
 *  names used throughout this program. Some of these registers are
 *  8-bit and some are 16-bit long.
 *
 *      mdrd:    MIDI Receive Data
 *      mdtd:    MIDI Transmit Data
 *      mdst:    MIDI Status
 *      mdcm:    MIDI Command
 *      pwmd:    Pulse Width Modulation Data
 *      timm:    Timer MSB data
 *      gpcm:    GPCC Command
 *      dtci:    DMA Transfer Count Buffer Interrupt Status
 *      adcm:    GPCC Analog to Digital Command
 *      dacm:    D/A Command and DMA Transfer Configuration
 *      gpis:    GPCC Interrupt Status
 *      gpdi:    GPCC DMA/Interrupt Enable
 *      gpst:    GPCC Status
 *      dad0:    Digital to Analog Data Channel 0
 *      addt:    A/D Data Transfer
 *      dad1:    Digital to Analog Data Channel 1
 *      timd:    Timer Data
 *      cmp0:    Compare Register Channel 0
 *      dtcd:    DMA Transfer Count Data
 *      cmp1:    Compare Register Channel 1
 *
 *    These defines indicate the offsets of the above registers
 *    from the Drive’s base address:
 */
#define MDRD    0x0
#define MDTD    0x0
#define MDST    0x1
#define MDCM    0x1
#define PWMD    0x2
#define TIMM    0x3
#define GPCM    0x3
#define DTCI    0x4
#define ADCM    0x4
#define DACM    0x5
#define GPIS    0x6
#define GPDI    0x6
#define GPST    0x8
#define DAD0    0x8
#define ADDT    0xa
#define DAD1    0xa
#define TIMD    0xc
#define CMP0    0xc
#define DTCD    0xe
#define CMP1    0xe

typedef struct rapReg {
    uchar_t  mdrd;
    uchar_t  mdtd;
    uchar_t  mdst;
    uchar_t  mdcm;
    uchar_t  pwmd;
    uchar_t  timm;
    uchar_t  gpcm;
    uchar_t  dtci;
    uchar_t  adcm;
    uchar_t  dacm;
    ushort_t gpis;
    ushort_t gpdi;
    ushort_t gpst;
    ushort_t dad0;
    ushort_t addt;
    ushort_t dad1;
    ushort_t timd;
    ushort_t cmp0;
    ushort_t dtcd;
    ushort_t cmp1;
} rapReg_t;
/*============================================================*
 *      dtct  (DMA Transfer Count)                            *
 *============================================================*/
#define DTCD_DRQ0  0x00FF   /*  DRQ 0 bits (0-7)  */
#define DTCD_DRQ1  0xFF00   /*  DRQ 1 bits (8-15) */
/*============================================================*
 *      gpst  (GPCC Status)                                   *
 *============================================================*/
#define GPST_PWM2 0x0800  /* PWM2 Busy (0=Write Enable, 1=Busy) */
#define GPST_PWM1 0x0400  /* PWM1 Busy (0=Write Enable, 1=Busy) */
#define GPST_PWM0 0x0200  /* PWM0 Busy (0=Write Enable, 1=Busy) */
#define GPST_EPB  0x0100  /* EP Convertor Busy (0=Write Enable, 1=Busy) */
#define GPST_GP1  0x0080  /* GP-chip, Ch 1 Acess (1 = Access)   */
#define GPST_GP0  0x0040  /* GP-chip, Ch 0 Acess (1 = Access)   */
#define GPST_MTE  0x0020  /* MIDI Tx Enable (0=Tx_Fifo buff full) */
#define GPST_ORE  0x0010  /* MIDI Overrun Error (1 = error) */
#define GPST_FE   0x0008  /* MIDI Framing Error (1 = error) */
#define GPST_ADE  0x0004  /* A/D Error (1 = error) */
#define GPST_DE1  0x0002  /* D/A Ch 1 Write Error (1 = error) */
#define GPST_DE0  0x0001  /* D/A Ch 0 Write Error (1 = error) */
/*============================================================*
 *        gpdi  (GPCC DMA/Interrupt Enable (pp: 4-18)      *
 *============================================================*/
#define GPDI_ITC  0x8000  /* DMA Transfer Cnt Match (0=Disable) */
#define GPDI_DC2  0x4000  /* DMA Chann. Assignment, bit2 (pp:4-18) */
#define GPDI_DC1  0x2000  /* DMA Chann. Assignment, bit1 (pp:4-18) */
#define GPDI_DC0  0x1000  /* DMA Chann. Assignment, bit0 (pp:4-18) */
#define GPDI_DT1  0x0800  /* DMA Trans. Mode, bit:1 (pp: 4-18) */
#define GPDI_DT0  0x0400  /* DMA Trans. Mode, bit:0 (pp: 4-18) */
#define GPDI_OVF  0x0200  /* Free Run.Cntr (FCR) Ov.Flow (0=Disable)*/
#define GPDI_TC1  0x0100  /* Timer 1 Compare Match (0=Disable) */
#define GPDI_TC0  0x0080  /* Timer 0 Compare Match (0=Disable) */
#define GPDI_RXD  0x0040  /* MIDI Data Read Request (0=Disable) */
#define GPDI_TXD  0x0020  /* MIDI Tx_fifo Buf Empty (0=Disable) */
#define GPDI_ADD  0x0010  /* A/D Data Ready (0=Disable)  */
#define GPDI_DN1  0x0008  /* D/A Ch1 Note ON Ready (0=Disable) */
#define GPDI_DN0  0x0004  /* D/A Ch0 Note ON Ready (0=Disable) */
#define GPDI_DQ1  0x0002  /* D/A Ch1 Data Request  (0=Disable) */
#define GPDI_DQ0  0x0001  /* D/A Ch0 Data Request  (0=Disable) */
/*============================================================*
 *         gpis  (GPCC Interrupt Status .. pp: 4-16)       *
 *============================================================*/
```

```
#define GPIS_ITC  0x8000  /*  DMA Transfer Count Match   */
#define GPIS_JSD  0x0400  /*  Joystick Data Ready */
#define GPIS_OVF  0x0200  /*  Free Running Countr Overflow */
#define GPIS_TC1  0x0100  /*  Timer1 Compare Match */
#define GPIS_TC0  0x0080  /*  Timer0 Compare Match */
#define GPIS_RXD  0x0040  /*  MIDI Data Read Request */
#define GPIS_TXD  0x0020  /*  MIDI Tx_fifo Buf. Empty */
#define GPIS_ADD  0x0010  /*  A/D Data Ready */
#define GPIS_DN1  0x0008  /*  D/A Ch1 Note ON Ready */
#define GPIS_DN0  0x0004  /*  D/A Ch0 Note ON Ready */
#define GPIS_DQ1  0x0002  /*  D/A Ch1 Data Request */
#define GPIS_DQ0  0x0001  /*  D/A Ch0 Data Request */
/*==================================================================*
 *         dacm (Digital To Analogue Cmd and DMA Transfer Config)    *
 *==================================================================*/
#define DACM_SCC   0x80  /* DMA Size Cmp. Cnt (0=in Sample, 1=in Bytes)*/
#define DACM_TS2   0x40  /* DMA Trnsfr Size, bit 2 (pp: 4-14) */
#define DACM_TS1   0x20  /* DMA Trnsfr Size, bit 1 (pp: 4-14) */
#define DACM_TS0   0x10  /* DMA Trnsfr Size, bit 0 (pp: 4-14) */
#define DACM_DL1   0x08  /* Ch1 DA Data Len (0=8 bit, 1=17 bit) */
#define DACM_DL0   0x04  /* Ch0 DA Data Len (0=8 bit, 1=17 bit) */
#define DACM_DS1   0x02  /* Ch1 DA Convrsion (0=Stop, 1=Start) */
#define DACM_DS0   0x01  /* Ch0 DA Convrsion (0=Stop, 1=Start) */
/*=====================================================*
 *             adcm (  GPCC AD Command )               *
 *=====================================================*/
#define  ADCM_MON    0x40  /* Monitor MIC (0=Monitor Off) */
#define  ADCM_GIN    0x20  /* Gain Input (0=Line, 1=Mic) */
#define  ADCM_AF1    0x10  /* Analog Freq Selection bit 1 (pp: 4-13) */
#define  ADCM_AF0    0x08  /* Analog Freq Selection bit 0 (pp: 4-13) */
#define  ADCM_ADL    0x04  /* Analog Data Length (0=8, 1=16) */
#define  ADCM_ADM    0x02  /* Analog Data Conv. Mode (0=Mono,1=Stereo) */
#define  ADCM_ADS    0x01  /* Analog Data Conv. Start(0=Stop,1=Start) */
/*=====================================================*
 *            dtci ( DMA Trans.Count Buf Intr. Stat    *
 *=====================================================*/
#define DTCI_BF1       0x08  /* DMA DRQ1 buff full (1 = full) */
#define DTCI_BH1       0x04  /* DMA DRQ1 buff half (1 = full) */
#define DTCI_BF0       0x02  /* DMA DRQ0 buff full (1 = full) */
#define DTCI_BH0       0x01  /* DMA DRQ0 buff half (1 = full) */
/*========================================*
 *             gpcm  ( GPCC Command )     *
 *========================================*/
#define GPCM_RST   0x80    /*   Reset bit */
#define GPCM_PWM2  0x10    /*   Select PWM channel 2 */
```

```
#define GPCM_PWM1  0x08    /*   Select PWM channel 1 */
#define GPCM_PWM0  0x04    /*   Select PWM channel 0 */
#define GPCM_FRCM  0x02    /*   Free Run. Counter (1=Start) */
#define GPCM_MTT   0x01    /*   MIDI Timed Trans */
                           /*   ( 1 = Timer INT enabled )   */
/*=====================================*
 *       timm  (Timer MSB data)        *
 *=====================================*/
#define TIMM_FRC   0x04    /*   Free Running Counter Bit 16 */
#define TIMM_CR1   0x02    /*   Compare Reg 1 Bit 16 */
#define TIMM_CR0   0x01    /*   Compare Reg 0 Bit 16 */
/*==================================*
 *      mdcm (MIDI Command)         *
 *==================================*/
#define MDCM_UART      0x3f   /*   UART mode */
#define MDCM_MPU       0xff   /*   MPU Reset */
#define MDCM_VERSION   0xac   /*   Version  */
#define MDCM_REVISION  0xad   /*   Revision */
/*==================================*
 *       mdst (MIDI Status)         *
 *==================================*/
#define  MDST_DSR  0x80 /*  DSR = 0 if ready */
#define  MDST_DDR  0x40 /*  DDR = 0 if ready */
/*===================================*
 *      RAP Card Info                *
 *===================================*
 *
 *   These are the information regarding the RAP Card.
 *   The info being tracked are:
 *
 *  ci_state:   Our state (Installed, Opened, Playing, Recording)
 *  ci_pid:     PID of process opened us.
 *  ci_addr[]:  EISA  Addresses
 *  ci_irq:     EISA Interrupt number we use
 *  ci_ctl:     Controller number we save from edt struct
 *  ci_adap:    Adaptor number we save from edt struct.
 *  ci_dmaCh6:  DMA Channel 6
 *  ci_dmaCh5:  DMA Channel 5
 *  ci_dmaBuf6: EISA DMA Buffer struct for Channel 6
 *  ci_dmaBuf5: EISA DMA Buffer struct for Channel 5
 *  ci_dmaCb6:  EISA DMA Control Block for Channel 6
 *  ci_dmaCb5:  EISA DMA Control Block for Channel 5
 *  di_state:   DMA buffers state (Idle, Progress)
 *  di_idx:     Current rwQue[] entry being used.
 *  di_ptr:     Address in rwQue buffer
 *  di_which:   Which half of DMA buffer (0=1st half, 1=2nd Half)
 *  di_bh:      Total DMA Buffer Half (BH) Interrupt received.
 *  di_bf:      Total DMA Buffer Full (BF) Interrupt received.
 *  ri_state:   State of Circular buffer (Wanted_Empty, etc.)
 *  ri_free:    Total Free entries in rwQue[]
 *  ri_full:    Total Full entries in rwQue[]
 *  ri_idx:     Current rwBuf for Read/Write
 *  ri_tout;    =1 if Timed out on read/write
 *  ri_note;    number of Note_On received
 *  ri_ptr:     Pointer in current rwBuf
 */
typedef  struct eisa_dma_buf   dmaBuf_t;
typedef  struct eisa_dma_cb    dmaCb_t;
typedef struct cardInfo_s {
    /*   Card Installation Info */
    ushort_t   ci_state;
    pid_t      ci_pid;
    caddr_t    ci_addr[NBASE];
    int    ci_irq;
    int    ci_ctl;
    int    ci_adap;
    int    ci_dmaCh6;
    int    ci_dmaCh5;
    dmaBuf_t   *ci_dmaBuf6;
    dmaBuf_t   *ci_dmaBuf5;
    dmaCb_t    *ci_dmaCb6;
    dmaCb_t    *ci_dmaCb5;
    /*   DMA Buffer Information data */
    uchar_t   di_state;
    short     di_idx;
    uchar_t   di_which;
    caddr_t   di_ptr;
    uchar_t   di_bh;
    uchar_t   di_bf;
    /*   Circular buffer Information data */
    uchar_t   ri_state;
    short     ri_free;
    short     ri_full;
    short     ri_idx;
    uchar_t   ri_tout;
    uchar_t   ri_note;
    caddr_t   ri_ptr;
} cardInfo_t;
/*   ci_state  values   */
#define  CARD_INSTALLED    0x0001
```

```
#define  CARD_STEREO       0x0002
#define  CARD_OPENED       0x0004
#define  CARD_PLAYING      0x0010
#define  CARD_RECORDING    0x0020
/*   di_state values    */
#define  DI_DMA_IDLE       0x00
#define  DI_DMA_PLAYING    0x01
#define  DI_DMA_RECORDING  0x02
#define  DI_DMA_END_PLAY   0x04
#define  DI_DMA_END_RECORD 0x08
/*   ri_state values   */
#define RI_WANTED_EMPTY    0x01
/*====================================*
 *      Read/Write Circular Buffers   *
 *====================================*
 *  This is the description of our circular buffers used
 *  to store D/A and A/D values. D/A values are stored from
 *  user's buffer and then moved to DMA buffers. A/D data is
 *  moved from DMA buffers to these buffers and then moved
 *  to user's buffer. The fields are as follow:
 *    rw_state:    buffer state (Empty, Busy, Full)
 *    rw_idx:      Index of this buffer in rwQue[];
 *    rw_count:    Total bytes in the buffer
 *    rw_buf[]:    The buffer itself.
 *      RW_MIN_FULL:  We will start a D/A DMA when we have this many
 *          full buffer on hand. This is done so that we can
 *          provide enough full buffers for DMA to process.
 */
#define  RW_BUF_SIZE    8192
#define  RW_BUF_COUNT   20
#define  RW_MIN_FULL    1
#define  RW_TIMEOUT     1600
typedef struct rwBuf_s {
    uchar_t     rw_state;
    short       rw_idx;
    int     rw_count;
    uchar_t     rw_buf[RW_BUF_SIZE];
} rwBuf_t;
/*   rw_state  values   */
#define RW_EMPTY        0x00  /*  used as parameter only  */
#define RW_FULL         0x01
#define RW_WANTED_FULL  0x02
#define RW_WANTED_EMPTY 0x04
/*==================================*
 *         Global values            *
 *=================================*/
#define   DMA_BUF_SIZE        8192
#define   DMA_HALF_SIZE       4096
int                   rapdevflag = 0;
static cardInfo_t     cardInfo;
static caddr_t        dmaRight;
static caddr_t        dmaLeft;
static paddr_t        dmaRightPhys;
static paddr_t        dmaLeftPhys;
static rwBuf_t        rwQue[RW_BUF_COUNT];
static caddr_t        eisa_addr;
/*
 *  Eisam Dma Channel semaphores..shoule be removed when
 *  proper way of releasing channels found
 */
extern struct eisa_ch_state {
    sema_t chan_sem;               /* inuse semaphore for each channel */
    sema_t dma_sem;                /* dma completion semaphore */
    struct eisa_dma_buf *cur_buf;  /* current eisa_dma_buf being dma'ed */
    struct eisa_dma_cb *cur_cb;    /* ptr to current command block */
    int    count;
} e_ch[];
/*=========================================*
 *        Driver Entry routines Data       *
 *=========================================*/
int    rapopen  ( dev_t *, int, int, cred_t  * );
int    rapread ( dev_t, uio_t *, cred_t * );
int    rapwrite ( dev_t, uio_t *, cred_t * );
int    rapclose ( dev_t, int, int, cred_t  * );
void   rapedtinit ( struct edt  * );
void   rapintr ( int );
int    rapioctl (dev_t, int, void *, int, cred_t *, int *);
/*=======================================*
 *  Misc and Internal routines           *
 *=======================================*/
static void   rapDisInt (cardInfo_t  *);
static int    rapGetDma( dmaBuf_t  **,  dmaCb_t **, int );
static int    rapClose(uchar_t);
static short  rapGetNextEmpty (short, uchar_t);
static short  rapGetNextFull (short, uchar_t);
static void   rapPrepEisa( dmaBuf_t *, dmaCb_t *, uchar_t, paddr_t);
static int    rapStart(uchar_t);
static void   rapStop(uchar_t);
static void   rapStartDA();
static void   rapStartAD();
static void   rapBufToDma( int );
static void   rapDmaToBuf( int );
static void   rapMarkBuf(rwBuf_t *, cardInfo_t *, uchar_t);
static int    rapKernMem(uchar_t);
static void   rapSetAutoInit(cardInfo_t *, uchar_t);
static void   rapTimeOut( void *);
static void   rapNoteOn(cardInfo_t *, ushort_t );
static void   rapNoteOff(cardInfo_t *);
static void   rapZeroDma(cardInfo_t *, int);
static void   rapReleaseDma (cardInfo_t *);
/***********************************************************************
 *    r a p e d t i n i t
 ***********************************************************************
 *   Name:      rapedtint
 *   Purpose:   Initializes the driver. Called once for each controller.
 *    Called only once.
 *   Returns:   None.
 ***********************************************************************/
void
rapedtinit ( struct edt *e )
{
    int      ctl, iospace, dmac, eirq;
    cardInfo_t   *ci;
    piomap_t      *pmap;
    iospace_t     eisa_io;

    ci = &cardInfo;
    cmn_err (CE_NOTE, “rapedtinit: Installing RAP board.”);
    bzero ((void *)ci, sizeof(cardInfo_t) );
    dmaRight = dmaLeft = (caddr_t)NULL;
    ci->ci_ctl = e->e_ctlr;
    ci->ci_adap = e->e_adap;
    /*
     *    Get the base address of Eisa bus (for rapSetAutoInit)
     */
    bzero (&eisa_io, sizeof(iospace_t));
    eisa_io.ios_iopaddr = 0;
    eisa_io.ios_size    = 1000;
    pmap = pio_mapalloc (e->e_bus_type, 0, &eisa_io, PIOMAP_FIXED, “eisa”);
    if ( pmap == (piomap_t *)NULL ) {
        cmn_err (CE_WARN, “rapedtinit: Cannot get Eisa bus address”);
        return;
    }
    eisa_addr = pio_mapaddr (pmap, eisa_io.ios_iopaddr);
    #ifdef DEBUG
```

```
cmn_err (CE_NOTE, “rapedtinit: Eisa base address = %x”, eisa_addr);
#endif
/*==================================================*
 *   map EISA IO/Memory addresses for RAP-10 card    *
 *==================================================*/
for ( iospace = 0; iospace < NBASE; iospace++ ) {
    /*  any address to map ?    */
    if ( !e->e_space[iospace].ios_iopaddr )
        continue;
    pmap = pio_mapalloc ( e->e_bus_type, e->e_adap,
            &e->e_space[iospace],
            PIOMAP_FIXED, “rap10” );
    ci->ci_addr[iospace] = pio_mapaddr ( pmap,
                 e->e_space[iospace].ios_iopaddr );
}
/*  is Card still there  ? */
if ( badaddr(ci->ci_addr[0], 1) ) {
    cmn_err (CE_WARN, “rapedtinit: RAP board not installed.”);
    return;
}
#ifdef DEBUG
cmn_err (CE_NOTE, “rapedtinit: First Load..allocating IRQ”);
#endif
eirq = eisa_ivec_alloc( e->e_adap, IRQ_MASK, EISA_EDGE_IRQ );
if ( eirq < 0 ) {
    cmn_err (CE_WARN,
    “rapedtinit: Could not allocate IRQ for RAP card.”);
    return;
}
/*    set Interrupt handler     */
#ifdef DEBUG
cmn_err (CE_NOTE, “rapedtinit: Setting Interrupt Handler for IRQ %d”,
          eirq);
#endif
if ( eisa_ivec_set(e->e_adap, eirq, rapintr, e->e_ctlr) == -1 ) {
    cmn_err (CE_NOTE,
    “rapedtinit: Could not set Interrupt handler for Irq %d”, eirq);
    ci->ci_state = 0;
    return;
}
ci->ci_irq = eirq;
/*=====================================*
 *  DMA Channels Allocation            *
 *=====================================*/
/*   DMA channel 5 */
```

```
    dmac = eisa_dmachan_alloc ( e->e_adap, DMAC_CH5 );
    if ( dmac < 0 ) {
        cmn_err (CE_WARN,
             "rapedtinit: Could not allocate DMA Channel 5.");
        return;
    }
    ci->ci_dmaCh5 = dmac;
    /*   DMA channel 6  */
    dmac = eisa_dmachan_alloc ( e->e_adap, DMAC_CH6 );
    if ( dmac < 0 ) {
        cmn_err (CE_WARN,
            "rapedtinit: Could not allocate DMA Chann 6.");
        cmn_err (CE_WARN,
            "rapedtinit: RAP is initialized as Mono.");
    }
    else {
        ci->ci_dmaCh6 = dmac;
        ci->ci_state |= CARD_STEREO;
    }
    /*==============================*
     *      DMA Buffer allocation   *
     *==============================*/
    if ( rapKernMem (1) ) {
        cmn_err (CE_WARN, "rapedtinit: Did not install RAP-10.");
        return;
    }
    ci->ci_state |= CARD_INSTALLED;
#ifdef DEBUG
    cmn_err (CE_NOTE, "rapedtinit: RAP installed, Addr: %x, Irq: %d.",
            ci->ci_addr[0], ci->ci_irq );
    cmn_err (CE_NOTE, "rapedtinit: Init as %s, Dma 1 = %d, Dma 0 = %d",
            (ci->ci_state & CARD_STEREO ? "Stereo":"Mono"),
            ci->ci_dmaCh5, ci->ci_dmaCh6);
#endif
    return;
}  /***   End rapedtinit    ***/
/*************************************************************************
 *     r a p o p e n
 *************************************************************************
 *  Name:      rapopen
 *  Purpose:   Opens the RAP board and initializes necessary data
 *  Returns:   0 = Success, or appropriate error number.
 *************************************************************************/
int
rapopen ( dev_t  *dev, int oflag, int otyp, cred_t  *cred)
```

```
{
    register int    i;
    cardInfo_t      *ci;
    rwBuf_t         *rw;
    dmaBuf_t    *dmaB;
    dmaCb_t         *dmaC;
    ci = &cardInfo;
#ifdef DEBUG
    cmn_err (CE_NOTE, "rapopen: Opening, Addr = %x, ci_state = %x",
         ci->ci_addr[0], ci->ci_state );
#endif
    /*
     *  No card is installed or card is already opened
     */
    if ( !(ci->ci_state & CARD_INSTALLED) )
        return (ENODEV);
    if ( ci->ci_state & CARD_OPENED )
        return (EBUSY);
    /*   Allocate DMA Buf and Cb for Channel 5  */
    if ( ci->ci_dmaBuf5 == (dmaBuf_t *)NULL ) {
        if ( rapGetDma(&dmaB, &dmaC, ci->ci_dmaCh5) ) {
            cmn_err (CE_WARN,"rapopen: Could not allocate DMA Buf 5.");
            return (ENOMEM);
        }
        ci->ci_dmaBuf5 = dmaB;
        ci->ci_dmaCb5  = dmaC;
    }
    /*   if in stereo, do the same for Channel 6  */
    if ( ci->ci_state & CARD_STEREO ) {
        if ( rapGetDma(&dmaB, &dmaC, ci->ci_dmaCh6) ) {
            cmn_err (CE_WARN,
                 "rapopen: Could not allocate DMA Buf 6.");
            return (ENOMEM);
        }
        ci->ci_dmaBuf6 = dmaB;
        ci->ci_dmaCb6  = dmaC;
    }
    /*   Initialize Card Info structure  */
    ci->ri_idx   = 0;
    ci->di_idx   = 0;
    ci->ri_state = 0;
    ci->di_state = 0;
    ci->di_ptr   = 0;
    ci->ri_ptr   = 0;
    ci->ri_free  = RW_BUF_COUNT;
    ci->ri_full  = 0;
    ci->ci_state &= ~(CARD_PLAYING | CARD_RECORDING );
    ci->ci_state |= CARD_OPENED;
    ci->ci_pid    = User_pid;
    /*   Initialize Circular Buffers   */
    for ( i = 0; i < RW_BUF_COUNT; i++ ) {
        rw = &rwQue[i];
        rw->rw_count = 0;
        rw->rw_state = 0;
        rw->rw_idx   = i;
        bzero (rw->rw_buf, RW_BUF_SIZE);
    }
    rapDisInt(ci);
    #ifdef DEBUG
    cmn_err (CE_NOTE, "rapopen: Opened succesfully");
    #endif
    return(0);
}  /***  End rapopen  ***/
/**************************************************************************
 *       r a p w r i t e
 **************************************************************************
 *  Name:       rapwrite
 *  Purpose:    Write entry routine. This routine will transfer user's
 *      data to current or an empty entry in rwQue[] and starts
 *      DMA if none is going.
 *  Returns:    0 = Success, or errno
 **************************************************************************/
int
rapwrite (dev_t  dev, uio_t  *uio, cred_t  *cred)
{
    cardInfo_t    *ci;
    rwBuf_t       *rw;
    toid_t        to_id;
    int       avail, size, totBytes, err, s;
    ci = &cardInfo;
    /*========================*
     *  Error Checking     *
     *========================*/
    /*  no card is installed  */
    if ( !(ci->ci_state & CARD_INSTALLED) )
        return (ENODEV);
    /*  card is not opened */
    if ( !(ci->ci_state & CARD_OPENED) )
        return (EACCES);
    /*  we are not the owner  */
```

```
if ( ci->ci_pid != User_pid )
    return (EACCES);
/*  is busy recording  */
if ( ci->ci_state & CARD_RECORDING )
    return (EACCES);
ci->ci_state |= CARD_PLAYING;
rw = &rwQue[ci->ri_idx];
#ifdef DEBUG
cmn_err (CE_NOTE,
    "rapwrite: %d bytes, buf = %d, rw_count = %d, free = %d, full = %d",
    uio->uio_resid, ci->ri_idx, rw->rw_count, ci->ri_free, ci->ri_full);
#endif
/*  if it is full, wait till it is Empty   */
s = LOCK();
if ( rw->rw_state & RW_FULL ) {
    ci->ri_ptr = NULL;
    ci->ri_tout = 0;
    to_id = itimeout (rapTimeOut, rw, RW_TIMEOUT, plbase, 0, 0, 0);
    while ( (rw->rw_state & RW_FULL) && !ci->ri_tout ) {
        #ifdef DEBUG
        cmn_err (CE_NOTE, "rapwrite: waiting for buf %d to be Empty",
                        rw->rw_idx );
        #endif
        rw->rw_state |= RW_WANTED_EMPTY;
        if ( sleep (rw, PUSER | PCATCH) ) {
            untimeout(to_id);
            #ifdef DEBUG
            cmn_err (CE_NOTE, "rapwrite: Interrupted");
            #endif
            rw->rw_state &= ~RW_WANTED_EMPTY;
            UNLOCK(s);
            return (EINTR);
        }
    } /* while */
    untimeout(to_id);
    /*   we timed out ..couldn't get the buffer  */
    if ( ci->ri_tout ) {
        #ifdef DEBUG
        cmn_err (CE_NOTE, "rapwrite: Timed out");
        #endif
        rw->rw_state &= ~RW_WANTED_EMPTY;
        UNLOCK(s);
        return (EIO);
    }
} /*  if (rw->rw_state & RW_FULL */
UNLOCK(s);
/*  adjuest the read/write address if necessary */
if ( ci->ri_ptr == NULL )
    ci->ri_ptr = rw->rw_buf;
totBytes = uio->uio_resid;
while ( totBytes > 0 )  {
    avail = RW_BUF_SIZE - rw->rw_count;
    /*  if this buffer is full, get next buffer */
    if ( avail <= 0 ) {
        #ifdef DEBUG
        cmn_err (CE_NOTE,
        “rapwrite: Buffer %d is Full now, rw_count = %d”,
        rw->rw_idx, rw->rw_count);
        #endif
        s = LOCK();
        rapMarkBuf(rw, ci, RW_FULL);
        /* wake anyone wanted this buffer full  */
        if ( rw->rw_state & RW_WANTED_FULL ) {
            #ifdef DEBUG
            cmn_err (CE_NOTE,”rapwrite: Buffer %d is Wanted_Full”,
                                rw->rw_idx );
            #endif
            rw->rw_state &= ~RW_WANTED_FULL;
            wakeup(rw);
        }
        /*
         *   start DMA if none is going and we filled the
         *   entire buffers.
         */
        if ( (ci->di_state == DI_DMA_IDLE) &&
             (rw->rw_idx >= RW_MIN_FULL )  ) {
            #ifdef DEBUG
            cmn_err (CE_NOTE,”rapwrite: Starting Play Dma”);
            #endif
            err = rapStart(DI_DMA_PLAYING);
            if ( err )   {
                cmn_err (CE_WARN,
                    “rapwrite: Could not start playing error %d”,err );
                UNLOCK(s);
                return(err);
            }
        }
        /*    get next empty buffer  */
        ci->ri_idx = rapGetNextEmpty(ci->ri_idx, FROM_USR);
        rw = &rwQue[ci->ri_idx];
```

```
            ci->ri_ptr = rw->rw_buf;
            UNLOCK(s);
            continue;
        }
        /*  start filling this buffer   */
        size = (totBytes > avail ? avail: totBytes);
        err = uiomove (ci->ri_ptr, size, UIO_WRITE, uio);
        if ( err ) {
            cmn_err (CE_NOTE, “rapwrite: uiomov error %d”, err);
            return(err);
        }
        rw->rw_count += size;
        ci->ri_ptr   += size;
        totBytes = uio->uio_resid;
        #ifdef DEBUG
        cmn_err (CE_NOTE,
            “rapwrite: Wrote  %d to Buffer %d, Left = %d, rw_count = %d”,
            size, rw->rw_idx, totBytes, rw->rw_count );
        #endif
    }
    return (0);
} /*** end rapwrite ***/
/*************************************************************************
 *      r a p r e a d
 *************************************************************************
 *
 *  Name:      rapread
 *
 *  Purpose:   Reads data from rwQue[] into user’s buffer.
 *         This routine waits for current DMA operation to end
 *         and then starts a A/D Dma (recording). If A/D is already
 *         going then it simply moves data from current Full buffer
 *         into user’s buffer. If buffer is not full, it waits for
 *         it to get full.
 *
 *  Returns:   0 = Success, or errno.
 *
 *************************************************************************/
int
rapread (dev_t  dev, uio_t *uio, cred_t *cred)
{
    cardInfo_t      *ci;
    rwBuf_t         *rw;
    toid_t          to_id;
    int             avail, size, totBytes, err, s;
```

```
ci = &cardInfo;
/*===============================*
 *      Error Checking           *
 *===============================*/
/*  card is not installed */
if ( !(ci->ci_state & CARD_INSTALLED) )
    return (ENODEV);
/*  card is not opened  */
if ( !(ci->ci_state & CARD_OPENED) )
    return (EACCES);
/*  we do not own the card */
if ( ci->ci_pid != User_pid )
    return (EACCES);
/*  card is in middle of a Play operation  */
if ( ci->ci_state & CARD_PLAYING )
    return (EIO);
ci->ci_state |= CARD_RECORDING;
/*   start a A/D Dma if none is going on  */
if ( ci->di_state == DI_DMA_IDLE ) {
    #ifdef DEBUG
    cmn_err (CE_NOTE, “rapread: Idle DMA. Starting one”);
    #endif
    if ( rapStart(DI_DMA_RECORDING) ) {
        #ifdef DEBUG
        cmn_err (CE_NOTE, “rapread: Could not start A/D”);
        #endif
        ci->ci_state &= ~CARD_RECORDING;
        UNLOCK(s);
        return (EIO);
    }
}
/*
 *  get the buffer we should be using and
 *  wait for it to become Full
 */
rw = &rwQue[ci->ri_idx];
#ifdef DEBUG
cmn_err (CE_NOTE,
    “rapread: %d bytes, buf = %d, rw_count = %d, free = %d, full = %d”,
    uio->uio_resid, ci->ri_idx, rw->rw_count, ci->ri_free, ci->ri_full);
#endif
s = LOCK();
if ( !(rw->rw_state & RW_FULL) ) {
    ci->ri_ptr = NULL;
    ci->ri_tout = 0;
```

```
        to_id = itimeout (rapTimeOut, rw, RW_TIMEOUT, plbase, 0, 0, 0);
        while ( !(rw->rw_state & RW_FULL) && !ci->ri_tout ) {
            #ifdef DEBUG
            cmn_err (CE_NOTE, “rapread: wating for buf %d to become Full”,
            rw->rw_idx );
            #endif
            rw->rw_state |= RW_WANTED_FULL;
            if ( sleep (rw, PUSER | PCATCH) ) {
                untimeout (to_id);
                #ifdef DEBUG
                cmn_err (CE_NOTE, “rapread: Interrupted”);
                #endif
                rw->rw_state &= ~RW_WANTED_FULL;
                UNLOCK(s);
                return(EINTR);
            }
        } /*  while  */
        untimeout (to_id);
        if ( ci->ri_tout ) {
            #ifdef DEBUG
            cmn_err (CE_NOTE, “rapread: Timed out”);
            #endif
            rw->rw_state &= ~RW_WANTED_FULL;
            UNLOCK(s);
            return (EIO);
        }
    }  /*  if !rw->rw_state & RW_FULL  */
    UNLOCK(s);
    /*  adjust read/write pointer if necessary   */
    if ( ci->ri_ptr == NULL )
        ci->ri_ptr = rw->rw_buf;
    /*===================================*
     *     Actual Read (Data movement)   *
     *===================================*/
    totBytes = uio->uio_resid;
    while ( totBytes > 0 )  {
        avail = rw->rw_count;
        /*  if this buffer is Empty, get next Full buffer */
        if ( avail <= 0 ) {
            #ifdef DEBUG
            cmn_err (CE_NOTE,
                “rapread: Buffer %d is Empty now, rw_count = %d”,
                    rw->rw_idx, rw->rw_count );
            #endif
            s = LOCK();
            rapMarkBuf(rw, ci, RW_EMPTY);
            /* wake anyone wanted this buffer Empty */
            if ( rw->rw_state & RW_WANTED_EMPTY ) {
                #ifdef DEBUG
                cmn_err (CE_NOTE,"rapread: Buffer %d is Wanted_Empty",
                        rw->rw_idx );
                #endif
                rw->rw_state &= ~RW_WANTED_FULL;
                wakeup(rw);
            }
            /*    get next Full buffer  */
            ci->ri_idx = rapGetNextFull(ci->ri_idx, FROM_USR);
            rw = &rwQue[ci->ri_idx];
            ci->ri_ptr = rw->rw_buf;
            UNLOCK(s);
            continue;
        }
        /*  start filling this buffer   */
        size = (totBytes > avail ? avail: totBytes);
        err = uiomove (ci->ri_ptr, size, UIO_READ, uio);
        if ( err ) {
            cmn_err (CE_PANIC, "rapread: uiomov error %d", err);
            return(err);
        }
        rw->rw_count -= size;
        ci->ri_ptr   += size;
        totBytes = uio->uio_resid;
        #ifdef DEBUG
        cmn_err (CE_NOTE,
            "rapread: Read %d, Buffer %d, Left = %d, rw_count = %d",
            size, rw->rw_idx, totBytes, rw->rw_count );
        #endif
    }
    return (0);
}  /***   End rapread    ***/
/**************************************************************************
 *         r a p c l o s e
 **************************************************************************
 *  Name:       rapclose
 *  Purpose:    closes connection to the card and makes it available
 *      for next process to open it.
 *  Returns:    0 = Success, or errno
 **************************************************************************/
int
rapclose (dev_t dev, int flag, int otyp, cred_t *cred)
```

```
{
    cardInfo_t      *ci;
    ci = &cardInfo;
    #ifdef DEBUG
    cmn_err (CE_NOTE,
    "rapclose: ci_state = %x, di_state = %x, full = %d, empty = %d",
        ci->ci_state, ci->di_state, ci->ri_full, ci->ri_free );
    #endif
    /*========================*
     *  Error Checking     *
     *========================*/
    /*  card is not installed */
    if ( !(ci->ci_state & CARD_INSTALLED) )
        return (ENODEV);
    /*  card is not opened  */
    if ( !(ci->ci_state & CARD_OPENED) )
        return (EACCES);
    /*  we do not own the card  */
    if ( ci->ci_pid != User_pid )
        return (EACCES);
    return ( rapClose(1) );
}
/************************************************************************
 *          r a p i n t r
 ************************************************************************
 *  Name:       rapintr
 *  Purpose:    Interrupt handling routine
 *  Returns:    None.
 ************************************************************************/
void
rapintr ( int ctl )
{
    ushort_t        gpis;
    uchar_t     dtci;
    uchar_t     stereo;
    uchar_t     totreq;
    uchar_t     playing;
    uchar_t     moveData;
    cardInfo_t  *ci;
    caddr_t     addr;
    ci = &cardInfo;
    addr = ci->ci_addr[0];
    /*
     *   moveData:  0 = we should move data between Buf/DMA  to DMA/Buf.
     *   totreq:    In stereo, we have to wait for 2 BF or BH interrupt
     *         but in Mono we have to wait for only one.
     *   playing:   1 = Playing, 0= Recording.
     */
    moveData = 0;
    totreq = (ci->ci_state & CARD_STEREO? 2:1); /* No. of Ints. we need */
    playing = ci->ci_state & CARD_PLAYING;
    gpis = INPW(addr+GPIS);
    /*
     *  First, check for stray interrupts and ignore them
     */
    if ( !(ci->ci_state & (CARD_PLAYING | CARD_RECORDING)) ) {
        #ifdef DEBUG
        cmn_err (CE_NOTE,
            "rapintr: Stray interupt, gpis = %x, ci_state = %x",
            ci->ci_state );
        #endif
        return;
    }
    #ifdef DEBUG
    cmn_err (CE_NOTE, "rapintr: New ..Gpis = %x", gpis );
    #endif
    /*********************************
     *     DMA Buffers Half/Full      *
     *********************************/
    while ( gpis & GPIS_ITC ) {
        /*   see which buffer is half/full   */
        dtci = INPB(addr+DTCI);
        #ifdef DEBUG
        cmn_err (CE_NOTE,
            "rapintr: Dma buffer status..Gpis = %x, Dtci = %x", gpis, dtci);
        #endif
        if ( dtci & DTCI_BF0 )
            ci->di_bf++;
        if ( dtci & DTCI_BF1 )
            ci->di_bf++;
        if (dtci & DTCI_BH0 )
            ci->di_bh++;
        if (dtci & DTCI_BH1 )
            ci->di_bh++;
        #ifdef DEBUG
        cmn_err (CE_NOTE, "rapintr: di_bf = %d, di_bh = %d",
            ci->di_bf, ci->di_bh );
        #endif
        /*  1st half of dma needs service  */
        if ( ci->di_bh == totreq ) {
            #ifdef DEBUG
            cmn_err (CE_NOTE,
                “rapintr: DMA First_Half needs service”);
            #endif
            ci->di_bh    = 0;
            ci->di_which = 0;  /* 1st half of DMA buffer */
            moveData     = 1;
        }
        /*   2nd half of dma needs service  */
        else if ( ci->di_bf == totreq ) {
            #ifdef DEBUG
            cmn_err (CE_NOTE,
            “rapintr: DMA Second_Half needs service”);
            #endif
            ci->di_bf    = 0;
            ci->di_which = 1;   /*  2nd half of DMA buffer */
            moveData     = 1;
        }
        /*
         *   Move data if needed
         */
        if ( moveData ) {
            /*  move data for Play if only data available */
            if ( playing ) {
                /*  No more data..end of play  */
                if ( ci->ri_full <= 0 ) {
                    if (ci->di_state & DI_DMA_END_PLAY ) {
                        #ifdef DEBUG
                        cmn_err (CE_NOTE,”rapintr: End of Play Reached”);
                        #endif
                        if ( ci->ri_state & RI_WANTED_EMPTY ) {
                            #ifdef DEBUG
                            cmn_err (CE_NOTE,
                            “rapintr: Cir.Buff is Wanted Empty”);
                            #endif
                            ci->ri_state &= ~RI_WANTED_EMPTY;
                            wakeup (ci);
                        }
                        else rapStop(DI_DMA_PLAYING);
                            return;
                    } else {
                        #ifdef DEBUG
                        cmn_err (CE_NOTE,
                            “rapintr: Playing but no Full buffers”);
                        #endif
```

```
                        return;
                    }
                }
                /*  Data is available to play  */
                #ifdef DEBUG
                cmn_err (CE_NOTE,
                "rapintr: Playing..which = %d, idx = %d, full = %d, Empty = %d",
                ci->di_which, ci->di_idx, ci->ri_full, ci->ri_free);
                #endif
                rapBufToDma(DMA_HALF_SIZE);
            } /* if playing  */
            else  {   /*  recording  */
                #ifdef DEBUG
                cmn_err (CE_NOTE,
                    "rapintr: Recording..which = %d, full = %d, Empty = %d",
                        ci->di_which, ci->ri_full, ci->ri_free);
                #endif
                rapDmaToBuf(DMA_HALF_SIZE);
            }
        }  /*  if move data  */
        else { /*   no need to move data  */
            #ifdef DEBUG
            cmn_err (CE_NOTE,
                "rapintr: Waiting for next interrupt, bf = %d, bh = %d",
                    ci->di_bf, ci->di_bh);
            #endif
        }
        gpis = INPW(addr+GPIS);
        #ifdef DEBUG
        cmn_err (CE_NOTE, "rapintr: next Gpis = %x", gpis);
        #endif
    } /* while ( gpis & ..  */
    #ifdef DEBUG
    cmn_err (CE_NOTE, "rapintr: finished ...");
    #endif
}  /*** End rapintr   ***/
/*************************************************************************
 *         r a p i o c t l
 *************************************************************************
 *  Name:       rapioctl
 *  Purpose:    handles IOCTL calls for RAP-10.
 *  Returns:    0 = Success, or errno
 *************************************************************************/
int
rapioctl (dev_t dev, int cmd, void *arg, int mode, cred_t *cred, int *ret)
```

```
{
    cardInfo_t      *ci;
    ci = &cardInfo;
    #ifdef DEBUG
    cmn_err (CE_NOTE, “rapioctl: Cmd = %d, full = %d, Empty = %d”,
        cmd, ci->ri_full, ci->ri_free );
    #endif
    /*
     *  No card is installed or card is already opened
     */
    if ( !(ci->ci_state & CARD_INSTALLED) )
        return (ENODEV);
    if ( !(ci->ci_state & CARD_OPENED) )
        return (EACCES);
    if ( ci->ci_pid != User_pid )
        return (EACCES);
    *ret = 0;
    switch ( cmd ) {
        case RAPIOCTL_END_PLAY:
        /*=======================*
         *   End PLAY            *
         *=======================*/
            if ( !(ci->ci_state & CARD_PLAYING) ) {
                #ifdef DEBUG
                cmn_err (CE_NOTE,
                    “rapioctl: End_PLay command in wrong state”);
                #endif
                return (EACCES);
            }
            return (rapClose (0) );
        case RAPIOCTL_END_RECORD:
        /*=======================*
         *   End RECORD          *
         *=======================*/
            if ( !(ci->ci_state & CARD_RECORDING) ) {
                #ifdef DEBUG
                cmn_err (CE_NOTE,
                    “rapioctl: End_Recrd command in wrong state”);
                #endif
                return (EACCES);
            }
            return (rapClose (0) );
    } /* switch  */
    return (0);
} /** End rapioctl  **/
```

```
/**************************************************************************
 ******            I n t e r n a l   R o u t i n e s              *******
 **************************************************************************/
/************************************************************************
 *                     r a p C l o s e
 ************************************************************************
 *  Name:       rapClose
 *  Purpose:    Routine to actually ends current operation and releases
 *              the card. It is written as a separate routine here so
 *              it can be shared by rapclose() and rapioctl() routines.
 *              One frees up the card, one does not. Also if we are called
 *              from ioctl, we will wait till all buffers are played (if
 *              in Playback mode).
 *  Returns:    0 = Success, or errno
 ************************************************************************/
int
rapClose( uchar_t relCard )
{
    cardInfo_t  *ci;
    rwBuf_t     *rw;
    int         s, totLeft;
    ci = &cardInfo;
    s = LOCK();
    rw = &rwQue[ci->ri_idx];
    #ifdef DEBUG
    cmn_err (CE_NOTE,
        "rapClose: relCard = %d, ci_state = %x, di_state = %x",
        relCard, ci->ci_state, ci->di_state );
    #endif
    /*
     *  if we are not recording and are not playing
     *  then simply mark the card as not opened and return
     */
    if ( !(ci->ci_state & (CARD_RECORDING | CARD_PLAYING)) ) {
        #ifdef DEBUG
        cmn_err (CE_NOTE, "rapClose: Idle card ..closing");
        #endif
        if ( relCard ) {
            ci->ci_state &= ~CARD_OPENED;
            ci->ci_pid    = 0;
        }
        UNLOCK(s);
        return(0);
    }
    /*
     *  Recording ? end it.
     */
    if ( ci->ci_state & CARD_RECORDING ) {
        #ifdef DEBUG
        cmn_err (CE_NOTE,”rapClose: Ending Record (A/D)”);
        #endif
        rapStop(DI_DMA_RECORDING);
        if ( relCard ) {
            ci->ci_state &= ~CARD_OPENED;
            ci->ci_pid    = 0;
        }
        UNLOCK(s);
        return(0);
    }
    /*
     *  playback and called from close() routine ?
     *  End the playback
     */
    if ( relCard ) {
        #ifdef DEBUG
        cmn_err (CE_NOTE,
        “rapClose: Ending Playback (D/A”);
        #endif
        rapStop(DI_DMA_PLAYING);
        ci->ci_state &= ~CARD_OPENED;
        ci->ci_pid    = 0;
        UNLOCK(s);
        return(0);
    }
    /*
     *  Called from Ioctl.
     *  Closing in middle of play is different based on we
     *  have been called from close() routine or not.
     *  If called from Ioctl (relCard = 0), we will wait till
     *  all buffers are played back.
     */
    if ( !(rw->rw_state & RW_FULL) && (rw->rw_count > 0) ) {
        totLeft = RW_BUF_SIZE - rw->rw_count;
        #ifdef DEBUG
        cmn_err (CE_NOTE,
            “rapClose: Current Buf %d has %d data. Filled with %d zeros”,
            rw->rw_idx, rw->rw_count, totLeft );
        #endif
        if ( totLeft > 0 ) {
            bzero (ci->ri_ptr, totLeft);
```

```
                ci->ri_ptr += totLeft;
            }
            rapMarkBuf(rw, ci, RW_FULL);
        }
        /*   some buffers to play  */
        if ( ci->ri_full > 0 )  {
            /*   Playback has not started yet */
            if ( ci->di_state == DI_DMA_IDLE ) {
                #ifdef DEBUG
                cmn_err (CE_NOTE,
                “rapClose: Starting playback, full = %d, empty = %d”,
                ci->ri_full, ci->ri_free);
                #endif
                rapStart(DI_DMA_PLAYING);
            }
            ci->di_state  = DI_DMA_IDLE;
            ci->di_state |= DI_DMA_END_PLAY;
            /*  wait till buffers are all played back */
            while ( ci->ri_full > 0 ) {
                #ifdef DEBUG
                cmn_err (CE_NOTE,
                    “rapClose: waiting for Play to end..full = %d, empty = %d, ri_idx
= %d, di_idx = %d”,
                    ci->ri_full, ci->ri_free, ci->ri_idx, ci->di_idx);
                #endif
                ci->ri_state |= RI_WANTED_EMPTY;
                if ( sleep (ci, PUSER | PCATCH) ) {
                    #ifdef DEBUG
                    cmn_err (CE_NOTE, “rapClose: Interrupted”);
                    #endif
                    rapStop(DI_DMA_PLAYING);
                    ci->ci_state &= ~CARD_OPENED;
                    ci->ci_pid    = 0;
                    UNLOCK(s);
                    return (EINTR);
                }
            }
            rapStop(DI_DMA_PLAYING);
        }
        else {
            #ifdef DEBUG
            cmn_err (CE_NOTE, “rapClose: Circular buffer empty..closing”);
            #endif
            rapStop(DI_DMA_PLAYING);
        }
```

```
    UNLOCK(s);
    return(0);
}  /***    End rapClose   ***/
/*************************************************************************
 *                   r a p S t o p
 *************************************************************************
 *  Name:       rapStop
 *  Purpose:    Stops D/A and A/D conversion.
 *  Returns:    None.
 *************************************************************************/
static void
rapStop( uchar_t what )
{
    cardInfo_t  *ci;
    rwBuf_t     *rw;
    caddr_t     addr;
    uchar_t     dacm, adcm;
    ushort_t    gpdi;
    int         s, i;
    s = LOCK();
    ci = &cardInfo;
    addr = ci->ci_addr[0];
    gpdi = adcm = dacm = 0;
    #ifdef DEBUG
    cmn_err (CE_NOTE,
        “rapStop:  Stoping %s, full = %d, Empty = %d”,
        (what == DI_DMA_PLAYING ? “Playback(D/A)”:”Record(A/D)”),
        ci->ri_full, ci->ri_free);
    #endif
    switch ( what ) {
        /*  stop D/A Conversion (Playing)  */
        case DI_DMA_PLAYING:
            ci->di_which = 0;
            rapZeroDma(ci, DMA_BUF_SIZE);
            OUTB(addr+DACM, dacm);
            rapNoteOff (ci);
            break;
        /*  stop A/D Conversion (recording) */
        case DI_DMA_RECORDING:
            OUTB(addr+ADCM, adcm);
            OUTB(addr+DACM, dacm);
            break;
    }
    OUTW(addr+GPDI, gpdi);
    rapReleaseDma(ci);
    /*   Initialize Card Info structure  */
    ci->ci_state &= ~(CARD_PLAYING | CARD_RECORDING);
    ci->ri_idx   = 0;
    ci->di_idx   = 0;
    ci->ri_state = 0;
    ci->di_state = 0;
    ci->di_ptr   = rwQue[0].rw_buf;
    ci->ri_ptr   = rwQue[0].rw_buf;
    ci->ri_free  = RW_BUF_COUNT;
    ci->ri_full  = 0;
    /*   Initialize Circular Buffers   */
    for ( i = 0; i < RW_BUF_COUNT; i++ ) {
        rw = &rwQue[i];
        rw->rw_count = 0;
        rw->rw_state = 0;
        rw->rw_idx   = i;
        bzero (rw->rw_buf, RW_BUF_SIZE);
    }

    /*   clear out any hanging GPIS and DACM   */
    gpdi = INPW(addr+GPIS);
    UNLOCK(s);
} /** End rapStop  **/
/***********************************************************************
 *            r a p S t a r t
 ***********************************************************************
 *  Name:       rapStart
 *  Purpose:    Prepares Eisa DMA buffers/Control block for Playing/Recording
 *              This function is called when DMA is Idle.
 *  Returns:    0 = Success or Error number.
 ***********************************************************************/
static int
rapStart (uchar_t  what)
{
    cardInfo_t  *ci;
    dmaBuf_t    *dmaB;
    dmaCb_t     *dmaC;
    uchar_t     stereo;
    int         err;
    ci = &cardInfo;
    stereo = (ci->ci_state & CARD_STEREO);
    #ifdef DEBUG
    cmn_err (CE_NOTE,
        "rapStart: Starting %s, full = %d, empty = %d",
        (what == DI_DMA_PLAYING ? "Playback(D/A)":"Record(A/D)"),
        ci->ri_full,  ci->ri_free );
    #endif
    /*    clear Dma buffers   */
    ci->di_which = 0;
    rapZeroDma(ci, DMA_BUF_SIZE);
    /*   check for Dma buffer addresses  */
    if ( (ci->ci_dmaBuf5 == (dmaBuf_t *)0) ||
         (ci->ci_dmaCb5  == (dmaCb_t  *)0) ) {
        cmn_err (CE_WARN,
            "rapStart: Chan 5 dmaBuf/dmaCb is NULL, what = %d", what);
        return(EIO);
    }
    if ( (ci->ci_dmaBuf6 == (dmaBuf_t *)0) ||
         (ci->ci_dmaCb6  == (dmaCb_t  *)0) ) {
        cmn_err (CE_WARN,
            "rapStart: Chan 6 dmaBuf/dmaCb is NULL, what = %d", what);
        return(EIO);
    }
    /*
     *  Prepare Eisa Buf and Cb for Channel 5. If in
     *  stereo mode, do the same for Channel 6.
     */
    dmaB = ci->ci_dmaBuf5;
    dmaC = ci->ci_dmaCb5;
    rapPrepEisa (dmaB, dmaC, what, dmaLeftPhys );
    if ( stereo ) {
        dmaB = ci->ci_dmaBuf6;
        dmaC = ci->ci_dmaCb6;
        rapPrepEisa (dmaB, dmaC, what, dmaRightPhys );
    }
    /*
     *   Program Eisa DMA Channels
     */
    err = eisa_dma_prog (ci->ci_adap, ci->ci_dmaCb5, ci->ci_dmaCh5,
                           EISA_DMA_NOSLEEP);
    if ( err == 0 ) {
        cmn_err (CE_WARN, "rapStart: DMA Channel %d is busy",
                ci->ci_dmaCh5 );
        return (EBUSY);
    }
    if ( stereo ) {
        err = eisa_dma_prog (ci->ci_adap, ci->ci_dmaCb6, ci->ci_dmaCh6,
            EISA_DMA_NOSLEEP);
        if ( err == 0 ) {
            cmn_err (CE_WARN,
                “rapStart: DMA Channel %d is busy”,
                ci->ci_dmaCh6 );
            return (EBUSY);
        }
    }
    /*   enable hardware recognition on Eisa Dma Channels */
    eisa_dma_enable (ci->ci_adap, ci->ci_dmaCb5, ci->ci_dmaCh5,
                EISA_DMA_NOSLEEP);
    if ( stereo ) {
        eisa_dma_enable (ci->ci_adap, ci->ci_dmaCb6, ci->ci_dmaCh6,
                EISA_DMA_NOSLEEP);
    }
    /*  set Eisa DMA register for Autoinit mode  */
    rapSetAutoInit(ci, what);
    ci->di_state |= what;
    /*  let’s do it ! */
    if ( what == DI_DMA_PLAYING ) {
        #ifdef DEBUG
        cmn_err (CE_NOTE, “rapStart: Starting DMA for D/A Play”);
        #endif
        rapStartDA();
    }
    else {
        #ifdef DEBUG
        cmn_err (CE_NOTE, “rapStart: Starting DMA for A/D Record”);
        #endif
        rapStartAD();
    }
    return(0);
} /**  End rapStart  **/
/************************************************************************
 *           r a p P r e p E i s a
 ************************************************************************
 *  Name:        rapPrepEisa
 *  Purpose:     prepares EISA Buf and Cb structures.
 *  Returns:     None.
 ************************************************************************/
static void
rapPrepEisa( dmaBuf_t *dmaB, dmaCb_t *dmaC, uchar_t  what, paddr_t addr)
{
    #ifdef DEBUG
    cmn_err (CE_NOTE,
        “rapPrepEisa: Preparing Eisa DMA buffers for %s”,
        (what == DI_DMA_PLAYING ? “Playback(D/A)” : “Record(A/D)” ) );
    #endif
    /*   prepare Eisa DMA Buf struct   */
    bzero (dmaB, sizeof(dmaBuf_t) );
    dmaB->count = DMA_BUF_SIZE;
    dmaB->address = addr;
    /*  prepare Eisa DMA Control Block  */
    bzero (dmaC, sizeof(dmaCb_t) );
    dmaC->reqrbufs   = dmaB;
    dmaC->reqr_path  = EISA_DMA_PATH_16;
    dmaC->trans_type = EISA_DMA_TRANS_DMND;
    dmaC->targ_step  = EISA_DMA_STEP_INC;
    dmaC->bufprocess = EISA_DMA_BUF_SNGL;
    if ( what == DI_DMA_PLAYING )
        dmaC->cb_cmd = EISA_DMA_CMD_READ;   /*  mem -> rap10     */
    else
        dmaC->cb_cmd = EISA_DMA_CMD_WRITE;  /*  rap10 -> mem     */
} /*** End rapPrepEisa  ***/
/*************************************************************************
 *             r a p S t a r t D A
 *************************************************************************
 *  Name:         rapStartDA
 *  Purpose:      Enables appropriate RAP interrupts and starts D/A Dma.
 *  Returns:     None
 *************************************************************************/
static void
rapStartDA()
{
    cardInfo_t  *ci;
    caddr_t     addr;
    ushort_t    gpdi, gpis, gpst, dtcd, mask;
    uchar_t     gpcm, pwmd, adcm, dacm;
    uchar_t     stereo;
    int         s;
     ci = &cardInfo;
    addr = ci->ci_addr[0];
    stereo = ci->ci_state & CARD_STEREO;
    #ifdef DEBUG
    cmn_err (CE_NOTE,
        "rapStartDA: Starting D/A Dma, full = %d, empty = %d",
        ci->ri_full, ci->ri_free );
    #endif
    /*
     *  Prepare the board for Record (A/D)
     *  Here is what we will do (in exact order):
     *
     *   GPDI:  Stereo = 0xA800, Mono = 0x9800
     *    itc = 1, dma transfer match count
     *    Stereo:   Drq1->Dma5, Drq0->Dma6
     *    Mono:     Drq1->Dma5
     *      Dt1, Dt0 = 10, Chan 1 ->Drq1, Chan 0 ->Drq0
     *    Left Chan->Drq1, Right Chan->Drq0
     *
     *   DACM:  Stereo: BF, Mono: BE
     *    scc = 1, Dma size in byte
     *    ts1 = ts2 = 1, transfer size of 4096 bytes
     *      dl1 = dl0 = 1; Data length of 16 bits for both Channels.
     *    Stereo ? ds1 = ds0 = 1  Start D/A on both Channels.
     *      Mono   ? ds1 = 1        Start D/A on Channel 1
     *
     *   GPCM:   Select Mike level = 0x04
     *         Aux  level = 0x08
     *   PWMD:   0xFF (Max level)
     */
    gpdi = (stereo ? 0xA800: 0x9800);
    dacm = (stereo ? 0xBF:0xBE);
    gpcm = 0x04;
    pwmd = 0xFF;
    mask = (stereo ? (GPIS_DN1|GPIS_DN0): GPIS_DN1);
    #ifdef DEBUG
    cmn_err (CE_NOTE,
        "rapStartDA: gpdi = %x, dacm = %x", gpdi, dacm);
    #endif
    /*  Set Rap-10 card   */
    OUTB(addr+GPCM, gpcm);
    OUTB(addr+PWMD, pwmd);
    OUTW(addr+GPDI, gpdi);
    OUTB(addr+DACM, dacm);
    /*
     *  Busy-wait for both Note_On interrupts
     *  The interrupt version is commenetd out for now.
     */
    gpis = INPW(addr+GPIS);
    #ifdef DEBUG
    cmn_err (CE_NOTE,
        "rapStartDA: Waiting for Note_On, gpis = %x, mask = %x",
         gpis, mask);
    #endif
    while ( !(gpis & mask) ) {
        gpis = INPW(addr+GPIS);
        #ifdef DEBUG
        cmn_err (CE_NOTE, "rapStartDA: Waiting ..new gpis = %x", gpis);
```

```
        #endif
    }
    #ifdef DEBUG
    cmn_err (CE_NOTE, "rapStartDA:  Note_On Interrupt Received, gpis = %x",
    gpis );
    #endif
    rapNoteOn(ci, gpis);
}  /***   End rapStartDA   ***/
/***************************************************************************
 *                      r a p S t a r t A D
 ***************************************************************************
 *  Name:         rapStartAD
 *  Purpose:      Enables appropriate RAP interrupts and starts A/D Dma.
 *  Returns:     None
 ***************************************************************************/
static void
rapStartAD()
{
    cardInfo_t  *ci;
    caddr_t     addr;
    ushort_t    gpdi;
    uchar_t     gpcm, pwmd, adcm, dacm;
    uchar_t     stereo, mic;
    ci = &cardInfo;
    addr = ci->ci_addr[0];
    stereo = ci->ci_state & CARD_STEREO;
#ifdef DEBUG
    cmn_err (CE_NOTE,
        "rapStartAD: Starting A/D Dma in %s, full = %d, empty = %d",
        (stereo ? "Stereo":"Mono"), ci->ri_full, ci->ri_free );
#endif
    /*
     *  Prepare the board for Record (A/D)
     *  Here is what we will do (in exact order):
     *
     *   GPDI:  Stereo = 0xA400, Mono = 0x9400
     *    itc = 1, dma transfer match count
     *    Stereo:   Drq1->Dma5, Drq0->Dma6
     *    Mono:     Drq1->Dma5
     *      Dt1, Dt0 = 01, Left Chan->Drq1, Right Chan->Drq0
     *
     *   DACM:  0xB0
     *    scc = 1, Dma size in byte
     *    ts1 = ts2 = 1, transfer size of 4096 bytes
     *
     *   GPCM:   Select Mic level = 0x04
     *        Aux level = 0x08
     *   PWMD:   0xFF (Max level)
     *
     *   ADCM:  Stereo: Mic 0x6F, line 0x4F,
     *    Mono:   Mic 0x6D, line 0x4D
     *      Mon = 1, Monitor ON
     *      Gin = 1, Head Amp Gain to Mic.
     *      Af1, Af0 = 01, 22.05 KHz
     *      Adl = 1, 16 bit data length
     *      Stereo,  Adm = 1, else = 0
     *      Ads = 1, Start A/D
     */
    gpdi = (stereo ? 0xA400: 0x9400);
    gpcm = 0x08;
    adcm = (stereo ? 0x6F:0x6D);
    dacm = 0xB0;
    gpcm = 0x04;
    pwmd = 0xFF;
    #ifdef DEBUG
    cmn_err (CE_NOTE,
    “rapStartAD: Rap init as: gpdi = %x, dacm = %x, gpcm = %x, adcm = %x”,
    gpdi, dacm, gpcm, adcm);
    #endif
    OUTW(addr+GPDI, gpdi);
    OUTB(addr+DACM, dacm);
    OUTB(addr+GPCM, gpcm);
    OUTB(addr+PWMD, pwmd);
    OUTB(addr+ADCM, adcm);
}  /***  End rapStartAD  ***/
/*************************************************************************
 *         r a p B u f T o D m a
 *************************************************************************
 *  Name:       rapBufToDma
 *  Purpose:    moves data from current rwQue[] entry to DMA buffers.
 *              This routine is called by INterrupt handler only except
 *              once before we startd D/A (when no DMA is programmed yet)
 *  Returns:    None
 *************************************************************************/
static void
rapBufToDma( int  bytes)
{
    cardInfo_t    *ci;
    rwBuf_t       *rw;
    uchar_t       *dmaR;
    uchar_t       *dmaL;
    uchar_t       stereo;
    int           i, j, s;
    ci = &cardInfo;
    rw = &rwQue[ci->di_idx];
    stereo = ci->ci_state & CARD_STEREO;
    /*
     *      filling 1st half or 2nd half of the buffers ?
     */
    if ( ci->di_which ) {
        dmaR = &dmaRight[DMA_HALF_SIZE];
        dmaL = &dmaLeft[DMA_HALF_SIZE];
        if ( bytes == DMA_BUF_SIZE ) {
            bytes = DMA_HALF_SIZE;
        }
    }
    /*  filling 1st half of dma buffers */
    else {
        dmaR = &dmaRight[0];
        dmaL = &dmaLeft[0];
    }
    #ifdef DEBUG
    cmn_err (CE_NOTE,
    "rapBufToDma: Bytes = %d, which = %d, Idx = %d, rw_count = %d, Full = %d, Emp
ty = %d",
    bytes, ci->di_which, ci->di_idx, rw->rw_count, ci->ri_full,
    ci->ri_free);
    #endif
    /*
     *   if buffer is not Full, we zero out dma buffers and
     *   return. We cannot wait till it gets Full.
     */
    if ( !(rw->rw_state & RW_FULL) ) {
        rapZeroDma(ci, bytes);
        ci->di_ptr = NULL;
        #ifdef DEBUG
        cmn_err (CE_NOTE,
            "rapBufToDma: Buf %d is not Full, rw_state = %x",
        rw->rw_idx, rw->rw_state );
        #endif
        return;
    }
    /*  buffer is full of data ..readjust the buffer pointer */
    if ( ci->di_ptr == NULL )
        ci->di_ptr = rw->rw_buf;
```

```
/*
 *  Fill buffers ...
 */
for ( i = 0; i < bytes; i++ ) {
    /*
     *  First check if buffer is empty. If it is, mark it
     *  as empty, wake anyone up who wants it and get the
     *  next full buffer.
     */
    if ( rw->rw_count <= 0 ) {
        #ifdef DEBUG
        cmn_err (CE_NOTE,
        "rapBufToDma: Buf %d is Empty now, rw_count = %d",
         rw->rw_idx, rw->rw_count );
        #endif
        rapMarkBuf(rw, ci, RW_EMPTY);
        ci->di_ptr = NULL;
        if ( rw->rw_state & RW_WANTED_EMPTY ) {
            #ifdef DEBUG
            cmn_err (CE_NOTE,
            "rapBufToDma: Buf %d is Wanted_Empty",
            rw->rw_idx );
            #endif
            rw->rw_state &= ~RW_WANTED_EMPTY;
            wakeup(rw);
        }
        j = rapGetNextFull (ci->di_idx, FROM_INTR);
        if ( j == -1 ) {
            #ifdef DEBUG
            cmn_err (CE_NOTE,
            "rapBufToDma: Could not get next Full buffer");
            #endif
            break;
        }
        ci->di_idx = j;
        rw = &rwQue[ci->di_idx];
        ci->di_ptr = rw->rw_buf;
        continue;
    }
    /*  buffer still has some data ..move them  */
    if ( stereo ) {
        dmaL[i] = *(ci->di_ptr++);
        dmaR[i] = *(ci->di_ptr++);
        rw->rw_count -= 2;
    }
```

```
        else {
            dmaL[i] = *(ci->di_ptr++);
            rw->rw_count--;
        }
    } /* for .. */
    /*  Flush the cache line so that Dma buffers contain all data */
    dki_dcache_wbinval (dmaL, (unsigned)bytes);
    if ( stereo )
        dki_dcache_wbinval (dmaR, (unsigned)bytes);
}  /*** end rapBufToDma  ***/
/************************************************************************
 *                      r a p D m a T o B u f
 ************************************************************************
 *  Name:       rapDmaToBuf
 *  Purpose:    Moves data from DMA buffers (Right and Left in stereo)
 *              into a rwQue entry.  This routine is called only by
 *              Interrupt Handler.
 *  Returns:    None
 ************************************************************************/
static void
rapDmaToBuf( int  bytes)
{
    cardInfo_t    *ci;
    rwBuf_t       *rw;
    uchar_t       *dmaR;
    uchar_t       *dmaL;
    uchar_t       stereo;
    int    i, j, s, inc;
    ci = &cardInfo;
    rw = &rwQue[ci->di_idx];
    stereo = ci->ci_state & CARD_STEREO;
    /*
     *      filling 1st half or 2nd half of the buffers ?
     */
    if ( ci->di_which ) {
        dmaR = &dmaRight[DMA_HALF_SIZE];
        dmaL = &dmaLeft[DMA_HALF_SIZE];
        if ( bytes == DMA_BUF_SIZE ) {
            bytes = DMA_HALF_SIZE;
        }
    }
    /*  filling 1st half of dma buffers */
    else {
        dmaR = &dmaRight[0];
        dmaL = &dmaLeft[0];
```

```
}
/*   Invalidate the Cache    */
dki_dcache_inval (dmaL, (unsigned)bytes);
if ( stereo )
    dki_dcache_inval (dmaR, (unsigned)bytes);
#ifdef DEBUG
cmn_err (CE_NOTE,
"rapDmaToBuf: Bytes= %d, Idx = %d, rw_count = %d, Full = %d, Empty= %d",
bytes, ci->di_idx, rw->rw_count, ci->ri_full, ci->ri_free);
#endif
/*
 *  if buffer is Full ..we cannot wait ! Ignore new data
 *  by simply returning.
 */
if ( rw->rw_state & RW_FULL ) {
    #ifdef DEBUG
    cmn_err (CE_NOTE,
    "rapDmaToBuf: Buf %d is not Empty ..Ignoring data",
    rw->rw_idx );
    #endif
    return;
}
/*  buffer is Empty ..calculate the end address */
if ( ci->di_ptr == NULL )
    ci->di_ptr = rw->rw_buf;
#ifdef DEBUG
cmn_err (CE_NOTE,
"rapDmaToBuf: Moving %s of DMA buffers in %s, rw_count = %x",
(ci->di_which ? "Second Half" : "First Half"),
(stereo ? "Stereo":"Monoe"), rw->rw_count);
#endif
/*
 *      Fill buffers ...in stereo bytes are Left:Right:Left:Right...
 */
for ( i = 0; i < bytes; i++ ) {
    /*
     *  First check if this buffer is Full or not.
     *  If it is, mark it as Full and wake anyone up who is
     *  waiting for it. Then get the next Empty buffer.
     */
    if ( rw->rw_count >= RW_BUF_SIZE ) {
        #ifdef DEBUG
        cmn_err (CE_NOTE,
        "rapDmaToBuf: Buf %d is Full now, rw_count = %d",
        rw->rw_idx, rw->rw_count );
```

```
                #endif
                rapMarkBuf(rw, ci, RW_FULL);
                if ( rw->rw_state & RW_WANTED_FULL ) {
                    #ifdef DEBUG
                    cmn_err (CE_NOTE,
                    “rapDmaToBuf: Buf %d is Wanted_Full”,
                    rw->rw_idx );
                    #endif
                    rw->rw_state &= ~RW_WANTED_FULL;
                    wakeup(rw);
                }
                j = rapGetNextEmpty(ci->di_idx, FROM_INTR);
                if ( j == -1 ) {
                    cmn_err (CE_NOTE,
                          “rapDmaToBuf: Could not get next empty”);
                    return;
                }
                ci->di_idx = j;
                rw = &rwQue[ci->di_idx];
                ci->di_ptr = rw->rw_buf;
                continue;
            }
            /*  buffer still has room ...move data  */
            if ( stereo ) {
                *(ci->di_ptr++) = dmaL[i];
                *(ci->di_ptr++) = dmaR[i];
                rw->rw_count     += 2;
            }
            else {
                *(ci->di_ptr++) = dmaL[i];
                rw->rw_count++;
            }
        } /* while bytes ... */
    }  /*** end rapDmaToBuf  ***/
    /**************************************************************************
     *                       r a p G e t N e x t F u l l
     **************************************************************************
     *  Name:       rapGetNextFull
     *  Purpose:    returns the index of next Full  entry in rwQue[],
     *              starting from a given index. Sleeps if the entry
     *              is not Full.
     *  Returns:    the index of the empty entry.
     **************************************************************************/
    static short
    rapGetNextFull (short idx, uchar_t fromIntr)
```

```
{
    cardInfo_t   *ci;
    int          s;
    toid_t       to_id;
    rwBuf_t      *rw;
    ci = &cardInfo;
    #ifdef DEBUG
    cmn_err (CE_NOTE,
    “rapGetNextFull: Getting Next Full Buffer..idx = %d, fromIntr: %d”,
    idx, fromIntr );
    #endif
    /*  go to beginning if at the end of the queu  */
    idx++;
    if ( idx >= RW_BUF_COUNT )
        idx = 0;
    rw = &rwQue[idx];
    /*
     *    if buffer is not available and we were called from Intrupt
     *    handler, simply ignore the request and return error
     */
    s = LOCK();
    if ( !(rw->rw_state & RW_FULL) && (fromIntr) ) {
        #ifdef DEBUG
        cmn_err (CE_NOTE,
        “rapGetNextFull: Buffer %d is not Full. ..Cannot Wait”,
        rw->rw_idx);
        #endif
        UNLOCK(s);
        return(-1);
    }
    /*   wait for the buffer to become Full  */
    if ( !(rw->rw_state & RW_FULL) ) {
        ci->ri_tout = 0;
        to_id = itimeout (rapTimeOut, rw, RW_TIMEOUT, plbase, 0, 0, 0);
        while ( !(rw->rw_state & RW_FULL) && !ci->ri_tout ) {
            #ifdef DEBUG
            cmn_err (CE_NOTE,
            “rapGetNextFull:  Waiting for Buf %d to become Full”,
            rw->rw_idx );
            #endif
            rw->rw_state |= RW_WANTED_FULL;
            if ( sleep(rw, PUSER | PCATCH) ) {
                untimeout(to_id);
                #ifdef DEBUG
                cmn_err (CE_NOTE, “rapGetNextFull: Interrupted”);
                #endif
                rw->rw_state &= ~RW_WANTED_FULL;
                UNLOCK(s);
                return(-1);
            }
        }
        untimeout (to_id);
        if ( ci->ri_tout ) {
            #ifdef DEBUG
            cmn_err (CE_NOTE, "raGetNextFull: Timed out");
            #endif
            rw->rw_state &= ~RW_WANTED_FULL;
            UNLOCK(s);
            return (-1);
        }
    }  /*  if !(rw->rw_state & RW_FULL)  */
    UNLOCK(s);
    #ifdef DEBUG
    cmn_err (CE_NOTE, "rapGetNextFull: next Full Buffer is %d", idx);
    #endif
    return(idx);
}  /***   End rapGetNextFull   ***/
/*************************************************************************
 *         r a p G e t N e x t E m p t y
 *************************************************************************
 *  Name:        rapGetNextEmpty
 *
 *  Purpose:     returns the index of next empty entry in rwQue[],
 *               starting from a given index. Sleeps if the entry
 *               is not empty.
 *  Returns:     the index of the empty entry.
 *************************************************************************/
static short
rapGetNextEmpty (short idx, uchar_t fromIntr)
{
    cardInfo_t  *ci;
    int         s;
    toid_t      to_id;
    rwBuf_t     *rw;
    ci = &cardInfo;
    #ifdef DEBUG
    cmn_err (CE_NOTE,
        "rapGetNextEmpty: Getting Next Empty Buffer..idx = %d, fromIntr: %d",
        idx, fromIntr );
    #endif
```

```
/*  go to beginning if at the end of the queu  */
idx++;
if ( idx >= RW_BUF_COUNT )
    idx = 0;
rw = &rwQue[idx];
s = LOCK();
/*
 *    if buffer is nit available and we were called from Intrupt
 *    handler, simply ignore the request and return error
 */
if ( (rw->rw_state & RW_FULL) && (fromIntr) ) {
    #ifdef DEBUG
    cmn_err (CE_NOTE,
    “rapGetNextEmpty: Buffer %d is not Empty ..Cannot Wait”,
    rw->rw_idx);
    #endif
    UNLOCK(s);
    return(-1);
}
/*   wait for the buffer to become Empty   */
if ( rw->rw_state & RW_FULL ) {
    ci->ri_tout = 0;
    to_id = itimeout (rapTimeOut, rw, RW_TIMEOUT, plbase, 0, 0, 0);
    while ( (rw->rw_state &  RW_FULL) && !ci->ri_tout ) {
        #ifdef DEBUG
        cmn_err (CE_NOTE,
        “rapGetNextEmpty: Waiting for Buf %d to become Empty”,
        rw->rw_idx );
        #endif
        rw->rw_state |= RW_WANTED_EMPTY;
        if ( sleep(rw, PUSER | PCATCH) ) {
            untimeout(to_id);
            #ifdef DEBUG
            cmn_err (CE_NOTE, “rapGetNextEmpty: Interrupted”);
            #endif
            rw->rw_state &= ~RW_WANTED_EMPTY;
            UNLOCK(s);
            return(-1);
        }
    }  /*  while .. */
    untimeout (to_id);
    if ( ci->ri_tout ) {
        #ifdef DEBUG
        cmn_err (CE_NOTE, “raGetNextEmpty: Timed out”);
        #endif
```

```
            rw->rw_state &= ~RW_WANTED_EMPTY;
            UNLOCK(s);
            return (-1);
        }
    }  /*  if (rw->rw_state & RW_FULL) */
    UNLOCK(s);
    #ifdef DEBUG
    cmn_err (CE_NOTE, “rapGetNextEmpty: next Empty Buffer is %d”, idx);
    #endif
    return(idx);
}  /***    End rapGetNextEmpty   ***/
/************************************************************************
 *         r a p D i s I n t
 ************************************************************************
 *  Name:        rapDisInt
 *  Purpose:     Disables RAP-10 interrupts.
 *  Returns:     None.
 ************************************************************************/
static void
rapDisInt( cardInfo_t   *ci)
{
    caddr_t     addr;
    ushort_t    s;
    uchar_t     c;
    #ifdef DEBUG
    cmn_err (CE_NOTE, “rapDisInt: full = %d, empty = %d, di_state = %d”,
    ci->ri_full, ci->ri_free, ci->di_state );
    #endif
    addr = ci->ci_addr[0];
    /*   disable all Interrupts   */
    s = 0;
    OUTW(addr+GPDI, s);
    OUTB(addr+DACM, 0x00);
    OUTB(addr+ADCM, 0x00);
    #ifdef DEBUG
    cmn_err (CE_NOTE, “rapDisInt: Rap is set”);
    #endif
} /*** End rapDisInt  ***/
/*************************************************************************
 *                       r a p G e t D m a                              *
 *************************************************************************
 *     Name:       rapGetDma
 *     Purpose:    allocates dma Buf and Cb structures
 *     Returns:    0 = Success,  1 = Error
 *************************************************************************/
```

```
static int
rapGetDma ( dmaBuf_t  **dmaB,  dmaCb_t  **dmaC, int ch )
{
    #ifdef DEBUG
    cmn_err (CE_NOTE,
    "rapGetDma: Getting Eisa Dma Buf and Cb for Channel %d", ch);
    #endif
    *dmaB = eisa_dma_get_buf (EISA_DMA_SLEEP);
    if ( *dmaB == NULL )
        return (1);
    *dmaC = eisa_dma_get_cb ( EISA_DMA_SLEEP );
    if ( *dmaC == NULL )
        return (1);
    return (0);
} /***  End rapGetDma   ***/
/***********************************************************************
 *          r a p M a r k B u f
 ***********************************************************************
 *  Name:       rapMarkBuf
 *  Purpose:    Marks a buffer (Empty, Busy, Full) and increments/decrements
 *              appropriate counters. Buffers status changed as:
 *              Empty -> Busy -> Full -> Empty -> Busy ..
 *  Returns:    None.
 ***********************************************************************/
static void
rapMarkBuf (rwBuf_t  *rw, cardInfo_t  *ci, uchar_t  m)
{
    int s;
    s = LOCK();
    switch ( m ) {
        case RW_EMPTY:
            rw->rw_state &= ~RW_FULL;
            if ( ci->ri_full )
                ci->ri_full--;
            ci->ri_free++;
            rw->rw_count = 0;
            #ifdef DEBUG
            cmn_err (CE_NOTE,
            "rapMarkBuf: Buf %d set EMPTY. Full = %d, Emp = %d",
            rw->rw_idx, ci->ri_full, ci->ri_free );
            #endif
            break;
        case RW_FULL:
            rw->rw_state |= RW_FULL;
            ci->ri_full++;
            if ( ci->ri_free )
               ci->ri_free--;
            rw->rw_count = RW_BUF_SIZE;
            #ifdef DEBUG
            cmn_err (CE_NOTE,
                "rapMarkBuf: Buf %d set FULL. Full = %d, Emp = %d",
                rw->rw_idx, ci->ri_full, ci->ri_free );
            #endif
            break;
    }
    UNLOCK(s);
}  /***   End rapMarkBuf   ***/
/*************************************************************************
 *          r a p K e r n M e m
 *************************************************************************
 *  Name:       rapKernMem
 *  Purpose:    Allocates/Disallocates Kernel memory for Right and
 *              Left DMA channels.
 *  Returns:    0 = Success,  1 = Failure.
 *************************************************************************/
static int
rapKernMem ( uchar_t what)
{
    #ifdef DEBUG
    cmn_err (CE_NOTE, "rapKernMem: %s Kernel Contigious Memory",
         (what == 1 ? "Allocating" : "Deallocating") );
    #endif
    switch ( what ) {
        /*=====================================*
         *     Allocate Right/Left DMA Channels  *
         *=====================================*/
        case 1:
            dmaRight = kmem_alloc (DMA_BUF_SIZE,
                            KM_NOSLEEP | KM_PHYSCONTIG | KM_CACHEALIGN );
            if ( dmaRight == (caddr_t)NULL ) {
                cmn_err (CE_WARN,
                 "rapKernMem: Cannot allocate DMA memory for R_chann");
                 return(1);
            }
             dmaLeft = kmem_alloc (DMA_BUF_SIZE,
                            KM_NOSLEEP | KM_PHYSCONTIG | KM_CACHEALIGN );
            if ( dmaLeft == (caddr_t)NULL ) {
                cmn_err (CE_WARN,
                    "rapKernMem: Cannot allocate DMA memory for L_chann");
                kmem_free (dmaRight, DMA_BUF_SIZE);
```

```
                    return(1);
                }
                /*   get the physicall address  */
                dmaRightPhys = kvtophys(dmaRight);
                dmaLeftPhys  = kvtophys(dmaLeft);
                return(0);
            /*=======================================*
             *   Deallocate Right/Left DMA Channels  *
             *=======================================*/
            case 2:
                if ( dmaRight != NULL ) {
                    kmem_free (dmaRight, DMA_BUF_SIZE);
                    dmaRight = (caddr_t)NULL;
                }
                if ( dmaLeft != NULL ) {
                    kmem_free (dmaLeft, DMA_BUF_SIZE);
                    dmaLeft = (caddr_t)NULL;
                }
                return(0);
        }  /* switch */
}  /***   End rapKernMem   ***/
/**************************************************************************
 *          r a p T i m e O u t
 **************************************************************************
 *  Name:       rapTimeOut
 *  Purpose:    is called when Read/Write waiting for buffers time out.
 *  Returns:
 **************************************************************************/
static void
rapTimeOut( void *addr )
{
    cardInfo_t    *ci;
    ci = &cardInfo;
    /*   indicate a timeout  */
    ci->ri_tout = 1;
    wakeup (addr);
}
/**************************************************************************
 *                      r a p N o t e O n
 **************************************************************************
 *  Name:       rapNoteOn
 *  Purpose:    Sends a MIDI Note_On message.
 *              This code is taken from RAP-10 manual.
 *  Returns:    None.
 **************************************************************************/
```

```
static void
rapNoteOn ( cardInfo_t  *ci, ushort_t orig_gpis)
{
    int         s, stereo;
    uchar_t     c, pan, rank, chksum, sum;
    caddr_t     addr;
    ushort_t    gpis;
    addr = ci->ci_addr[0];
    stereo = ci->ci_state & CARD_STEREO;
    pan = 0x40;
    rank = 0x01;    /* for 22050 Hz  */
    gpis = orig_gpis;
    /*
     *      Busy wait till Txd Fifo is empty
     *  The interrupt version is commenetd out below
     */
    #ifdef DEBUG
    cmn_err (CE_NOTE, “rapNoteOn: Waiting for Txd Fifo Empty, gpis = %x”,
            gpis);
    #endif
    while ( !(gpis & GPIS_TXD) ) {
        gpis = INPW(addr+GPIS);
        #ifdef DEBUG
        cmn_err (CE_NOTE, “rapNoteOn: Waiting ..new gpis = %x”, gpis);
        #endif
    }
    #ifdef DEBUG
    cmn_err (CE_NOTE, “rapNoteOn: Issuing a Note_On SysEx Cmd”);
    #endif
    /*   send Note_On   */
    c = 0xf0; OUTB(addr+MDTD, c);
    c = 0x41; OUTB(addr+MDTD, c);
    c = 0x10; OUTB(addr+MDTD, c);
    c = 0x56; OUTB(addr+MDTD, c);
    c = 0x12; OUTB(addr+MDTD, c);
    if ( stereo ) {
        c = 0x03; OUTB(addr+MDTD, c);
        c = 0x00; OUTB(addr+MDTD, c);
        c = 0x01; OUTB(addr+MDTD, c);
        sum = 0x03 + 0x01;
    }
    else {
        c = 0x02; OUTB(addr+MDTD, c);
        c = 0x00; OUTB(addr+MDTD, c);
        c = 0x0A+0x01; OUTB(addr+MDTD, c);
        sum = 0x02+0x0A+0x01;
    }
    c = 0x01; OUTB(addr+MDTD, c);
    c = 0x7F; OUTB(addr+MDTD, c);
    c = 0x7F; OUTB(addr+MDTD, c);
              OUTB(addr+MDTD, rank);
    sum += (0x01+0x7F+0x7F+rank);
    c = 0x40; OUTB(addr+MDTD, c);
    c = 0x00; OUTB(addr+MDTD, c);
    c = 0x40; OUTB(addr+MDTD, c);
              OUTB(addr+MDTD, pan);
    sum += (0x40+0x40+pan);
    /*  calculate the checksum  */
    chksum = (0x80 - (sum % 0x80)) & 0x7F;
    OUTB(addr+MDTD, chksum);
    c = 0xF7; OUTB(addr+MDTD, c);
    #ifdef DEBUG
    cmn_err (CE_NOTE, “rapNoteOn: Note_On Issued, chksum = %x”, chksum);
    #endif
}  /* end rapNoteOn  */

/**************************************************************************
 *                       r a p N o t e O f f
 **************************************************************************
 *  Name:       rapNoteOff
 *  Purpose:    Sends a MIDI Note_Off message.
 *              This code is taken from RAP-10 manual.
 *  Returns:    None.
 **************************************************************************/
static void
rapNoteOff ( cardInfo_t  *ci)
{
    int         s, stereo;
    uchar_t     pan, b, rank, sum, chksum;
    caddr_t     addr;
    ushort_t    gpis;
    addr = ci->ci_addr[0];
    stereo = ci->ci_state & CARD_STEREO;
    pan = 0x40;
    rank = 0x01;    /* for 22050 Hz  */
    #ifdef DEBUG
    cmn_err (CE_NOTE, “rapNoteOff: Waiting for Txd Empty”);
    #endif
    /*  wait till Txd is Empty   */
    gpis = INPW(addr+GPIS);
```

```
while ( !(gpis & GPIS_TXD) ) {
    us_delay(10);
    gpis = INPW(addr+GPIS);
    #ifdef DEBUG
    cmn_err (CE_NOTE, “rapNoteOff: Waiting ..new gpis = %x”, gpis);
    #endif
}
#ifdef DEBUG
cmn_err (CE_NOTE, “rapNoteOff: Issuing Note_Off”);
#endif
/*   send Note_On   */
OUTB(addr+MDTD, 0xF0);
OUTB(addr+MDTD, 0x41);
OUTB(addr+MDTD, 0x10);
OUTB(addr+MDTD, 0x56);
OUTB(addr+MDTD, 0x12);
if ( stereo ) {
    OUTB(addr+MDTD, 0x03);
    OUTB(addr+MDTD, 0x00);
    OUTB(addr+MDTD, 0x01);
    sum = 0x03 + 0x01;
}
else {
    OUTB(addr+MDTD, 0x02);
    OUTB(addr+MDTD, 0x00);
    OUTB(addr+MDTD, 0x0A+0x01);
    sum = 0x02 + 0x0A + 0x01;
}
OUTB(addr+MDTD, 0x00);
OUTB(addr+MDTD, 0x7F);
OUTB(addr+MDTD, 0x7F);
OUTB(addr+MDTD, 0x00);
sum += 0x7F + 0x7F;
OUTB(addr+MDTD, 0x40);
OUTB(addr+MDTD, 0x00);
OUTB(addr+MDTD, 0x40);
OUTB(addr+MDTD, pan);
sum += 0x40 + 0x40 + pan;
/*   calculate checksum   */
chksum = (0x80 - (sum % 0x80)) & 0x7F;
OUTB(addr+MDTD, chksum);
OUTB(addr+MDTD, 0x7F);
#ifdef DEBUG
cmn_err (CE_NOTE, “rapNoteOff: Note_On Issued, chksum = %x”, chksum);
#endif
```

```
}  /*  end rapNoteOff  */
/************************************************************************
 *                     r a p Z e r o D m a
 ************************************************************************
 *  Name:      rapZeroDma
 *  Purpose:   Zero outs DMA buffers.
 *  Returns:   None.
 ************************************************************************/
static void
rapZeroDma (cardInfo_t *ci, int bytes)
{
    caddr_t dmaL, dmaR;
    int     stereo, s;
    s = LOCK();
    stereo = ci->ci_state & CARD_STEREO;
    /*
     *     Zero out which half ?
     */
    if ( ci->di_which ) {
        dmaR = &dmaRight[DMA_HALF_SIZE];
        dmaL = &dmaLeft[DMA_HALF_SIZE];
        if ( bytes == DMA_BUF_SIZE ) {
            bytes = DMA_HALF_SIZE;
        }
    }
    /*  Zer out 1st half of dma buffers */
    else {
        dmaR = &dmaRight[0];
        dmaL = &dmaLeft[0];
    }
    #ifdef DEBUG
    cmn_err (CE_NOTE,
        "rapZeroDma: Zeroing out %s of Dma buffers in %s for %d bytes",
        (ci->di_which ? "2nd half":"1st half"),
         (stereo ? "Stereo":"Mono"),
         bytes);
    #endif
    bzero (dmaL, bytes);
    dki_dcache_wbinval (dmaL, (unsigned)bytes);
    if ( stereo ) {
        bzero (dmaR, bytes);
        dki_dcache_wbinval (dmaR, (unsigned)bytes);
    }
    UNLOCK(s);
}  /***  end rapZeroDma   ***/
```

```
/**********************************************************************
 *                   r a p R e l e a s e D m a
 **********************************************************************
 *  Name:      rapReleaseDma
 *  Purpose:   Releases Dma channel(s).
 *             Note that we access kernel’s Dma structure and later on
 *             a routine will be provided for us to avoid this.
 *  Returns:   None.
 **********************************************************************/
static void
rapReleaseDma (cardInfo_t *ci)
{
    /*   disable Eisa Dma  */
    #ifdef DEBUG
    cmn_err (CE_NOTE, “rapReleaseDma: Releasing Eisa Dma Chann %d”,
        ci->ci_dmaCh5);
    #endif
    eisa_dma_disable(0, ci->ci_dmaCh5);
    if ( ci->ci_state & CARD_STEREO ) {
        #ifdef DEBUG
        cmn_err (CE_NOTE, “rapReleaseDma: Releasing Eisa Dma Chann %d”,
                ci->ci_dmaCh6);
        #endif
        eisa_dma_disable(0, ci->ci_dmaCh6);
    }
} /***  end rapReleaseDma    ***/
/**********************************************************************
 *         r a p S e t A u t o I n i t
 **********************************************************************
 *  Name:      rapSetAutoInit
 *  Purpose:   sets Eisa DMA register for Autoinit. In Autoinit, DMA
 *             starts over from the beginning of the buffer again once it
 *             has transfered all bytes in the buffer.
 *  Returns:   None.
 **********************************************************************/
#define  EISA_MODE_REG  0xd6
#define  EISA_CH5       0x01
#define  EISA_CH6       0x02
#define  EISA_WRITE     0x04
#define  EISA_READ      0x08
#define  EISA_AUTO      0x10
static void
rapSetAutoInit( cardInfo_t  *ci, uchar_t what)
{
    uchar_t    b;
    #ifdef DEBUG
    cmn_err (CE_NOTE,
        “rapSetAutoInit: setting Autoinit DMA for %s, Eisa Addr = %x”,
        ( what == DI_DMA_PLAYING ? “Playback(D/A)” : “Record(A/D)” ),
        eisa_addr );
    #endif
    b = 0;
    if ( what == DI_DMA_PLAYING )
        b |= EISA_READ;        /*  Memory -> Device  */
    else
        b |= EISA_WRITE;       /*  Device -> Memory  */
    /*   Autoinit for Channel 5 - Demand Mode select is default    */
    b |= (EISA_AUTO | EISA_CH5);
    OUTB(eisa_addr+EISA_MODE_REG, b);
    /*   Autoinit for Channel 6 (if in stereo mode)  */
    if ( ci->ci_state & CARD_STEREO ) {
        b &= ~EISA_CH5;
        b |= EISA_CH6;
        OUTB(eisa_addr+EISA_MODE_REG, b);
    }
}  /***   End rapSetAutoInit    ***/
```

PART EIGHT

GIO Drivers

Chapter 18: GIO Device Drivers
Overview of the architecture of the GIO bus and the special services offered by the kernel to GIO drivers.

Chapter 18

GIO Device Drivers

The GIO bus is a synchronous, multiplexed address-data bus connecting high-speed devices to main memory and CPU for Silicon Graphics workstations. This chapter gives an overview of the GIO architecture, and describes the special kernel functions used to manage a device on the GIO bus. The main topics are as follows:

- “GIO Bus Overview” on page 512 describes the hardware implementation of the GIO bus.
- “Configuring a GIO Device” on page 514 discusses the use of the VECTOR line to describe a GIO device to IRIX.
- “Writing a GIO Driver” on page 515 discusses the work done in each entry point of a GIO device driver.
- “Memory Parity Workarounds” on page 526 covers an important hardware problem.
- “Example GIO Driver” on page 528 displays major parts of a driver for a hypothetical GIO device.

GIO Bus Overview

The GIO bus is a family of buses with different electrical requirements and form factors.

GIO Bus Varieties

Two varieties of GIO bus are found in workstations supported by IRIX 6.2:

- The GIO64 bus is a 64-bit, synchronous, multiplexed address-data bus that can run at speeds up to 33 MHz. It supports both 32- and 64-bit devices. GIO64 has two slightly different varieties: non-pipelined for internal system memory, and pipelined for graphics and pipelined GIO64 slot devices. The GIO64 bus is used in the Indigo^2 and Indigo^2 Maximum Impact systems.
- The GIO32-bis variety is a 32-bit version of the non-pipelined GIO64 bus or a GIO32 bus with pipelined control signals. This bus is found on R4000-based Indigo and Indy workstations.

An additional variety, GIO32 (without the *-bis* suffix) was used in Indigo workstations with R3000 processors. These workstations are not supported by IRIX 6.2.

Card Form Factors and Compatibility

GIO devices designed for the GIO32 (Indigo R3000) bus are compatible with GIO32-bis (Indy, Indigo^2), but are not compatible with GIO64 buses.

Devices designed for GIO32-bis can be used in GIO64 bus systems.

The Indigo and Indy systems have two GIO slots. The Indigo^2 has three physical sockets, but the lower two are paired as a single logical slot—the double socket provides extra electrical and mechanical support for heavy cards. The Indigo^2 Maximum Impact has four physical sockets, with each pair ganged as one logical slot. Thus all systems have two GIO slots, electrically speaking.

The form factor depends on the specific platform in which the device is installed. GIO32-bis devices can be either single or double-wide (that is, taking one or two sockets), while GIO64 boards are the size of an EISA board. Slots in Indigo^2 systems can accept either an EISA board or a GIO64 board. These two types of boards share common board dimensions but have different connectors for attaching to their respective buses.

GIO Bus Address Spaces

Each GIO device has a range of bus addresses to which it responds. These addresses correspond to device registers or on-board memory, depending on the GIO device.

The address range for a GIO bus device is determined in part by the slot number of the device. The hardware must be designed to determine which slot the device is in and make the appropriate adjustments to respond to that slot's address range.

Indigo and Indy systems provide two GIO address spaces known as *exp0* and *exp1*. Indigo^2 systems also have two slots, but support three GIO address spaces, referred to as *gfx*, *exp0*, and *exp1*. The *gfx* address space is used by the graphics card.

Table 18-1 shows the slot names and address spaces available on the Indigo, Indigo^2, and Indy platforms.

Table 18-1 GIO Slot Names and Addresses

Slot Name	Address	Indigo/Indy	Indigo^2
gfx	0x1f00 0000–0x1f3f ffff	N/A	Available
exp0	0x1f40 0000–0x1f5f ffff	Available	Available
exp1	0x1f60 0000–0x1f9f ffff	Available	Available

In 64-bit systems (Indigo2 Maximum Impact), two additional high-order bits are needed to select the physical address of the GIO space, so each of the above addresses is prefixed by 0x9000 0000. The 64-bit *gfx* space, for example, is at 0x9000 0000 1f00 0000.

GIO-bus devices use only one interrupt level — interrupt 1. Interrupts 0 and 2 are used by the graphics system and may not be used by GIO-bus devices.

Configuring a GIO Device

A GIO device is described to the system, and related to its device driver, using a VECTOR line in a file in the */var/sysgen/system* directory (see “Configuring a Kernel” on page 236).

GIO VECTOR Line

The VECTOR line for a GIO device uses the “old style” syntax documented in */var/sysgen/system/irix.sm*. The important elements in a VECTOR line for GIO are as follows:

bustype	Specified as *GIO* for GIO devices. The VECTOR statement can be used for other types of buses as well.
module	The base name of the device driver for this device, as used in the */var/sysgen/master.d* database (see “Master Configuration Database” on page 38 and “How Names Are Used in Configuration” on page 232).
adapter	Always 0, or omitted, for GIO, since there is never more than one GIO bus adapter in current systems.
ctlr	The “controller” number is simply an integer parameter that is passed to the device driver at boot time. It can be used, for example, to specify a logical unit number.
base	Device base address, as shown in Table 18-1.
probe or *exprobe*	Specify a hardware test that can be applied at boot time to find out if the device exists.

You use the *probe* or *exprobe* parameter to program a test for the existence of the device at boot time. If the device does not respond (because it is offline or because it has been removed from the system), the *lboot* command will not invoke the device driver for this device. This facility is used in distributed */var/sysgen/system/irix.sm* files in order to choose between the graphics board in slot *gfx* or in slot *exp0*.

Writing a GIO Driver

GIO bus devices are controlled only from kernel-level drivers; there is no provision for memory-mapping GIO devices into user-level address spaces.

A GIO device driver is a kernel-level driver compiled, linked, and loaded into the kernel as described in Chapter 10, "Building and Installing a Driver." A GIO driver can call on the kernel functions described in Chapter 8, "Structure of a Kernel-Level Driver." However, a GIO driver has to use some special features in its *pfx***edtinit()** and *pfx***intr()** entry points.

GIO-Specific Kernel Functions

Three GIO-specific functions are used in setting up a GIO device. They are only documented here; there are no reference pages for them. The functions are declared as external in the CPU-specific include files *sys/IP20.h* and *sys/IP22.h*. (When compiling for a POWER Indigo^2, which uses an IP26 CPU, you include *sys/IP22.h* as well as *sys/IP26.h*.)

Function setgiovector()

The **setgiovector()** function registers an interrupt service function for a GIO device interrupt with the kernel's interrupt dispatcher, or unregisters one. The function prototype is

```
void
setgiovector(int level, int slot,
            void (*func)(__psint_t, struct eframe_s *),
            __psint_t arg);
```

The arguments are as follows:

level	The interrupt level; must be GIO_INTERRUPT_1 for all devices except the graphics board.
slot	The slot number, 0 or 1.
func	The address of the interrupt handling function (typically the *pfx***intr()** entry point of the device driver), or else NULL to unregister.
arg	A "pointer-sized integer" value to be passed as the first argument of the interrupt handler when it is invoked.

Note: If either the *level* or *slot* number is out of range, **setgiovector()** issues an error message with the CE_PANIC level, causing a kernel panic.

When *func* is not NULL, the specified function is registered to receive interrupts at the given *level* from the given *slot*. When an interrupt occurs, the function is called with two arguments. The first is the value specified as *arg*, a "pointer-sized integer," typically the address of device-specific information. The second is the interrupt registers. The structure *eframe_s* is declared in *sys/reg.h*. However, this structure is of no interest.

This function can be used with a NULL for the *func* argument to unregister an interrupt routine that was previously registered. You must unregister an interrupt handler in a loadable device driver prior to unloading, when called at the *pfx***unload()** entry point (see "Entry Point unload()" on page 167).

Function setgioconfig()

The function **setgioconfig()** configures the GIO slot for a particular use. The function prototype is

```
void
setgioconfig(int slot, int flags);
```

The arguments are as follows:

slot	The slot number, 0 or 1.
flags	A set of bit-flags from the constants GIO_ARB_* declared in *sys/mc.h*.

Note: If the *slot* number is out of range, **setgioconfig()** either issues an error message with the CE_PANIC level or suffers an assertion failure, causing a kernel panic.

The flags that can be combined to make the *flags* argument are

GIO64_ARB_EXP0_SIZE_64	Configure for 64-bit transfers; otherwise transfers will be 32-bit.
GIO64_ARB_EXP0_RT	Configure as a real-time device; otherwise it will be a long burst device.
GIO64_ARB_EXP0_MST	Configure as a bus master; otherwise it will be a slave.
GIO64_ARB_EXP0_PIPED	Configure slot as a pipelined device, otherwise it will be a non-pipelined device. For Indigo^2 systems, this must be set.

splgio0, splgio1, splgio2

Three functions can be used to set the processor interrupt mask to block GIO-bus interrupts. As of IRIX 6.2, the only systems that support the GIO bus are uniprocessor systems, in which **spl()**-type functions are effective. When writing a device driver that might be ported to a multiprocessor, you should avoid functions of this type, and use other means of getting mutual exclusion (see “Priority Level Functions” on page 213).

The prototypes of the GIO **spl()** functions are

```
long splgio0();
long splgio1();
long splgio2();
```

Devices other than graphics drivers would typically only have a reason to use **splgio1()**, because 1 is the interrupt level of non-graphics GIO devices.

GIO Driver edtinit() Entry Point

The device driver specified by the *module* parameter is invoked at its *pfx***edtinit()** entry point, where it receives most of the other information specified in the VECTOR statement (see “Entry Point edtinit()” on page 144).

The *pfx***edtinit()** entry point is called only in response to a VECTOR line. However, a VECTOR line need not contain a *probe* or *exprobe* test of the hardware.

The driver should not assume that its hardware exists; instead it should use the **badaddr()** kernel function to test the addresses passed in the edt_t object to make sure they are usable (see “Testing Device Physical Addresses” on page 195).

Example 18-1 displays a skeleton version of the *pfx***edtinit()** entry point of a hypothetical GIO device driver. This example uses GIO-specific functions that are described in a following section, “GIO-Specific Kernel Functions” on page 515.

Example 18-1 GIO Driver edtinit() Entry Point

```
#include <sys/edt.h>
void
hypoth_edtinit(register struct edt *e)
{
   int slot, val;
   /* Check to see if the device is present */
   if(badaddr_val(e->e_base, sizeof(int), &val) ||
         (val && GBD_MASK) != GBD_BOARD_ID) {
      if (showconfig)
         cmn_err (CE_CONT,
            "gbdedtinit: board not installed.");
         return;
   }
   /* figure out slot from base on VECTOR line in
   /* system file*/
   if(e->e_base == (caddr_t)0xBf400000)
      slot = GIO_SLOT_0;
   else if(e->e_base == (caddr_t)0xBF600000)
      slot = GIO_SLOT_1;
   else {
      cmn_err (CE_NOTE,
      "ERROR from edtinit: Bad base address %x\n",e->e_base);
      return;
   }
#ifdef IP20   /* For Indigo R4000, set up board as a
                 realtime bus master */
   setgioconfig(slot,GIO64_ARB_EXP0_RT|GIO64_ARB_EXP0_MST);
#endif
#ifdef (IP22|IP26)   /* For Indy, Indigo2, set up board as a
                        pipelined realtime bus master  */
   setgioconfig(slot,GIO64_ARB_EXP0_RT|GIO64_ARB_EXP0_PIPED);
#endif
   /* Save the device addresses, because
    * they won’t be available later.
    */
   gbd_device[slot == GIO_SLOT_0 ? 0 : 1] =
            (struct gbd_device *)e->e_base;
   gbd_memory[slot == GIO_SLOT_0 ? 0 : 1] =
            (char *)e->e_base2;
              /* Where "unit_#" is any parameter passed to
              /* the interrupt handler (gbdintr) */
   setgiovector(GIO_INTERRUPT_1,slot,gbdintr,unit_#);
}
```

GIO Driver Interrupt Handler

A GIO driver must contain an interrupt entry point. It does not have to be named *pfx***intr()** because it is registered using the **giosetvector()** function.

When the device generates an interrupt, the general GIO interrupt handler calls your driver's registered interrupt routine and passes it the argument that was specified to **setgiovector()** as the argument. This is typically a unit number, or the address of a device-specific information structure.

Within the interrupt routine, the driver must wake up the sleeping upper-half process, if one is waiting on the transfer to complete. In a block device driver, the interrupt routine calls **iodone()** to indicate that a block type I/O transfer for the buffer is complete (see "Waiting for Block I/O to Complete" on page 217).

Using PIO

Programmed I/O (PIO) is used to transfer small amounts of data between memory and device registers. PIO is typically used for control functions and to set up device registers prior to DMA (see "Using DMA" on page 521).

PIO can be as simple as storing a variable into a bus address (as passed to the *pfx***edtinit()** entry point). Example 18-2 displays fragmentary code of a hypothetical character device driver for a GIO device that controls a printer. This *pfx***write()** entry point copies data from the user address space to device memory using the **uiomove()** function (see "Transferring Data Through a uio_t Object" on page 194). Then it stores an explicit command in the device to start it, and sleeps until the device interrupts.

Example 18-2 Hypothetical PIO Routine for GIO

```
/* device write routine entry point (for character devices)*/
int
hypoth_write(dev_t dev, uio_t *uio)
{
   int unit = geteminor(dev)&1;
   int size, err=0, s;
   /* while there is data to transfer */
   while((size=uio->uio_resid) > 0) {
      /* Transfer no more than GBD_MEMSIZE bytes */
      size = size < GBD_MEMSIZE ? size : GBD_MEMSIZE;
      /* decrements size, updates uio fields, copies data */
      if(err=uiomove(gbd_memory[unit], size, UIO_WRITE, uio))
         break;
      /* prevent interrupts until we sleep */
      s = splgio1();
      /* Transfer is complete; start output */
      gbd_device[unit]->count = size;
      gbd_device[unit]->command = GBD_GO;
      gbd_state[unit] = GBD_SLEEPING;
      while (gbd_state[unit] != GBD_DONE) {
         sleep(&gbd_state[unit], PRIBIO);
      }
      /* restore the interrupt level after waking up */
      splx(s);
   }
   return err;
}
```

An expression like *gdb_device[unit]->command=GBD_GO* represents storing a command value in a device register. Presumably the *gdb_device* array is set up with a device address for each slot in the *pfx***edtinit()** entry point.

The code in Example 18-2 uses **splgio1()** to block an interrupt from occurring after it has started the device in operation and before it has entered the blocked state using **sleep()**. If this was not done, there is a small window of time during which an interrupt could occur and be handled before the upper-half routine had begun sleeping. Then it would sleep forever.

An alternate way to handle this same situation in a multiprocessor system is to use a mutual-exclusion lock to get exclusive use of the device registers, and a synchronization variable to wait for the interrupt (see "Using Synchronization Variables" on page 220).

Using DMA

DMA access achieves higher throughput than PIO when the device transfers more than a few words of data at a time. DMA is typically set up by programming device registers with the target address and length, and leaving the device to generate a series of stores or loads from memory. The details of device control are hardware-dependent.

The direction of a DMA transfer is measured with respect to the device, which operates independently. A DMA operation is either a DMA *read* (of memory data out to the device) or a DMA *write* (by the device, of data into memory).

DMA buffers should be cache-aligned in memory (see “Setting Up a DMA Transfer” on page 198). Prior to a DMA read, the driver should make sure that cached data has been written to memory using **dki_cache_wb()**. Prior to a DMA write, the driver should make sure the CPU knows that cached data is invalid (or is about to become invalid) using **dki_cache_inval()** (see “Managing Memory for Cache Coherency” on page 200).

DMA To Multiple Pages

Some devices can perform DMA only in a single transfer of data to a range of contiguous addresses. Such a device must be programmed separately for each individual page of data. Other devices are capable of transferring a series of page units to different addresses; that is, they support “scatter/gather” capability. These devices can be programmed once to transfer an entire buffer of data, regardless of whether the buffer spans multiple pages.

In either case, the *pfx***strategy()** entry point of a block device driver must calculate the physical addresses of a series of one or more pages, and program them into the device. When the device does not support scatter/gather, it is set up and started on each page of data individually, with an interrupt after each page. When the device supports scatter/gather, it is programmed with a list of page addresses all at once.

DMA With Scatter/Gather Capability

Example 18-3 shows the skeleton of a *pfx***strategy()** entry point for a block device driver for a hypothetical GIO device that supports scatter/gather capability.

Example 18-3 Strategy Code for Hypothetical Scatter/Gather GIO Device

```
/* Actual device setup for DMA, etc., if your board has
 * hardware scatter/gather DMA support.
 * Called from the hypo_write() routine via physio().
 */
void
hypo_strategy(struct buf *bp)
{
   int unit = geteminor(bp->b_dev)&1;
   int npages;
   volatile unsigned *sgregisters; /* ->device regs */
   int i, v_addr;
   /* MISSING: any checking for initial state. */
   /* Get address of the scatter/gather registers */
   sgregisters = gbd_device[unit]->sgregisters;
   /* Get the kernel virtual address of the data; note
    * b_dmaaddr may be NULL if the BP_ISMAPPED(bp) macro
    * indicates false; in that case, the field bp->b_pages
    * is a pointer to a linked list of pfdat structure
    * pointers; that saves creating a virtual mapping and
    * then decoding that mapping back to physical addresses.
    * BP_ISMAPPED will never be false for character devices,
    * only block devices.
    */
   if(!BP_ISMAPPED(bp)) {
      cmn_err(CE_WARN,
         "gbd driver can’t handle unmapped buffers");
      bioerror(bp, EIO);
      biodone(bp);
      return;
   }
v_addr = bp->b_dmaaddr;
   /* Compute number of pages affected by this request.
    * The numpages() macro (sysmacros.h) returns the number of pages
    * that span a given length starting at a given address, allowing
    * for partial pages.  Unrealistically, we limit this to the
    * number of scatter/gather registers on board.
    * Note that this sample driver doesn’t handle the
    * case of requests > than # of registers!
    */
   npages = numpages (v_addr, bp->b_bcount);
   if(npages > GBD_NUM_DMA_PGS) {
       bp->b_resid = IO_NBPP * (npages - GBD_NUM_DMA_PGS);
       npages = GBD_NUM_DMA_PGS;
```

```
            cmn_err(CE_WARN,
                "request too large, only %d pages max", npages);
        }
        /* Translate the virtual address of each page to a
         * physical page number and load it into the next
         * scatter/gather register.
         * btop() converts the byte value to a page value after
         * rounding down the byte value to a full page.
         */
        for (i = 0; i < npages; i++) {
            *sgregisters++ = btop(kvtophys(v_addr));
            v_addr += IO_NBPP;
        }
        /* Program the device for input or output */
        if ((bp->b_flags & B_READ) == 0)
            gbd_device[unit]->direction = GBD_WRITE;
        else
            gbd_device[unit]->direction = GBD_READ;
    /* Start the device going and return. The caller, either a
     * file system or uiophysio(), waits for the iodone() call
     * from the interrupt routine.
     */
        gbd_device[unit]->command = GBD_GO;
    }
```

DMA Without Scatter/Gather Support

When the GIO device does not provide scatter/gather capability, the driver must program the transfer of each memory page individually, ensuring that the device does not attempt to store or load across a page boundary. The usual method is as follows:

- In the *pfx***strategy()** routine, save the address of the *buf_t* for use by the *pfx***intr()** entry point.
- In the *pfx***strategy()** routine, program the device to transfer the data for the first page, and start the device going.
- In the *pfx***intr()** entry point, calculate the number of bytes remaining to transfer. If the count is zero, signal **biodone()**. If the count is nonzero, program the device to transfer the next page of data.

Under this design, there is no explicit loop over the successive pages of the transfer visible in the code. The loop is implicit in the fact that the *pfx***intr()** entry point starts a new transfer, and so will be called again, until the transfer is complete.

Example 18-4 shows the code of the *pfx***strategy()** routine for a hypothetical GIO device without scatter/gather.

Example 18-4 Strategy() Code for GIO Device Without Scatter/Gather

```
/* Actual device setup for DMA, etc., when the board
 * does NOT have hardware scatter/gather DMA support.
 * Called from the hypo_write() routine via physio().
 */
void
hypo_strategy(struct buf *bp)
{
   int unit = geteminor(bp->b_dev)&1;
   /* MISSING: any checking for initial state. */
   /* Get the kernel virtual address of the data; note
   * b_dmaaddr may be NULL if the BP_ISMAPPED(bp) macro
   * indicates false; in that case, the field bp->b_pages
   * is a pointer to a linked list of pfdat structure
   * pointers; that saves creating a virtual mapping and
   * then decoding that mapping back to physical addresses.
   * BP_ISMAPPED will never be false for character devices,
   * only block devices.
   */
   if(!BP_ISMAPPED(bp)) {
      cmn_err(CE_WARN,
         "gbd driver can’t handle unmapped buffers");
      bioerror(bp, EIO);
      biodone(bp);
      return;
   }
   /* Save ->buf_t where interrupt handler can find it */
   gbd_curbp[unit] = bp;
   /*
   * Initialize the current transfer address and count.
   * The first transfer should finish the rest of the
   * page, but do no more than the total byte count.
   */
   gbd_curaddr[unit] = bp->b_dmaaddr;
   gbd_totcount[unit] = bp->b_count;
   gbd_curcount[unit] = IO_NBPP-
      ((unsigned int)gbd_curaddr[unit] & (IO_NBPP-1));
   if (bp->b_count < gbd_curcount[unit])
      gbd_curcount[unit] = bp->b_count;
   /* Tell the device starting physical address, count,
   * and direction */
   gbd_device[unit]->startaddr = kvtophys(gbd_curaddr[unit]);
   gbd_device[unit]->count = gbd_curcount[unit];
   if (bp->b_flags & B_READ) == 0)
      gbd_device[unit]->direction = GBD_WRITE;
   else
      gbd_device[unit]->direction = GBD_READ;
   gbd_device[unit]->command = GBD_GO;   /* start DMA */
   /* and return; upper layers of kernel wait for iodone(bp) */
}
```

An alternate design might seem conceptually simpler: to put an explicit loop in the *pfx***strategy()** routine, starting each page transfer and waiting on a semaphore until the *pfx***intr()** routine is called. Such a design keeps the complexity in the *pfx***strategy()** routine, making the *pfx***intr()** routine as simple as possible. However, it has a high cost in performance because the *pfx***strategy** routine must wake up and be dispatched for every page.

Scatter/gather programming can be simplified by the use of the **sgset()** function, which calculates the physical addresses and lengths for each page in the transfer (see the sgset(D3) reference page). The **sgset()** function is limited to use with hardware that uses a fixed mapping of bus addresses to memory addresses, which is the case in the workstations supporting GIO. For example, **sgset()** cannot be used in the Challenge or Onyx line; it always returns -1 in those systems.

Memory Parity Workarounds

Beginning with IRIX 5.3, parity checking is enabled on the SysAD bus, which connects the CPU to memory in workstations that use the GIO bus (see Figure 18-1). Unfortunately, with certain GIO cards, errors can occur if memory reads complete before the Memory Controller (MC) finishes calculating parity.

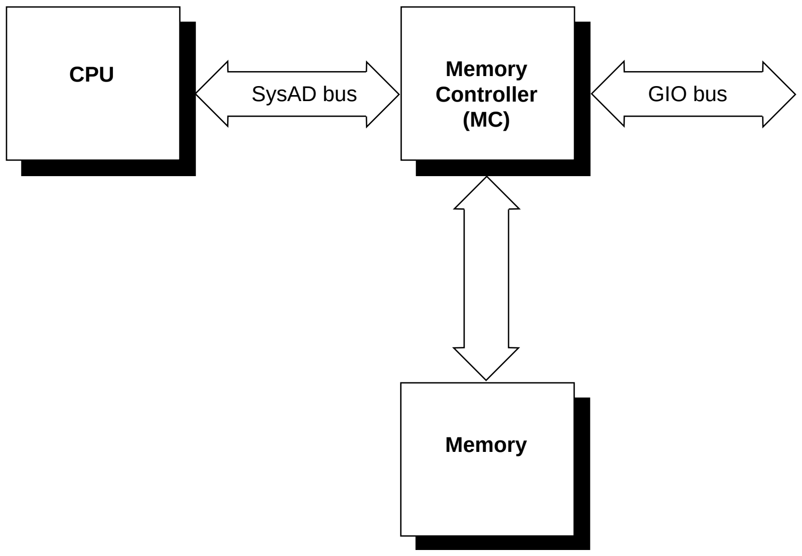

Figure 18-1 The SysAD Bus in Relation to GIO

Some GIO cards do not drive all 32 GIO data lines during CPU PIO reads. These reads from the GIO card are either 8-bit or 16-bit transfers, so the lines are left floating. The problem is that to generate parity bits for the SysAD bus, the Memory Controller (MC) must calculate parity for all 32 bits. Since the calculation must occur before the CPU read completes, it is possible that one (or more) of the floating bits may change while parity is being calculated. Thus, when the CPU read completes, it may be received as a parity error on the SysAD bus.

Note: Diagnosis is complicated by the fact that this problem may not show up on every transaction. It occurs only when one of the data lines that is left floating happens to change state between the start of the MC parity calculation and the completion of the CPU read. A device and its driver can appear to function correctly for some time before the problem occurs.

When writing a driver for a GIO card that does not drive all 32 data lines, you must either disable SysAD parity checking completely, or disable it during the time your driver is performing PIO transfers. Three kernel functions are supplied for these purposes; none of them take arguments.

- **is_sysad_parity_enabled()** returns a nonzero value if SysAD parity checking is enabled.
- **disable_sysad_parity()** turns off parity checking on the SysAD bus.
- **enable_sysad_parity()** returns SysAD parity checking to normal.

To completely disable SysAD parity checking removes the system's ability to recover from a parity error in main memory. As a short-term fix, a driver could simply call **disable_sysad_parity()** in the *pfx***init()** or *pfx***edtinit()** entry point.

It is much better to disable parity checking only during the time the device is being used. The advantage here is that the software recovery procedures for memory parity errors are almost always in effect.

To selectively disable parity checking, put wrappers around your driver's PIO transactions to disable SysAD parity checking before a transfer, and to re-enable it after the PIO completes. Example 18-5 shows a skeleton of such a wrapper.

Example 18-5 Disabling SysAD Parity Checking During PIO

```
void
do_PIO_without_parity()
{
   int was_enabled = is_sysad_parity_enabled();
   if (was_enabled) disable_sysad_parity();
/* do driver PIO transfers */
if (was_enabled) enable_sysad_parity();
}
```

The reason that the function in Example 18-5 saves the current state of parity, and only re-enables parity when it was enabled on entry, is that parity checking could have been turned off in some higher-level routine. For example, an interrupt handler could be entered during execution of a device driver function that disables parity checking. If the interrupt handler turned parity checking back on regardless of its former state, errors would occur.

Example GIO Driver

The code in Example 18-6 displays a complete device driver for a hypothetical device. The driver prefix is *gbd* (for "GIO board").

Example 18-6 Complete Driver for Hypothetical GIO Device

```
/* Source for a hypothetical GIO board device; it can be compiled for
 * devices that support DMA (with or without scatter gather support),
 * or for PIO mode only.  This version is designed for IRIX 6.2 or later.
 * Dave Olson, 5/93.  6.2 port by Dave Cortesi 9/95.
*/

/* Compilation: Define the environment variable CPUBOARD as IP20, IP22,
 * or IP26 (the only GIO platforms). Then include the build rules from
 * /var/sysgen/Makefile.kernio to set $CFLAGS including:
#   _K32U32     kernel in 32 bit mode running only 32 bit binaries
#   _K64U64     kernel in 64 bit mode running 32/64 bit binaries (IP26)
#   -DR4000     R4000 machine (IP20, IP22)
#   -DTFP       R8000 machine (IP26)
#   -G 8        global pointer set to 8 (GIO drivers cannot be loadable)
#   -elf        produce an elf executable
 */

/* the following definitions choose between PIO vs DMA supporting
 * boards, and if DMA is supported, whether hardware scatter/gather
 * is supported. */
#define GBD_NODMA       0   /* non-zero for PIO version of driver */
#define GBD_NUM_DMA_PGS 8   /* 0 for no hardware scatter/gather
                             * support, else number of pages of
                             * scatter/gather per request */
#include <sys/param.h>
#include <sys/systm.h>
#include <sys/cpu.h>
#include <sys/buf.h>
#include <sys/cred.h>
#include <sys/uio.h>
#include <sys/ddi.h>
#include <sys/errno.h>
#include <sys/cmn_err.h>
#include <sys/edt.h>
#include <sys/conf.h> /* for flags D_MP */

/* gbd (for Gio BoarD) is the driver prefix, specified in the
 * file /var/sysgen/master.d/gbd and in VECTOR module=gbd lines.
 * This driver is multiprocessor-safe (even though no GIO platform
 * is a multiprocessor).
 */
int gbddevflags = D_MP;

/* these defines and structures defining the (hypothetical) hardware
 * interface would normally be in a separate header file
 */
#define GBD_BOARD_ID    0x75
#define GBD_MASK        0xff    /* use 0xff if using only first byte
                                 * of ID word, use 0xffff if using
                                 * whole ID word
                                 */
#define GBD_MEMSIZE 0x8000
/* command definitions */
#define GBD_GO 1
/* state definitions */
#define GBD_SLEEPING 1
#define GBD_DONE 2
/* direction of DMA definitions */
#define GBD_READ 0
#define GBD_WRITE 1
/* status defines */
#define GBD_INTR_PEND   0x80

/* device register interface to the board */
typedef struct gbd_device {
    __uint32_t  command;
    __uint32_t  count;
    __uint32_t  direction;
    __uint32_t  offset;
    __uint32_t  status; /* errors, interrupt pending, etc. */
#if (!GBD_NODMA)        /* if hardware DMA */
#if (GBD_NUM_DMA_PGS)   /* if hardware scatter/gather */
    /* board register points to array of GBD_NUM_DMA_PGS target
     * addresses in board memory.  Board can relocate the array
     * by changing the content of sgregisters.
     */
    volatile paddr_t    *sgregisters;
#else                   /* dma to contiguous segment only */
    paddr_t     startaddr;
#endif
#endif
} gbd_regs;
```

```
static struct gbd_info {
    gbd_regs    *gbd_device;    /* ->board regs */
    char        *gbd_memory;    /* ->on-board memory */
    sema_t      use_lock;       /* upper-half exclusion from board */
    lock_t      reg_lock;       /* spinlock for interrupt exclusion */
#if GBD_NODMA
    int         gbd_state;      /* transfer state of PIO driver */
    sv_t        intr_wait;      /* sync var for waiting on intr */
#else /* DMA supported somehow */
    buf_t       *curbp;         /* current buf struct */
#if (0 == GBD_NUM_DMA_PGS)  /* software scatter/gather */
    caddr_t     curaddr;        /* current address to transfer */
    int         curcount;       /* count being transferred */
    int         totcount;       /* total size this transfer */
#endif
#endif
} gbd_globals[2];

void gbdintr(int, struct eframe_s *);

/* early device table initialization routine. Validate the values
 * from a VECTOR line and save in the per-device info structure.
 */
void
gbdedtinit(register edt_t *e)
{
    int slot;           /* which slot this device is in */
    __uint32_t val = 0; /* board ID value */
    register struct gbd_info *inf;

    /* Check to see if the device is present */
    if(!badaddr(e->e_base, sizeof(__uint32_t)))
        val = *(__uint32_t *)(e->e_base);
    if ((val && GBD_MASK) != GBD_BOARD_ID) {
        if (showconfig) {
            cmn_err (CE_CONT, “gbdedtinit: board not installed.”);
        }
        return;
    }
    /* figure out slot from VECTOR base= value */
    if(e->e_base == (caddr_t)0xBF400000)
        slot = GIO_SLOT_0;
    else if(e->e_base == (caddr_t)0xBF600000)
        slot = GIO_SLOT_1;
    else {
        cmn_err (CE_NOTE,
        “ERROR from edtinit: Bad base address %x\n”, e->e_base);
        return;
    }
#if IP20 /* for Indigo R4000, set up board as a realtime bus master */
    setgioconfig(slot,GIO64_ARB_EXP0_RT | GIO64_ARB_EXP0_MST);
#endif
#if (IP22|IP26) /* for Indigo2, set up as a pipelined, realtime bus master */
    setgioconfig(slot,GIO64_ARB_EXP0_RT | GIO64_ARB_EXP0_MST);
#endif
    /* Initialize the per-device (per-slot) info, including the
     * device addresses from the edt_t.
     */
    inf = &gbd_globals[GIO_SLOT_0 ? 0 : 1];
    inf->gbd_device = (struct gbd_device *)e->e_base;
    inf->gbd_memory = (char *)e->e_base2;
    initsema(&inf->use_lock,1);
    spinlock_init(&inf->reg_lock,NULL);
    setgiovector(GIO_INTERRUPT_1,slot,gbdintr,0);
    if (showconfig) {
        cmn_err (CE_CONT, “gbdedtinit: board %x installed\n”, e->e_base);
    }
}
/* OPEN: minor number used to select slot. Merely test that
 * the device was initialized.
 */
/* ARGSUSED */
gbdopen(dev_t *devp, int flag, int otyp, cred_t *crp)
{
    if(! (gbd_globals[geteminor(*devp)&1].gbd_device) )
        return ENXIO;   /* board not present */
    return 0;   /* OK */
}
/* CLOSE: Nothing to do. */
/* ARGSUSED */
gbdclose(dev_t dev, int flag, int otyp, cred_t *crp)
{
    return 0;
}
#if (GBD_NODMA) /***** Non-DMA, therefore character, device ******/
/* WRITE: for character device using PIO */
/* READ entry point same except for direction of transfer */
int
gbdwrite(dev_t dev, uio_t *uio)
{
    int unit = geteminor(dev)&1;
    struct gbd_info *inf = &gbd_globals[unit];
    int size, err=0, lk;
    /* Exclude any other top-half (read/write) user */
    psema(&inf->use_lock,PZERO)
    /* while there is data to transfer */
    while((size=uio->uio_resid) > 0) {

        /* Transfer no more than GBD_MEMSIZE bytes per operation */
        size = (size < GBD_MEMSIZE) ? size : GBD_MEMSIZE;

        /* Copy data from user-process memory to board memory.
         * uiomove() updates uio fields and copies data
         */
        if(! (err=uiomove(inf->gbd_memory, size, UIO_WRITE, uio)) )
            break;

        /* Block out the interrupt handler with a spinlock, then
         * program the device to start the transfer.
         */
        lk = mutex_spinlock(&inf->reg_lock);
        inf->gbd_device->count = size;
        inf->gbd_device->command = GBD_GO;
        inf->gbd_state = GBD_INTR_PEND; /* validate an interrupt */
        /* Give up the spinlock and sleep until gdbintr() signals */
        sv_wait(&inf->intr_wait,PZERO,&inf->reg_lock,lk);
    } /* while(size) */
    vsema(&inf->use_lock); /* let another process use board */
    return err;
}
/* INTERRUPT: for PIO only board */
/* ARGSUSED1 */
void
gbdintr(int unit, struct eframe_s *ef)
{
    register struct gbd_info *inf = &gbd_globals[unit];
    int lk;
    /* get exclusive use of device regs from upper-half */
    lk = mutex_spinlock(&inf->reg_lock);

    /* if the interrupt is not from our device, ignore it */
    if(inf->gbd_device->status & GBD_INTR_PEND) {
        /* MISSING: test device status, clean up after interrupt,
         * post errors into inf->state for upper-half to see.
         */
        /* Provided the upper-half expected this, wake it up */
        if (inf->gbd_state & GBD_INTR_PEND)
            sv_signal(&inf->intr_wait);
    }
    mutex_spinunlock(&inf->reg_lock,lk);
}

#else /******** DMA version of driver ************/

void gbd_strategy(struct buf *);

/* WRITE entry point (for character driver of DMA board).
 * Call uiophysio() to set up and call gbd_strategy routine,
 * where the transfer is actually done.
*/
int
gbdwrite(dev_t dev, uio_t *uiop)
{
    return uiophysio((int (*)())gbd_strategy, 0, dev, B_WRITE, uiop);
}
/* READ entry point same except for direction of transfer */
#if GBD_NUM_DMA_PGS > 0

/* STRATEGY for hardware scatter/gather DMA support.
 * Called from gbdwrite()/gbdread() via physio().
 * Called from file-system/paging code directly.
 */
void
gbd_strategy(register struct buf *bp)
{
    int unit = geteminor(bp->b_edev)&1;
    register struct gbd_info *inf = &gbd_globals[unit];
    register gbd_regs *regs = inf->gbd_device;
    volatile paddr_t *sgregisters;
    int npages;
    int i, lk;
    caddr_t v_addr;

    /* Get the kernel virtual address of the data. Note that
     * b_dmaaddr is NULL when the  BP_ISMAPPED(bp) macro
     * indicates false; in that case, the field bp->b_pages
     * is a pointer to a linked list of pfdat structure
     * pointers; that saves creating a virtual mapping and
     * then decoding that mapping back to physical addresses.
     * BP_ISMAPPED will never be false for character devices,
     * only block devices.
     */
     if(!BP_ISMAPPED(bp)) {
        cmn_err(CE_WARN, “gbd driver can’t handle unmapped buffers”);
        bp->b_flags |= B_ERROR;
        iodone(bp);
        return;
    }
    v_addr = bp->b_dmaaddr;

    /* Compute number of pages affected by this request.
     * The numpages() macro (sysmacros.h) returns the number of pages
     * that span a given length starting at a given address, allowing
     * for partial pages.  Unrealistically, we limit this to the
     * number of scatter/gather registers on board.
     * Note that this sample driver doesn’t handle the
     * case of requests > than # of registers!
     */
    npages = numpages (v_addr, bp->b_bcount);
    if(npages > GBD_NUM_DMA_PGS) {
        bp->b_resid = IO_NBPP * (npages - GBD_NUM_DMA_PGS);
        npages = GBD_NUM_DMA_PGS;
        cmn_err(CE_WARN,
            “request too large, only %d pages max”, npages);
    }

    /* Get exclusive upper-half use of device. The sema is released
     * wherever iodone() is called, here or in the int handler.
     */
    psema(&inf->use_lock,PZERO)
    inf->curbp = bp;

    /* Get exclusive use of the device regs, blocking the int handler */
    lk = mutex_spinlock(&inf->reg_lock);

    /* MISSING: set up board to transfer npages discreet segments. */
    /* Get address of the scatter-gather registers */
    sgregisters = regs->sgregisters;

    /* Provide the beginning byte offset and count to the device. */
    regs->offset = io_poff(bp->b_dmaaddr); /* in immu.h */
    regs->count = (IO_NBPP - inf->gbd_device->offset)
                    + (npages-1)*IO_NBPP;

    /* Translate the virtual address of each page to a
```

```
         * physical page number and load it into the next
         * scatter-gather register.  The btoct(K) macro
         * converts the byte value to a page value after
         * rounding down the byte value to a full page.
         */
         for (i = 0; i < npages; i++) {
            *sgregisters++ = btoct(kvtophys(v_addr));
            v_addr += IO_NBPP;
        }

        if ((bp->b_flags & B_READ) == 0)
            regs->direction = GBD_WRITE;
        else
            regs->direction = GBD_READ;
        regs->command = GBD_GO; /* start DMA */

        /* release use of the device regs to the interrupt handler */
        mutex_spinunlock(inf->reg_lock,lk);

        /* and return; upper layers of kernel wait for iodone(bp) */
    }

    /* INTERRUPT: for hardware DMA support. This is over-simplified
     * because the above strategy routine never accepts a transfer
     * larger than the device can handle in a single operation.
     */
    /* ARGSUSED1 */
    void
    gbdintr(int unit, struct eframe_s *ef)
    {
        register struct gbd_info *inf = &gbd_globals[unit];
        register gbd_regs *regs = inf->gbd_device;
        int error = 0;
        int lk;

        /* get exclusive use if device regs from upper-half */
        lk = mutex_spinlock(&inf->reg_lock);

        /* If interrupt was not from this device, exit quick */
        if (! (regs->status & GBD_INTR_PEND) ) {
            mutex_spinunlock(&inf->reg_lock,lk);
            return;
        }

        /* MISSING: read board registers, clear interrupt,
         * and note any errors in the “error” variable. */
        if(error)
            inf->curbp->b_flags |= B_ERROR;

        /* release lock on exclusive use of device regs */
        mutex_spinunlock(&inf->reg_lock,lk);

        /* wake up any kernel/file-system waiting for this I/O */
        iodone(inf->curbp);

        /* unlock use of device to other upper-half driver code */
        vsema(&inf->use_lock);
    }

    #else /******  GBD_NUM_DMA_PGS == 0; no hardware scatter/gather ******/

    /* STRATEGY: for software-controlled scatter/gather.
     * Called from the gbdwrite() routine via uiophysio().
     */
    void
    gbd_strategy(struct buf *bp)
    {
        int unit = geteminor(bp->b_edev)&1;
        register struct gbd_info *inf = &gbd_globals[unit];
        register gbd_regs *regs = inf->gbd_device;
        int lk;

        /* Get the kernel virtual address of the data; note
         * b_dmaaddr may be NULL if the  BP_ISMAPPED(bp) macro
         * indicates false; in that case, the field bp->b_pages
         * is a pointer to a linked list of pfdat structure
         * pointers; that saves creating a virtual mapping and
         * then decoding that mapping back to physical addresses.
         * BP_ISMAPPED will never be false for character devices,
         * only block devices.
         */
         if(!BP_ISMAPPED(bp)) {
            cmn_err(CE_WARN, “gbd driver can’t handle unmapped buffers”);
            bp->b_flags |= B_ERROR;
            iodone(bp);
            return;
        }

        /* Get exclusive upper-half use of device. The sema is released
         * wherever iodone() is called, here or in the int handler.
         */
        psema(&inf->use_lock,PZERO)
        inf->curbp = bp;

        /* Initialize the current transfer address and count.
         * The first transfer should finish the rest of the
         * page, but do no more than the total byte count.
         */
        inf->curaddr = bp->b_dmaaddr;
        inf->totcount = bp->b_bcount;
        inf->curcount = IO_NBPP - io_poff(inf->curaddr);
        if (bp->b_bcount < inf->curcount)
            inf->curcount = bp->b_bcount;

        /* Get exclusive use of the device regs and start the transfer
         * of the first/only segment of data. */
        lk = mutex_spinlock(&inf->reg_lock);
        regs->startaddr = kvtophys(inf->curaddr);
        regs->count = inf->curcount;
        regs->direction = (bp->b_flags & B_READ) ? GBD_READ : GBD_WRITE;
        regs->command = GBD_GO; /* start DMA */

        /* release use of the device regs to the interrupt handler */
        mutex_spinunlock(inf->reg_lock,lk);
        /* and return; upper layers of kernel wait for iodone(bp) */
    }

    /* INTERRUPT: for software scatter/gather. This version is more typical
     * of boards that do have DMA, and more typical of devices that support
     * block i/o, as opposed to character i/o.
     */
    /* ARGSUSED1 */
    void
    gbdintr(int unit, struct eframe_s *ef)
    {
        register struct gbd_info *inf = &gbd_globals[unit];
        register gbd_regs *regs = inf->gbd_device;
        register buf_t *bp = inf->curbp;
        int error = 0;
        int lk;


        /* get exclusive use if device regs from upper-half */
        lk = mutex_spinlock(&inf->reg_lock);
        /* If interrupt was not from this device, exit quick */
        if (! (regs->status & GBD_INTR_PEND) ) {
            mutex_spinunlock(&inf->reg_lock,lk);
            return;
        }

        /* MISSING: read board registers, clear interrupt,
         * and note any errors in the “error” variable. */
        if(error) {
            bp->b_resid = inf->totcount; /* show bytes undone */
            bp->b_flags |= B_ERROR; /* flag error in transfer */
            iodone(bp); /* we are done, tell upper layers */
            vsema(&inf->use_lock); /* make device available */
        }
        else {
            /* Note the successful transfer of one segment. */
            inf->curaddr += inf->curcount;
            inf->totcount -= inf->curcount;
            if(inf->totcount <= 0) {
                iodone(bp); /* we are done, tell upper layers */
                vsema(&inf->use_lock); /* make device available */
            }
            else {
                /* More data to transfer. Reprogram the board for
                 * the next segment and start the next DMA.
                 */
                inf->curcount = (inf->totcount < IO_NBPP) ? inf->totcount : IO_NBPP;
                regs->startaddr = kvtophys(inf->curaddr);
                regs->count = inf->curcount;
                regs->direction = (bp->b_flags & B_READ) ? GBD_READ : GBD_WRITE;
                regs->command = GBD_GO; /* start next DMA */
            }
        }
        /* release lock on exclusive use of device regs */
        mutex_spinunlock(&inf->reg_lock,lk);
    }
    #endif /*  GBD_NUM_DMA_PGS */
    #endif /* GBD_NODMA */
```

PART NINE

STREAMS Drivers

Chapter 19: STREAMS Drivers
How STREAMS drivers are integrated into the IRIX system.

Chapter 19

STREAMS Drivers

The IRIX implementation of STREAMS drivers is intended to be compatible with the multiprocessor implementation of STREAMS in UNIX version SVR4.2.

STREAMS programming in SVR4.2 is documented in *STREAMS Modules and Drivers, UNIX SVR4.2*. That book contains detailed discussion and many examples of STREAMS programming.

References in this chapter to *STREAMS Modules and Drivers* are to the edition copyright 1992 by UNIX System Laborartories, published by UNIX Press/Prentice-Hall, and bearing ISBN 0-13-066879. If you are using an earlier edition, you should upgrade it. If you have a later edition, you may have to interpret references carefully.

This chapter contains the following major sections:

- "Driver Exported Names" on page 542 summarizes the public names and functions that a STREAMS driver must export.
- "Building and Debugging" on page 546 describes the ways that building a STREAMS driver are like and unlike other kernel-level drivers.
- "Special Considerations for Multiprocessing" on page 547 describes the methods you must use to work with the multi-threaded STREAMS monitor.
- "Special Considerations for IRIX" on page 549 details the points at which IRIX differs from the SVR4 STREAMS environment.
- "Summary of Standard STREAMS Functions" on page 554 lists the available kernel functions used by STREAMS drivers.
- "STREAMS Modules for X Input Devices" on page 556 describes the use of configuration files for special input devices used by the X display manager.

Driver Exported Names

A STREAMS driver or module must define certain public names for use by *lboot*, as described in “Summary of Driver Structure” on page 136. Only one of these names, the info structure, is unique to a STREAMS driver or module; all the others are also defined by kernel-level device drivers.

The public names all begin with a prefix (see “Driver Name Prefix” on page 136); the same prefix is specified in the configuration file (see “Describing the Driver in /var/sysgen/master.d” on page 233).

Streamtab Structure

A STREAMS driver or module must provide a global *streamtab* structure containing pointers to the *qinit* structures for its read and write queues. These structures in turn point to required *module_info* structures. The name of the streamtab is *pfx***info**.

Driver Flag Constant

A STREAMS driver or module should provide a driver flag constant containing either 0 or the flag D_MP. (See “Driver Flag Constant” on page 140 and “Flag D_MP” on page 141). The name of the constant is *pfx***devflag**.

Note: A driver or module that does not export *pfx***devflag** is assumed to use SVR3 calling conventions at its *pfx***open()** and *pfx***close()** entry points. However, this support will be withdrawn in a release of IRIX in the very near term. If you are porting a STREAMS driver or module to IRIX you are urged to make sure it uses SVR4 conventions and exports a *pfx***devflag** containing at least 0.

Initialization Entry Points

A STREAMS driver or module can define an entry point *pfx***init()**, or an entry point *pfx***start()**, or both. These entry points will be called during boot if the driver or module is included in the kernel, or when the driver or module is loaded if it is loadable. The operation of these entry points is the same as for device drivers (see “Initialization Entry Points” on page 143).

Many STREAMS drivers perform all initialization at open time, and have no *pfx***init()** or *pfx***start()** entry points. Many STREAMS modules perform initialization when they receive the I_PUSH ioctl message.

Entry Point open()

A STREAMS driver (but not module) must export a *pfx***open()** entry point. The argument list for a STREAMS driver’s open differs from that of a device driver. The prototype for a STREAMS *pfx***open()** entry point is:

```
int
pfxopen(queue_t *q, dev_t *devp, int oflag, int sflag, cred_t *crp);
```

The argument values are

**q*	Pointer to the *queue* structure being opened.
**devp*	Pointer to a *dev_t* value from which you can extract both the major and minor device numbers.
oflag	Flag bits specifying user mode options on the **open()** call.
sflag	Flag bits specifying the type of STREAM open: driver, module or clone.
**crp*	Pointer to a *cred_t* object—an opaque structure for use in authentication.

The *pfx***open()** entry point is a public name. In addition a pointer to it must be defined in the *qinit* structure for the read queue.

Entry Point close()

A STREAMS driver (but not module) must export a *pfx***close()** entry point. The argument list for a STREAMS driver's close differs from that of a device driver. The prototype for a STREAMS *pfx***close()** entry point is:

```
int
pfxclose(queue_t *q, int oflag, cred_t *crp);
```

The argument values are the same as passed to *pfx***open()**. The *pfx***close()** entry point is a public name. In addition a pointer to it must be defined in the *qinit* structure for the read queue.

Put Functions wput() and rput()

Every STREAMS driver and module must define a **put()** function to handle messages as they are delivered to a queue.

The prototype of a **put()** function is as follows:

```
int
name(queue_t *q, mblk_t *mp);
```

Because the **put()** function for a given queue is addressed from the associated *qinit* structure, there is no requirement that the **put()** function be a public name, and no requirement that it begin with the prefix string. The **put()** function for the write queue, which handles messages moving "downstream" from the user process toward the driver, is conventionally called the **wput()** function. All write queues need a **wput()** function.

The **put()** function for the read queue, which handles messages moving "upstream" from the driver toward the user process, is conventionally called the **rput()** function. In some cases the **rput()** function is not required, for example in a driver where all upstream messages are generated by an interrupt handler.

Typically, a **put()** function decides what to do by switching on the message type value from *mp->b_datap->db_type*. A **put** routine must do at least one of the following:

- Process the message, if immediate processing is required, consuming the message or transforming it.
- Pass the original or processed message to the next component in the stream by calling the **putnext()** function (see the putnext(D3) reference page).
- Queue the message for deferred processing by the service routine with the **putq()** function (see the putq(D3) reference page).

When all processing is deferred to the service function, the address of the kernel function **putq()** can be given as a queue's **put()** function.

In a multiprocessor, a **put()** function can be called concurrently with user-level code, and concurrently with another **put()** function for the same or a different queue. A service function for the same or different queue can also be executing concurrently.

Service Functions rsrv() and wsrv()

When a STREAMS driver defers message processing by setting the kernel function **putq()** address as the driver's **put()** function, the queue must also define a service function **srv()**.

Because the **srv()** function for a given queue is addressed from the associated *qinit* structure, there is no requirement that the **srv()** function be a public name, and no requirement that it begin with the prefix string.

The prototype of a **svr()** function is as follows:

```
int
name(queue_t *q);
```

The **srv()** function for the write queue, which handles messages moving "downstream" from the user process toward the driver, is conventionally called the **wsrv()** function. The **srv()** function for the read queue, which handles messages moving "upstream" from the driver toward the user process, is conventionally called the **rsrv()** function.

An **srv()** function is called by the STREAMS monitor to deal with queued messages. It is called at a time chosen by the monitor, not necessarily related to any call to the **put()** function for the same queue. In a multiprocessor, only one instance of **srv()** is called per queue at any time. However, one or more instances of the **put()** function could execute concurrently with the **srv()** function—so any data that is used in common by **put()** and **srv()** must be protected with a lock (see "Waiting and Mutual Exclusion" on page 204). User-level code can also execute concurrently with a service function.

The service function is expected to dispose of all queued messages through one of the following actions:

- Consuming and freeing the message.
- Passing the message on to the following queue using **putnext()** (see the putnext(D3) reference page).
- Replacing the message on the same queue using **putbq()** for processing later (see the putbq(D3) reference page).

The service function implements flow control (which the **put()** function cannot do). Before applying **putnext()**, the service function calls a flow control function such as **canputnext()** to find out if the following queue can accept a message. If the following queue cannot accept a message, the service function replaces the message with **putbq()** and exits.

A STREAMS module or driver that is not multiprocessor-aware (lacks D_MP in its *pfx***devflags**) uses one set of functions for flow control (see the canput(D3) and bcanputnext(D3) reference pages), while one that is multiprocessor-aware uses a different set (see canputnext(D3) and bcanputnext(D3)).

Building and Debugging

A STREAMS driver or module is a kernel module and is compiled using the same compiler options as any driver (see "Compiling and Linking" on page 228).

You configure each STREAMS driver or module as part of the IRIX kernel by:

- Placing the executable module in */var/sysgen/boot*
- Writing a descriptive file and placing it in */var/sysgen/master.d* (see "Describing the Driver in /var/sysgen/master.d" on page 233)

- Placing a USE or INCLUDE line in */var/sysgen/system* (see “Configuring a Kernel” on page 236)

When a STREAMS driver or module is loadable, you specify the appropriate options in the descriptive file (see “Master File for Loadable Drivers” on page 238). You can configure a STREAMS driver or module to be autoregisterd and loaded automatically (see “Registration” on page 239). Alternatively, you can require a STREAMS driver or module to be loaded manually using the *ml* command (see “Loading” on page 239).

When you have configured a debugging kernel (see “Preparing the System for Debugging” on page 243), the symbols of a STREAMS driver or module are available for display. You can set breakpoints using *symmon* (see “Using symmon” on page 252). You can display symbols using *symmon* or *idbg* (see “Using idbg” on page 262). In particular, *idbg* has built-in support for displaying the contents of structures used by a STREAMS module or driver (see “Commands to Display STREAMS Structures” on page 268).

Special Considerations for Multiprocessing

In IRIX releases prior to 6.2, the STREAMS monitor was single-threaded, so that only one **put()** or **srv()** function in the entire system could execute at any time. That one **put()** or **srv()** function might execute concurrently with user-level code, but no two STREAMS functions could execute concurrently.

Beginning with IRIX 6.2, the STREAMS monitor is multi-threaded. Depending on the version of IRIX and on the number of CPUs in the system, the following functions can run concurrently in any combination: one **srv()** function for each queue; any number of **put()** functions for each queue; and one or more user processes. For general discussion of the consequences, see “Planning for Multiprocessor Use” on page 171.

In the multithreaded monitor, when a module or driver calls **putq()** or **qenable()**, the service function for the enabled queue can begin to execute at any time. It can begin execution before **putq()** or **qenable()** call has returned, and can run concurrently with the module or driver that enabled the queue.

The STREAMS monitor runs concurrently with interrupt handling. For this reason, the interrupt handler of a STREAMS driver must take an extra step before it performns any STREAMS-related processing such as **allocb()**, **putq()**, or **qenable()**. The IRIX-unique functions provided for this purpose are summarized in Table 19-1.

Table 19-1 Multiprocessing STREAMS Functions

Name	Can Sleep?	Summary
streams_interrupt(D3)	N	Synchronize interrupt-level function with STREAMS mechanism.
STREAMS_TIMEOUT(D3)	N	Synchronize timeout with STREAMS mechanism.

Suppose that the interrupt handler of a STREAMS driver needs to add a message to the read queue with **putq()**. It cannot simply call that function, since the STREAMS monitor might be using the queue at the same time in another CPU. The driver must define a function in which the **putq()** call is written. The name of this function and the pointer to the queue are passed to **streams_interrupt()**. As soon as possible, **streams_interrupt()** gets control of the queue and executes the passed function.

A callback function scheduled using **itimeout()** and similar functions (see “Waiting for Time to Pass” on page 214) must also be synchronized with the STREAMS monitor.

Suppose that a STREAMS driver or module needs to schedule a function to execute at a later time. (In a nonSTREAMS driver the function would be scheduled with **itimeout()**.) In the time-delayed function is a call to **qenable()**. That call cannot be executed freely whenever the interval expires, because the state of the STREAMS monitor is not known at that time.

The STREAMS_TIMEOUT macros provide a solution. Like **itimeout()**, it schedules a function to be executed at a later time. However, it defers calling the function until the function is synchronized with the STREAMS monitor, so that it can execute calls such as **qenable()**.

Special Considerations for IRIX

While IRIX is largely compatible with UNIX SVR4.2, there are points of difference in the implementation of IRIX that have to be reflected in the design of a STREAMS driver or module. This topic lists points at which the contents of *STREAMS Modules and Drivers, UNIX SVR4.2* is not a correct description of IRIX and STREAMS use within IRIX.

Extension of Poll and Select

Under IRIX, the **poll()** system function is not limited to testing STREAMS, but can be applied to file descriptors of all types (see the poll(2) and select(2) reference pages). In addition the **select()** function can be applied to STREAMS file descriptors. You may want to note this under the heading "STREAMS System Calls" in Chapter 2 of *STREAMS Modules and Drivers, UNIX SVR4.2*.

Support for Pipes

IRIX supports two kinds of pipes with different semantics, as described in the pipe(2) reference page. The default type of pipe is compatible with UNIX SVR3, and does not conform to the description in Chapter 2 of *STREAMS Modules and Drivers, UNIX SVR4.2* under the heading "Creating a STREAMS-based Pipe."

The SVR4 pipe semantics are enabled on a system-wide basis by using the *systune* command to set the tuning parameter *svr3pipe* to 0. First test the configuration as shown in Example 19-1.

Example 19-1 Testing Pipe Configuration

```
# systune | grep svr3pipe
       svr3pipe = 1 (0x1)
```

Service Scheduling

At two points in *STREAMS Modules and Drivers, UNIX SVR4.2* (Under "Service Procedure" in Chapter 4 and under "Message Processing" in Chapter 5), the book explicitly says that in a uniprocessor, enabled service functions are always executed before returning to user-level processing. This promise is not supported by IRIX. In both uniprocessors and multiprocessors, user-level processes can potentially execute after a service function is enabled and before it executes.

Supplied STREAMS Modules

STREAMS Modules and Drivers, UNIX SVR4.2, Chapter 4, refers to some example STREAMS drivers named CHARPROC, CANONPROC, and ASCEBC. These examples are not supplied with IRIX.

The following STREAMS-based modules are supplied with IRIX. You can read their reference pages in volume 7:

alp(7)	Algorithm pool management module.
clone(7)	Clone-open driver; see "Support for CLONE Drivers" on page 552.
connld(7)	Line discipline for unique stream connections.
kbd(7)	Generalized string translation module.
log(7)	Interface to STREAMS error logging and event tracing.
sad(7)	STREAMS Administrative Driver.
streamio(7)	STREAMS ioctl commands.
timod(7)	Transport Interface cooperating STREAMS module.
tirdwr(7)	Transport Interface read/write interface STREAMS module.
tsd(7)	TELNET server protocol STREAMS device.

No #idefs

Chapter 4 of *STREAMS Modules and Drivers, UNIX SVR4.2* refers in a note to the use of the #idef and a transition period for SVR3-compatible drivers. None of this material is relevant to IRIX. IRIX is SVR4-compatible, with no special provision for SVR3 drivers.

Different I/O Hardware Model

Chapter 5 of *STREAMS Modules and Drivers, UNIX SVR4.2* discusses the use of memory-mapped hardware and of Dual-Access RAM (DARAM). None of these considerations are relevant in a MIPS processor. The MIPS I/O model is discussed in Chapter 1, “Physical and Virtual Memory.”

Different Network Model

Chapter 10 of *STREAMS Modules and Drivers, UNIX SVR4.2* describes the TPI interface model. This model is supported in IRIX. When an application uses the TLI library functions such as **t_open()**, the library uses IRIX-provided TPI STREAMS modules which implement the protocol described in chapter 10.

Chapter 11 of *STREAMS Modules and Drivers, UNIX SVR4.2* describes the Data Link Provider Interface (DLPI) as implemented using STREAMS facilities.

The IRIX networking support is not STREAMS-based, but rather is based on BSD *ifnet* architecture. This is discussed in Chapter 16, “Network Device Drivers.” The IRIX network support includes DLPI support as an add-on feature to the *ifnet* driver interface. If you are porting a network device driver to IRIX, it is better to convert it to the *ifnet* interface. You can install a DLPI-based network device driver, but only other STREAMS modules could use it—there would be no connection to the rest of the IRIX networking system.

Support for CLONE Drivers

STREAMS Modules and Drivers, UNIX SVR4.2 discusses CLONE drivers; that is, STREAMS drivers that generate a new minor device number for each open. Refer to Chapter 3, "The CLONE Driver," and to Chapter 8, "Cloning." Clone opens and the clone driver are implemented under IRIX; this section clarifies the discussion in the SVR4 manual.

The essence of cloned access to a STREAMS driver is that the user process is indifferent to the minor device number, and simply wants to open a stream from this driver. A cloned stream is created using the following steps:

1. Recognize that the process calling **open()** is indifferent to the minor device number and simply wants cloned access.
2. Choose an unused minor device number from the set of minor numbers the driver supports.
3. Construct a new device number *dev_t* value based on the chosen minor number, and assign it to the argument passed to *pfx***open()**.

Using the CLONE Driver

The IRIX-supplied clone driver automates some of these steps for your driver. In order to use it, prepare a device special file with these characteristics:

- A device name that is related to the actual device name
- The major device number (10 decimal) that specifies the clone driver
- A minor device number equal to the major number of the actual driver

You can view the descriptive file for the clone driver in */var/sysgen/master.d/clone*. This file sets its major number (10) and states that it is not loadable. Although the clone driver is not specifically configured in the */var/sysgen/system/irix.sm* file, it is included in any kernel because it is listed as a dependency in the descriptive file of several other drivers (use *fgrep clone /var/sysgen/master.d/** to see which drivers depend on it; and see "Listing Dependencies" on page 234). You can specify it as a dependency in the same way, if your driver depends on it.

When a user process opens a device special file with the major number of the clone driver, the kernel naturally calls the clone driver's open entry point. The clone driver verifies that the minor number passed is the major number of an existing, STREAMS driver. (If it is not, the clone driver returns ENXIO).

The clone driver sets up the *qinit* structure appropriately for the target driver's queue and calls that driver's *pfx***open()** entry point, passing the CLONEOPEN flag in the *sflag* argument (see "Entry Point open()" on page 543).

Recognizing a Clone Request Independently

It is not essential to use the clone driver. You can instead designate a particular minor device number to stand for "clone open." You prepare a device special file with these characteristics:

- A device name related to the actual device name
- The major number of your driver
- Some minor number you define to mean "clone open"

When a user process opens this device special file, the kernel calls the *pfx***open()** entry point of your driver. It does not pass the CLONEOPEN flag in *sflag*, but your driver can recognize a request for a clone open based on the minor device number.

Responding to a Clone Request

In response to a clone request coming from either of the two methods described, your *pfx***open()** entry point must select an unused minor device number. (If no minor number is available, return EBUSY.)

Text in Chapter 3 of *STREAMS Modules and Drivers, UNIX SVR4.2* seems to suggest that your driver should scan through the kernel's *cdevsw* table to find an unused minor number (see "Kernel Switch Tables" on page 137). Under IRIX, the *cdevsw* table is not accessible to drivers. The reason is that the table layout differs between 32-bit and 64-bit kernels, and can change between releases. Instead, your driver must know the minor numbers that it supports, and must know which ones are currently in use.

Tip: You can design your driver so that the number of supported devices is specified in the descriptive file in */var/sysgen/master.d*, and passed in to the driver through that descriptive file (see "Variables Section" on page 235). Your driver can allocate and initialize an array of device information structures in its *pfx***init()** entry point.

Your driver constructs a new *dev_t* value, specifying its major number and the selected minor number. The **makedevice()** function is used for this (see the makedevice(D3) reference page, which has some sample code for use in a clone open). The new *dev_t* value is stored into the **devp* argument passed to *pfx***open()**.

Summary of Standard STREAMS Functions

The supported kernel functions for STREAMS operations are summarized for reference in Table 19-2. To declare the necessary prototypes and data types, include *sys/types*.h and *sys/stream.h*.

Table 19-2 Kernel Entry Points

Name	Can Sleep?	Summary
adjmsg(D3)	N	Trim bytes from a message.
allocb(D3)	N	Allocate a message block.
bcanput(D3)	N	Test for flow control in a specified priority band.
bcanputnext(D3)	N	Test for flow control in a specified priority band.
bufcall(D3)	N	Call a function when a buffer becomes available.
canput(D3)	N	Test for room in a message queue.
canputnext(D3)	N	Test for room in a message queue.
copyb(D3)	N	Copy a message block.
copymsg(D3)	N	Copy a message.
datamsg(D3)	N	Test whether a message is a data message.
dupb(D3)	N	Duplicate a message block.
dupmsg(D3)	N	Duplicate a message.
enableok(D3)	N	Allow a queue to be serviced.
esballoc(D3)	N	Allocate a message block using an externally-supplied buffer.
esbbcall(D3)	N	Call a function when an externally-supplied buffer can be allocated.
flushband(D3)	N	Flush messages in a specified priority band.
flushq(D3)	N	Flush messages on a queue.
freeb(D3)	N	Free a message block.
freemsg(D3)	N	Free a message.

Table 19-2 (continued) Kernel Entry Points

Name	Can Sleep?	Summary
freezestr(D3)	N	Freeze the state of a stream.
getq(D3)	N	Get the next message from a queue.
insq(D3)	N	Insert a message into a queue.
linkb(D3)	N	Concatenate two message blocks.
msgdsize(D3)	N	Return number of bytes of data in a message.
msgpullup(D3)	N	Concatenate bytes in a message.
noenable(D3)	N	Prevent a queue from being scheduled.
OTHERQ(D3)	N	Get a pointer to queue's partner queue.
pcmsg(D3)	N	Test whether a message is a priority control message.
pullupmsg(D3)	N	Concatenate bytes in a message.
putbq(D3)	N	Place a message at the head of a queue.
putctl(D3)	N	Send a control message to a queue.
putctl1(D3)	N	Send a control message with a one-byte parameter to a queue.
putnext(D3)	N	Send a message to the next queue.
putnextctl(D3)	N	Send a control message to a queue.
putnextctl1(D3)	N	Send a control message with a one-byte parameter to a queue.
putq(D3)	N	Put a message on a queue.
qenable(D3)	N	Schedule a queue's service routine to be run.
qprocsoff(D3)	Y	Enable put and service routines.
qprocson(D3)	Y	Disable put and service routines
qreply(D3)	N	Send a message in the opposite direction in a stream.
qsize(D3)	N	Find the number of messages on a queue.
RD(D3)	N	Get a pointer to the read queue.

Table 19-2 (continued) Kernel Entry Points

Name	Can Sleep?	Summary
rmvb(D3)	N	Remove a message block from a message.
rmvq(D3)	N	Remove a message from a queue.
SAMESTR(D3)	N	Test if next queue is of the same type.
strqget(D3)	N	Get information about a queue or band of the queue.
strqset(D3)	N	Change information about a queue or band of the queue.
unbufcall(D3)	N	Cancel a pending bufcall request.
unfreezestr(D3)	N	Unfreeze the state of a stream.
unlinkb(D3)	N	Remove a message block from the head of a message.
WR(D3)	N	Get a pointer to the write queue.

STREAMS Modules for X Input Devices

The Silicon Graphics, Inc. implementation of the X display manager, *Xsgi*, is a customized version of the MIT X11 Sample Server. Besides other enhancements such as integration with Silicon Graphics proprietary graphics subsystems, *Xsgi* implements a generalized input subsystem so that unusual input devices can easily be integrated into the X window system. The input system is based on STREAMS modules.

The X Input Subsystem

While X mandates that every X server support a keyboard and mouse, there is no standard system interface for accessing such devices on UNIX systems. This means each vendor has its own input subsystem for its X server. SGI's input subsystem not only meets the basic requirement to support a keyboard and mouse but also has the following features:

- A shared memory input queue is supported for high performance
- A wide variety of input devices is supported, including 3D devices such as the Spaceball

- Input devices are supported abstractly; knowledge of specific input devices is isolated to modular kernel-level device drivers
- Hardware cursor tracking is supported in the kernel

These features provide a more functional, responsive input subsystem than that available in the MIT Sample Server.

The programming interface to the input subsystem from the X client API is covered in the *X11 Input Extension Library Specification*, an online book that is distributed with the IRIX Developer's Option.

Note: Numerous code examples demonstrating the X input system are available in the X developer component (x_dev component) of the IRIX Developer Option. Source for STREAMS modules to integrate a Spaceball, a dial-and-button box, and other devices can be found in subdirectories of */usr/share/src/X*.

Shared Memory Input Queue

A shared memory input queue (called a *shmiq* in Silicon Graphics code comments, and pronounced "shmick") is a fast way of receiving input device events by eliminating the filesystem overhead to receive data from input devices. Instead of the X server reading the input devices through file descriptors, a kernel-level driver deposits input events directly into a region of the X server's address space, organized as a ring buffer.

The IRIX shmiq device driver is implemented as a STREAMS multiplexor. This allows an arbitrary number of input sources (in the form of STREAMS modules) to be linked to it so all input sources are funneled through the shmiq.

In addition to processing input events from input device modules, the schmiq driver also processes events from the graphics subsystem, and updates the screen cursor position. This allows smooth cursor movement since cursor positioning is done in kernel code, without *Xsgi* involvement.

IDEV Interface

X input devices are integrated into the shmiq driver by implementing STREAMS modules that translate raw device input into abstract events which are sent to the shmiq driver (and on to the server). For example, an input device that connects to a serial port can be integrated in the form of a STREAMS module that is pushed onto the stream from that serial device, and translates incoming bytes into event messages.

The shmiq driver expects messages from all input devices to be in the form of IDEV events, as documented in the */usr/include/sys/idev.h* header file; hence this is called the IDEV interface. IDEV device events appear as valuator, button, and pointer state changes.

The IDEV interface defines two-way communications between the input device and *Xsgi*. Besides the uniform set of IDEV input events, the interface defines a standard set of abstract commands that *Xsgi* can send down (using IOCTL messages) to initialize and control input devices. This allows the server to see input devices as abstract input sources and does not require special server code to be written every time a new input device is supported. Instead, device specific knowledge of each devices is encapsulated in an IDEV-based STREAMS module linked into the kernel.

Input Device Naming

Xsgi recognizes as input devices, any device special files in the */dev/input* directory. At a minimum this includes */dev/input/keyboard* and */dev/input/mouse*. Other input devices that are to be integrated into the IDEV interface must also appear in */dev/input*.

Typically an X input device is defined as a link from */dev/input* to some other device special file, for example a serial port in the */dev/tty** series. The filename in */dev/input* determines the name of the STREAMS module that is used to interface that device to the IDEV input system. For example, if the file is */dev/input/calcomp*, the *calcomp* STREAMS module is loaded and pushed onto the stream from the device.

When a single STREAMS module is used to support two or more devices, you can use a hyphen-digit suffix on the filename. For example, the *calcomp* STREAMS module would be used for both */dev/input/calcomp-1* and */dev/input/calcomp-2*.

When a device is initialized (as described in the next section), the STREAMS module is asked to return the X name of the input device. This name can be the same as the name of the device and the module, or it can be different. Typically the device and module names will reflect the hardware type (for example *calcomp*), while the X name reflects the kind of device (for example *tablet*).

Opening Input Devices

An input device is opened at one of two times: when the X server starts up, and when an X client requests an open.

Starting Up the Server

When *Xsgi* starts up, it opens each device name in */dev/input* and for each one it:

- Loads a STREAMS module that has the same name as the name of the device special file, and pushes it onto the stream from the device, below the shmiq multiplexor.

  The STREAMS module may be loadable, and most IDEV modules are loadable.
- Looks for a file in */usr/lib/X11/input/config* having the same name as the module. The device controls in that file are sent down the stream as IOCTL messages.

  The format of device controls is discussed under “Device Controls” on page 560.
- Asks the device to describe itself. This is done by sending down an IOCTL message of the type IDEVDESC. The module must return the IOCTL message with descriptive data.

  The IDEV IOCTL structures are declared in */usr/include/sys/idev.h*. A key element of the device description is the X name of the input device.
- Looks for a file in */usr/lib/X11/input/config* having the X name of the device as returned in the device description. The X init controls in this file are processed by the X server.

  The format of X init controls is discussed under “Device Controls” on page 560.
- Unless autostart was specified for this device, the device is closed.

Opening from a Client

An X application can use the **XListInputDevices()** function to get a list of available input devices. Then it can call **XOpenDevice()** to open a selected device, so that input events from that device will be processed by the X server (see the XListInputDevices(3X) and XOpenDevice(3X) reference pages).

When **XOpenDevice()** is called for an input device that is not already open, it repeats the process done at startup time:

- Loads the STREAMS module and pushes it on the device stream, feeding the shmiq multiplexor.
- Sends device controls from a file in */usr/lib/X11/input/config* having the same name as the module.
- Asks the device (module) to describe itself, including the X name of the device.
- Processes X init controls from a file in */usr/lib/X11/input/config* having the X name of the device.

Device Controls

Device controls are string values that are passed via an IOCTL message to the STREAMS module for an input device at the time the device is opened. You can use device controls as a way of configuring the device module at runtime. Device controls are interpreted only by the module.

X init controls have the same syntax as device controls, but are processed by the X server after the device has been initialized.

Where Controls Are Stored

You can issue X server device controls on the fly by calling **XSGIDeviceControl** from within a program, or by storing them in configuration files in the */usr/lib/X11/input/config* directory. Specific documentation on controls can be found in */usr/lib/X11/input/config/README*.

There are (potentially) two configuration files per device. As noted under “Opening Input Devices” on page 559, the X server looks for device controls in a file with the same name as the STREAMS module that implements the device. After the module returns the X name of the device, the X server looks for X init controls in a file with the X name of the device.

Some devices use the same name for the STREAMS module and for the X device (*tablet*, *mouse*), but some use different names for the two. For example, the STREAMS module for the Spaceball device is *sball*, while the X name is *spaceball*.

The X server intercepts about a dozen **x_init** controls. For a list of the **x_init** controls and some of the more common **device_init** controls, see the file

Control Syntax

When the X server opens a file to look for device controls, it searches the file for a single set of controls with the following format:

```
device_init {
   name     "value"
   ...
}
```

Each *name* may have at most 15 characters. Each *value* may have at most 23 characters. Each pair of name and value are put in an IOCTL message of *idevOtherControl* type and sent down to the device module for interpretation.

When the X server opens a file to look for X init controls, it searches the file for a single set of controls with the following format:

```
x_init {
   name     "value"
   ...
}
```

The syntax is the same, except for the use of x_init instead of device_init.

The specific *name* and *value* strings that the X server supports are documented in the file */usr/lib/X11/input/config/README*. Any *name* strings that are not recognized by the X server are sent down to the device module, just as if they were device controls.

Silicon Graphics Driver/Kernel API

This appendix summarizes the Silicon Graphics Driver/Kernel Authorized Programming Interface in tabular form. The data structures, entry points, and kernel functions are listed alphabetically with cross-references to the pages where they are discussed. The tables also show which functions and structures are compatible with SVR4 and which are unique to IRIX.

The tables in this appendix are based on the reference pages in volume D. The reference pages in volume D constitute the formal, engineering definition of the Driver/Kernel API. When discussion in this book disagrees with the contents of a reference page, the reference page takes precedence (however, any such disagreement should be reported by email to techpubs@sgi.com).

- "Driver Exported Names" on page 564 tabulates the names of data and functions that a driver must export.
- "Kernel Data Structures and Declarations" on page 565 tabulates the objects used in the interface.
- "Kernel Functions" on page 566 tabulates the IRIX kernel services used by drivers.

Each table in this appendix has a column headed "Versions." The codes in this column have the following meanings:

SV	Syntactically and semantically portable from SVR4 UNIX, as documented in the *UNIX SVR4.2 Device Driver Reference*.
SV*	Syntactically portable from UNIX SVR4, but semantics may differ. Read the discussion and reference page carefully when porting.
5.3	Portable from IRIX version 5.3.
5.3*	Portable from IRIX 5.3, but interface has changed in some detail or new ability has been added.
6.2	Introduced in IRIX version 6.2.

Driver Exported Names

A kernel driver is required to export certain names of static data and functional entry points. These exported names are summarized in Table A-1.

Table A-1 Driver Exported Names

Name	Summary	Discussed	Versions
close(D2)	Notify driver of final close of minor device.	page 149	SV, 5.3
devflag(D1)	Show driver attributes to *lboot*.	page 140	SV*, 5.3*
halt(D2)	Notify driver of system shutdown.	page 168	SV, 5.3
info(D1)	Show driver entries to STREAMS interface.	page 542	SV, 5.3
init(D2)	Initialize driver early in system startup.	page 144	SV*, 5.3
intr(D2)	Notify driver of device interrupt.	page 164	SV, 5.3
ioctl(D2)	Call driver to implement ioctl() call.	page 150	SV*, 5.3
map(D2)	Call driver to implement IRIX mmap().	page 159	5.3
mmap(D2)	Call driver to implement mmap().	page 161	SV*, 5.3
open(D2)	Call driver to open a device.	page 146	SV, 5.3
print(D2)	Call block driver to display filesystem error.	page 169	SV, 5.3
put(D2)	Call STREAMS driver to receive message.	page 544	SV, 5.3
read(D2)	Call character driver to read data.	page 152	SV, 5.3
size(D2)	Call block driver to get device capacity.	page 169	SV, 5.3
srv(D2)	Call driver to service queued messages.	page 545	SV, 5.3
start(D2)	Initialize driver late in system startup.	page 145	SV, 5.3
strategy(D2)	Call block driver to read or write data.	page 154	SV*, 5.3
unload(D2)	Call loadable driver prior to unloading it.	page 167	5.3
unmap(D2)	Call driver to notify it of unmap() call.	page 162	5.3
write(D2)	Call character driver to write data.	page 152	SV, 5.3

The following reference pages have overview information on exported names: intro(D1), intro(D2), and prefix(D1).

Note: The following SVR4 exported names are not used in IRIX drivers: chpoll, _load, and _unload. The latter is replaced by *pfx***load()** without the leading underscore.

Kernel Data Structures and Declarations

The driver/kernel interface is based on shared use of certain data types and defined constant values. For general information on these interface objects, see the intro(D4) and intro(D5) reference pages.

The interface objects used by device drivers are summarized in Table A-2. .

Table A-2 Device Driver Interface Objects

Name	Summary	Discussed	Versions
buf(D4)	Block read/write request structure.	page 183	SV*, 5.3*
eisa_dma_cb(D4)	DMA command block for EISA slave DMA.	page 446	5.3
eisa_dma_buf(D4)	DMA command buffer for EISA slave DMA.	page 446	5.3
errnos(D5)	Error numbers valid for driver use.		SV*, 5.3
iovec(D4)	Describes an I/O buffer segment to the read or write entry points.	page 182	SV, 5.3
signals(D5)	Lists signal numbers valid for driver use.		SV*, 5.3
uio(D4)	Describes an I/O request to the read or write entry points.	page 182	SV*, 5.3

Note: The following data structures used in SVR4 drivers are not used in IRIX: *dma_buf* and *dma_cb*. The *eisa_dma_buf* and *eisa_dma_cb* structures are similar but are used only in EISA drivers.

The interface objects used by STREAMS drivers are summarized in Table A-3

Table A-3 STREAMS Driver Interface Objects

Name	Summary	Discussed	Versions
copyreq(D4)	Copy request structure.		SV, 5.3
copyresp(D4)	Copy response structure.		SV, 5.3
datab(D4)	Message data block.		SV, 5.3
free_rtn(D4)	Describes a message-free routine.		SV, 5.3
iocblk(D4)	Describes ioctl() data or response.		SV, 5.3
linkblk(D4)	Describes multiplexed link.		SV, 5.3
module_info(D4)	Describes module attributes.		SV, 5.3
msgb(D4)	Describes all or part of a message.		SV, 5.3
qinit(D4)	Points to handlers and parameters for a queue.		SV, 5.3
queue(D4)	Describes a queue of messages.		SV, 5.3
streamtab(D4)	Points to the queues handled by a driver.		SV, 5.3
stroptions(D4)	Lists stream-head options.		SV, 5.3

Kernel Functions

The IRIX kernel makes available the Table A-4 functions summarized in .

Table A-4 Kernel Functions

Name	Summary	Discussed	Versions
adjmsg(D3)	Trim bytes from a message.		SV, 5.3
allocb(D3)	Allocate a message block.		SV, 5.3
ASSERT(D3)	Debugging macro designed for use in the kernel (compare to assert(3X)).	page 252	5.3
badaddr(D3)	Test physical address for input.	page 195	5.3

Table A-4 (continued) Kernel Functions

Name	Summary	Discussed	Versions
badaddr_val(D3)	Test physical address for input and return the input value received.	page 195	6.2
bcanput(D3)	Test for flow control in a specified priority band.		SV, 5.3
bcanputnext(D3)	Test for flow control in a specified priority band.		SV, 5.3
bcmp(D3)	Compare data between kernel locations.	page 193	SV, 5.3
bcopy(D3)	Copy data between locations in the kernel.	page 193	SV, 5.3
biodone(D3)	Mark a *buf_t* as complete and wake any process waiting for it.	page 217	SV, 5.3
bioerror(D3)	Manipulate error fields within a *buf_t*.	page 217	SV, 5.3
biowait(D3)	Suspend process pending completion of block I/O.	page 217	SV, 5.3
bp_mapin(D3)	Map buffer pages into kernel virtual address space.	page 199	SV, 5.3
bp_mapout(D3)	Release mapping of buffer pages.	page 199	SV, 5.3
bptophys(D3)	Get physical address of buffer data.	page 198	5.3
brelse(D3)	Return a buffer to the system's free list.	page 190	SV, 5.3
btod(D3)	Return number of 512-byte "sectors" in a byte count (round up).	page 197	5.3
btop(D3)	Return number of I/O pages in a byte count (truncate).	page 197	SV, 5.3
btopr(D3)	Return number of I/O pages in a byte count (round up).	page 197	SV, 5.3
bufcall(D3)	Call a function when a buffer becomes available.		SV, 5.3
bzero(D3)	Clear kernel memory for a specified size.	page 192	SV, 5.3
canput(D3)	Test for room in a message queue.		SV, 5.3

Table A-4 (continued) Kernel Functions

Name	Summary	Discussed	Versions
canputnext(D3)	Test for room in a message queue.		SV, 5.3
clrbuf(D3)	Erase the contents of a buffer desribed by a buf_t.	page 199	SV, 5.3
cmn_err(D3)	Display an error message or panic the system.	page 249	SV*, 5.3
copyb(D3)	Copy a message block.		SV, 5.3
copyin(D3)	Copy data from user address space.	page 192	SV, 5.3
copymsg(D3)	Copy a message.		SV, 5.3
copyout(D3)	Copy data to user address space.	page 192	SV, 5.3
cpsema(D3)	Conditionally decrement a semaphore's state.	page 222	5.3
cvsema(D3)	Conditionally increment a semaphore's state	page 222	5.3
datamsg(D3)	Test whether a message is a data message.		SV, 5.3
delay(D3)	Delay for a specified number of clock ticks.	page 214	SV, 5.3
disable_sysad_parity()	Disable memory parity checking on SysAD bus.	page 526	
dki_dcache_inval(D3)	Invalidate the data cache for a given range of virtual addresses.	page 200	5.3
dki_dcache_wb(D3)	Write back the data cache for a given range of virtual addresses.	page 200	5.3
dki_dcache_wbinval(D3)	Write back and invalidate the data cache for a given range of virtual addresses.	page 200	5.3
dma_map(D3)	Load DMA mapping registers for an imminent transfer.	page 330	5.3
dma_mapbp(D3)	Load DMA mapping registers for an imminent transfer.	page 330	5.3

Table A-4 (continued) Kernel Functions

Name	Summary	Discussed	Versions
dma_mapaddr(D3)	Return the "bus virtual" address for a given map and address.	page 330	5.3
dma_mapalloc(D3)	Allocate a DMA map.	page 330	5.3
dma_mapfree(D3)	Free a DMA map.	page 330	5.3
drv_getparm(D3)	Retrieve kernel state information.	page 202	SV*, 5.3
drv_hztousec(D3)	Convert clock ticks to microseconds	page 214	SV, 5.3
drv_priv(D3)	Test for privileged user.	page 202	SV, 5.3
drv_setparm(D3)	Set kernel state information.	page 202	SV, 5.3
drv_usectohz(D3)	Convert microseconds to clock ticks.	page 214	SV, 5.3
drv_usecwait(D3)	Busy-wait for a specified interval.	page 214	SV, 5.3
dtimeout(D3)	Schedule a function execute on a specified processor after a specified length of time.	page 214	5.3
dupb(D3)	Duplicate a message block.		SV, 5.3
dupmsg(D3)	Duplicate a message.		SV, 5.3
eisa_dma_disable(D3)	Disable recognition of hardware requests on a DMA channel.	page 446	5.3
eisa_dma_enable(D3)	Enable recognition of hardware requests on a DMA channel.	page 446	5.3
eisa_dma_free_buf(D3)	Free a previously allocated DMA buffer descriptor.	page 446	5.3
eisa_dma_free_cb(D3)	Free a previously allocated DMA command block.	page 446	5.3
eisa_dma_get_buf(D3)	Allocate a DMA buffer descriptor.	page 446	5.3
eisa_dma_get_cb(D3)	Allocate a DMA command block.	page 446	5.3
eisa_dma_prog(D3)	Program a DMA operation for a subsequent software request.	page 446	5.3

Table A-4 (continued) Kernel Functions

Name	Summary	Discussed	Versions
eisa_dma_stop(D3)	Stop software-initiated DMA operation and release channel.	page 446	5.3
eisa_dma_swstart(D3)	Initiate a DMA operation via software request.	page 446	5.3
eisa_dmachan_alloc()	Allocate a DMA channel for EISA slave DMA.	page 444	5.3
eisa_ivec_alloc()	Allocate an IRQ level for EISA.	page 442	5.3
eisa_ivec_set()	Associate a handler with an EISA IRQ.	page 442	5.3
enableok(D3)	Allow a queue to be serviced.		SV, 5.3
enable_sysad_parity()	Reenable parity checking on SysAD bus.	page 526	
esballoc(D3)	Allocate a message block using an externally-supplied buffer.		SV, 5.3
esbbcall(D3)	Call a function when an externally-supplied buffer can be allocated.		SV, 5.3
etoimajor(D3)	Convert external to internal major device number.	page 181	SV, 5.3
fast_itimeout(D3)	Same as itimeout() but takes an interval in "fast ticks."	page 214	6.2
fasthzto(D3)	Returns the value of a *struct timeval* as a count of "fast ticks."	page 214	6.2
flushband(D3)	Flush messages in a specified priority band.		SV, 5.3
flushbus(D3)	Make sure contents of the write buffer are flushed to the system bus	page 200	5.3
flushq(D3)	Flush messages on a queue.		SV, 5.3
freeb(D3)	Free a message block.		SV, 5.3
freemsg(D3)	Free a message.		SV, 5.3
freerbuf(D3)	Free a buf_t with no buffer.	page 190	SV, 5.3

Table A-4 (continued) Kernel Functions

Name	Summary	Discussed	Versions
freesema(D3)	Free the resources associated with a semaphore.	page 222	5.3*
freezestr(D3)	Freeze the state of a stream.		SV, 5.3
fubyte(D3)	Load a byte from user space.	page 192	5.3
fuword(D3)	Load a word from user space.	page 192	5.3
geteblk(D3)	Get a buf_t with no buffer.	page 190	SV, 5.3
getemajor(D3)	Get external major device number.	page 181	SV, 5.3
geteminor(D3)	Get external minor device number.	page 181	SV, 5.3
geterror(D3)	retrieve error number from a buffer header	page 217	SV, 5.3
getmajor(D3)	Get internal major device number.	page 181	SV, 5.3
getminor(D3)	Get internal minor device number.	page 181	SV, 5.3
getnextpg(D3)	Return *pfdat* structure for next page.	page 199	5.3
getq(D3)	Get the next message from a queue.		SV, 5.3
getrbuf(D3)	Allocate a *buf_t* with no buffer.	page 190	SV, 5.3
hwcpin(D3)	Copy data from device registers to kernel memory.	page 192	5.3
hwcpout(D3)	Copy data from kernel memory to device registers.	page 192	5.3
initnsema(D3)	Initialize a semaphore to a specified count.	page 222	5.3
initnsema_mutex(D3)	Initialize a semaphore to a count of 1.	page 222	5.3
insq(D3)	Insert a message into a queue.		SV, 5.3
ip26_enable_ucmem(D3)	Change memory mode on IP26 processor.	page 27	6.2
ip26_return_ucmem(D3)	Change memory mode on IP26 processor.	page 27	SV, 5.3
is_sysad_parity_enabled( )	Test for parity checking on SysAD bus.	page 526	5.3

Table A-4 (continued) Kernel Functions

Name	Summary	Discussed	Versions
itimeout(D3)	Schedule a function to be executed after a specified number of clock ticks.	page 214	SV, 5.3
itoemajor(D3)	Convert internal to external major device number.	page 181	SV, 5.3
kern_calloc(D3)	Allocate and clear space from kernel memory.	page 187	5.3
kern_free(D3)	Free kernel memory space.	page 187	5.3
kern_malloc(D3)	Allocate kernel virtual memory.	page 187	5.3
kmem_alloc(D3)	Allocate space from kernel free memory.	page 187	SV, 5.3
kmem_free(D3)	Free previously allocated kernel memory.	page 187	SV, 5.3
kmem_zalloc(D3)	Allocate and clear space from kernel free memory.	page 187	SV, 5.3
kvtophys(D3)	Get physical address of kernel data.	page 199	5.3
linkb(D3)	Concatenate two message blocks.		SV*, 5.3*
LOCK(D3)	Acquire a basic lock, waiting if necessary.	page 205	SV*, 5.3*
LOCK_ALLOC(D3)	Allocate and initialize a basic lock.	page 205	SV*, 5.3*
LOCK_DEALLOC(D3)	Deallocate an instance of a basic lock.	page 205	SV*, 5.3*
LOCK_INIT(D3)	Initialize a basic lock that was allocated statically, or reinitialize an allocated lock.	page 205	6.2
LOCK_DESTROY(D3)	Uninitialize a basic lock that was allocated statically.	page 205	6.2
makedevice(D3)	Make device number from major and minor numbers.	page 181	SV, 5.3
max(D3)	Return the larger of two integers.		SV, 5.3
min(D3)	Return the lesser of two integers.		SV, 5.3
msgdsize(D3)	Return number of bytes of data in a message.		SV, 5.3

Table A-4 (continued) Kernel Functions

Name	Summary	Discussed	Versions
msgpullup(D3)	Concatenate bytes in a message.		SV, 5.3
MUTEX_ALLOC(D3)	Allocate and initialize a mutex lock.	page 207	6.2
MUTEX_DEALLOC(D3)	Deinitialize and free a dynamically allocated mutex lock.	page 207	6.2
MUTEX_DESTROY(D3)	Deinitialize a mutex lock.	page 207	6.2
MUTEX_INIT(D3)	Initialize an existing mutex lock.	page 207	6.2
MUTEX_ISLOCKED(D3)	Test if a mutex lock is owned.	page 207	6.2
MUTEX_LOCK(D3)	Claim a mutex lock.	page 207	6.2
MUTEX_MINE(D3)	Test if a mutex lock is owned by this process.	page 207	6.2
MUTEX_TRYLOCK(D3)	Conditionally claim a mutex lock.	page 207	6.2
MUTEX_UNLOCK(D3)	Release a mutex lock.	page 207	6.2
MUTEX_WAITQ(D3)	Get the number of processes blocked by mutex lock.	page 207	6.2
ngeteblk(D3)	Allocate a *buf_t* and a buffer of specified size.	page 190	SV, 5.3
noenable(D3)	Prevent a queue from being scheduled.		SV, 5.3
OTHERQ(D3)	Get a pointer to queue's partner queue.		SV, 5.3
pcmsg(D3)	Test whether a message is a priority control message.		SV, 5.3
phalloc(D3)	Allocate and initialize a pollhead structure.	page 189	SV, 5.3
phfree(D3)	Free a pollhead structure.	page 189	SV, 5.3
physiock(D3)	Validate and issue a raw I/O request	page 217	SV, 5.3
pio_andb_rmw(D3)	Byte read-and-write.	page 325	5.3
pio_andh_rmw(D3)	16-bit read-and-write.	page 325	5.3
pio_andw_rmw(D3)	32-bit read-and-write.	page 325	5.3

Table A-4 (continued) Kernel Functions

Name	Summary	Discussed	Versions
pio_badaddr(D3)	Check for bus error when reading an address.	page 325	5.3
pio_badaddr_val(D3)	Check for bus error when reading an address and return the value read.	page 325	5.3
pio_bcopyin(D3)	Copy data from a bus address to kernel's virtual space.	page 325	5.3
pio_bcopyout(D3)	Copy data from kernel's virtual space to a bus address.	page 325	5.3
pio_mapaddr(D3)	Convert a bus address to a virtual address.	page 325	5.3
pio_mapalloc(D3)	Allocate a PIO map.	page 325	5.3
pio_mapfree(D3)	Free a PIO map.	page 325	5.3
pio_orb_rmw(D3)	Byte read-or-write.	page 325	5.3
pio_orh_rmw(D3)	16-bit read-or-write.	page 325	5.3
pio_orw_rmw(D3)	32-bit read-or-write.	page 325	5.3
pio_wbadaddr(D3)	Check for bus error when writing to an address.	page 325	5.3
pio_wbadaddr_val(D3)	Check for bus error when writing a specified value to an address.	page 325	5.3
pollwakeup(D3)	Inform polling processes that an event has occurred.	page 156	SV, 5.3
pptophys(D3)	Convert page pointer to physical address.	page 199	SV, 5.3
proc_ref(D3)	Obtain a reference to a process for signaling.	page 202	SV, 5.3
proc_signal(D3)	Send a signal to a process.	page 202	SV, 5.3
proc_unref(D3)	Release a reference to a process.	page 202	SV, 5.3
psema(D3)	Perform a "P" or wait semaphore operation.	page 222	SV, 5.3
ptob(D3)	Convert size in pages to size in bytes.	page 197	SV, 5.3
pullupmsg(D3)	Concatenate bytes in a message.		SV, 5.3

Table A-4 (continued) Kernel Functions

Name	Summary	Discussed	Versions
putbq(D3)	Place a message at the head of a queue.		SV, 5.3
putctl(D3)	Send a control message to a queue.		SV, 5.3
putctl1(D3)	Send a control message with a one-byte parameter to a queue.		SV, 5.3
putnext(D3)	Send a message to the next queue.		SV, 5.3
putnextctl(D3)	Send a control message to a queue.		SV, 5.3
putnextctl1(D3)	Send a control message with a one-byte parameter to a queue.		SV, 5.3
putq(D3)	Put a message on a queue.		SV, 5.3
qenable(D3)	Schedule a queue's service routine to be run.		SV, 5.3
qprocsoff(D3)	Enable put and service routines.		SV, 5.3
qprocson(D3)	Disable put and service routines		SV, 5.3
qreply(D3)	Send a message in the opposite direction in a stream.		SV, 5.3
qsize(D3)	Find the number of messages on a queue.		SV, 5.3
RD(D3)	Get a pointer to the read queue.		SV, 5.3
rmalloc(D3)	Allocate space from a private space management map.	page 191	SV, 5.3
rmallocmap(D3)	Allocate and initialize a private space management map.	page 191	SV, 5.3
rmalloc_wait(D3)	Allocate resources from a space management map.	page 191	SV, 5.3
rmfree(D3)	Release resources into a space management map.	page 191	SV, 5.3
rmfreemap(D3)	Free a private space management map.	page 191	SV, 5.3
rmvb(D3)	Remove a message block from a message.		SV, 5.3
rmvq(D3)	Remove a message from a queue.		SV, 5.3

Table A-4 (continued) Kernel Functions

Name	Summary	Discussed	Versions
RW_ALLOC(D3)	Allocate and initialize a reader/writer lock.	page 211	SV*, 5.3*
RW_DEALLOC(D3)	Deallocate a reader/writer lock.	page 211	SV*, 5.3*
RW_DESTROY(D3)	Deinitialize an existing reader/writer lock.	page 211	6.2
RW_INIT(D3)	Initialize an existing reader/writer lock.	page 211	6.2
RW_RDLOCK(D3)	Acquire a reader/writer lock as reader, waiting if necessary.	page 211	SV*, 5.3*
RW_TRYRDLOCK(D3)	Try to acquire a reader/writer lock as reader, returning a code if it is not free.	page 211	SV*, 5.3*
RW_TRYWRLOCK(D3)	Try to acquire a reader/writer lock as writer, returning a code if it is not free.	page 211	SV*, 5.3*
RW_UNLOCK(D3)	Release a reader/writer lock as reader or writer.	page 211	SV*, 5.3*
RW_WRLOCK(D3)	Acquire a reader/writer lock as writer, waiting if necessary.	page 211	SV*, 5.3*
SAMESTR(D3)	Test if next queue is of the same type.		SV, 5.3
scsi_abort()	Transmits a SCSI ABORT command.	page 357	5.3*
scsi_alloc(D3)	Open a connection between a driver and a target device.	page 357	5.3*
scsi_command(D3)	Transmit a SCSI command on the bus and return results.	page 357	5.3*
scsi_free(D3)	Release connection to target device.	page 357	5.3*
scsi_info(D3)	Issue the SCSI Inquiry command and return the results.	page 357	5.3*
scsi_reset()	Resets the SCSI adapter or bus.	page 357	5.3*
setgiovector()	Register a GIO interrupt handler.	page 515	5.3
setgioconfig()	Prepare a GIO slot for use.	page 516	5.3
sgset(D3)	Assign physical addresses to a vector of software scatter-gather registers.	page 199	5.3

Table A-4 (continued) Kernel Functions

Name	Summary	Discussed	Versions
sleep(D3)	Suspend process execution pending occurrence of an event.	page 219	SV, 5.3
SLEEP_ALLOC(D3)	Allocate and initialize a sleep lock.	page 209	SV*, 5.3*
SLEEP_DEALLOC(D3)	Deinitialize and deallocate a dynamically allocated sleep lock.	page 209	SV*, 5.3*
SLEEP_DESTROY	Deinitialize a sleep lock.	page 209	6.2
SLEEP_INIT(D3)	Initialize an existing sleep lock.	page 209	6.2
SLEEP_LOCK(D3)	Acquire a sleep lock, waiting if necessary until the lock is free.	page 209	SV*, 5.3*
SLEEP_LOCKAVAIL(D3)	Query whether a sleep lock is available.	page 209	SV*, 5.3*
SLEEP_LOCK_SIG(D3)	Acquire a sleep lock, waiting if necessary until the lock is free or a signal is received.	page 209	SV*, 5.3*
SLEEP_TRYLOCK(D3)	Try to acquire a sleep lock, returning a code if it is not free.	page 209	SV*, 5.3*
SLEEP_UNLOCK(D3)	Release a sleep lock.	page 209	SV*, 5.3*
splbase(D3)	Block no interrupts.	page 213	SV, 5.3
spltimeout(D3)	Block only timeout interrupts.	page 213	SV, 5.3
spldisk(D3)	Block disk interrupts.	page 213	SV, 5.3
splstr(D3)	Block STREAMS interrupts.	page 213	SV, 5.3
spltty(D3)	Block disk, VME, serial interrupts.	page 213	SV, 5.3
splhi(D3)	Block all I/O interrupts.	page 213	SV, 5.3
spl0(D3)	Same as **splbase**().	page 213	SV, 5.3
splx(D3)	Restore previous interrupt level.	page 213	SV, 5.3
strcat(D3)	Append one string to another.		SV, 5.3
strcpy(D3)	Copy a string.		SV, 5.3

Table A-4 (continued) Kernel Functions

Name	Summary	Discussed	Versions
streams_interrupt(D3)	Synchronize interrupt-level function with STREAMS mechanism.		5.3
STREAMS_TIMEOUT(D3)	Synchronize timeout with STREAMS mechanism.		5.3
strlen(D3)	Return length of a string.		SV, 5.3
strlog(D3)	Submit messages to the log driver.		SV, 5.3
strncmp(D3)	Compare two strings for a specified length.		SV, 5.3
strncpy(D3)	Copy a string for a specified length.		SV, 5.3
strqget(D3)	Get information about a queue or band of the queue.		SV, 5.3
strqset(D3)	Change information about a queue or band of the queue.		SV, 5.3
subyte(D3)	Store a byte to user space.	page 192	5.3
suword(D3)	Store a word to user space.	page 192	5.3
SV_ALLOC(D3)	Allocate and initialize a synchronization variable.	page 220	SV*, 5.3*
SV_BROADCAST(D3)	Wake all processes sleeping on a synchronization variable.	page 220	SV*, 5.3*
SV_DEALLOC(D3)	Deinitialize and deallocate a synchronization variable.	page 220	SV*, 5.3*
SV_DESTROY	Deinitialize a synchronization variable.	page 220	6.2
SV_INIT	Initialize an existing synchronization variable.	page 220	6.2
SV_SIGNAL(D3)	Wake one process sleeping on a synchronization variable.	page 220	SV*, 5.3*
SV_WAIT(D3)	Sleep until a synchronization variable is signalled.	page 220	SV*, 5.3*

Table A-4 (continued) Kernel Functions

Name	Summary	Discussed	Versions
SV_WAIT_SIG(D3)	Sleep until a synchronization variable is signalled or a signal is received.	page 220	SV*, 5.3*
timeout(D3)	Schedule a function to be executed after a specified number of clock ticks.	page 214	SV, 5.3
TRYLOCK(D3)	Try to acquire a basic lock, returning a code if the lock is not currently free.	page 205	SV*, 5.3*
uiomove(D3)	Copy data using *uio_t*.	page 194	SV, 5.3
uiophysio(D3)	Validate a raw I/O request and pass to a strategy function.	page 217	5.3
unbufcall(D3)	Cancel a pending bufcall request.		SV, 5.3
undma(D3)	Unlock physical memory in user space.	page 217	5.3
unfreezestr(D3)	Unfreeze the state of a stream.		SV, 5.3
unlinkb(D3)	Remove a message block from the head of a message.		SV, 5.3
UNLOCK(D3)	Release a basic lock.	page 205	SV*, 5.3*
untimeout(D3)	Cancel a previous itimeout or fast_itimeout request.	page 214	SV*, 5.3*
ureadc(D3)	Copy a character to space described by *uio_t*.	page 194	SV, 5.3
userdma(D3)	Lock physical memory in user space.small number of	page 217	5.3
userabi()	Get data sizes for the ABI of the user process (32- or 64-bit).	page 170	6.2
uwritec(D3)	Return a character from space described by *uio_t*.	page 194	SV, 5.3
v_getaddr(D3)	Get the user virtual address associated with a *vhandl_t*.	page 196	5.3
v_gethandle(D3)	Get a unique identifier associated with a *vhandl_t*.	page 196	5.3

Table A-4 (continued) Kernel Functions

Name	Summary	Discussed	Versions
v_getlen(D3)	Get the length of user address space associated with a *vhandl_t*.	page 196	5.3
v_mapphys(D3)	Map kernel address space into user address space.	page 196	5.3
valusema(D3)	Return the value associated with a semaphore.	page 222	5.3
vme_adapter(D3)	Determine VME adapter that corresponds to a given memory address.	page 330	5.3
vme_ivec_alloc(D3)	Allocate a VME bus interrupt vector.	page 332	5.3
vme_ivec_free(D3)	Free a VME bus interrupt vector.	page 332	5.3
vme_ivec_set(D3)	Register a VME bus interrupt vector.	page 332	5.3
vsema(D3)	Perform a "V" or signal semaphore operation.	page 222	5.3
wakeup(D3)	Waken a process waiting for an event.	page 219	SV, 5.3
wbadaddr(D3)	Test physical address for output.	page 195	SV, 5.3
wbadaddr_val(D3)	Test physical address for output of specific value.	page 195	SV, 5.3
WR(D3)	Get a pointer to the write queue.		SV, 5.3

The following SVR4 kernel functions are not implemented in IRIX: bioreset, dma_disable, dma_enable, dma_free_buf, dma_free_cb, dma_get_best_mode, dma_get_buf, dma_get_cb, dma_pageio, dma_prog, dma_swstart, dma_swsetup, drv_gethardware, hat_getkpfnum, hat_getppfnum, inb, inl, inw, kvtoppid, mod_drvattach, mod_drvdetach, outb, outl, outw, physmap, physmap_free, phystoppid, psignal, rdma_filter, repinsb, repinsd, repinsw, repoutsb, repoutsd, repoutsw, rminit, rmsetwant, SLEEP_LOCKOWNED, strncat, vtop.

Challenge DMA with Multiple IO4 Boards

In late 1995 a subtle hardware problem was identified in the IO4 board that is the primary I/O interface subsystem to systems using the Challenge/Onyx architecture. The problem can be prevented with a software fix. The software fix is included in all device drivers distributed with IRIX 6.2 and a software patch is available for IRIX 5.3. However, some third-party device drivers also need to incorporate the sofware fix. This appendix explains the IO4 problem as it affects device drivers produced outside Silicon Graphics.

The issue in a nutshell: if you are responsible for a kernel-level device driver for a DMA device for the Challenge/Onyx architecture, you probably need to insert a function call in the driver interrupt handler.

The IO4 Problem

The IO4 hardware problem involves a subtle interaction between two IO4 boards when they perform DMA to the identical cache line of memory. If one IO4 performs a partial update of a 128-byte cache line, and another IO4 accesses the same cache line for DMA between partial updates, the second IO4 can suffer a change to a different, unrelated cache line in its on-board cache. That modified cache line may not be used again, but if it is used, invalid data can be transferred.

It is important to note that the IO4 problem is specific to interactions between multiple IO4 boards. It does not affect memory interactions between CPUs, or between CPUs and IO4s. Cache coherency is properly maintained in these cases.

An unusual coincidence is required to trigger the modification of the IO4 cache memory; then the modified cache line must be used for output before the error has any effect. The right combinations are sufficiently rare that many systems with multiple IO4 boards have never enountered it. For example, the problem has occurred on a system that acted as a network gateway between ATM and FDDI network, with ATM and FDDI adapters on different IO4 boards; and it has been seen when “raw” (not filesystem) disk input was copied to a tape on a different IO4.

Software Fix

The software solution involves a number of behind-the-scenes changes to kernel functions that manage I/O mapping. However, for third-party device drivers, the fix to the IO4 problem consists of ensuring that any IO4 doing DMA input (when a device uses DMA to write to memory) flushes its cache on any interrupt. This change has been made in IRIX 6.2 to all device drivers supplied by Silicon Graphics.

A patch containing all necessary fixes is available for IRIX 5.3. Contact the Silicon Graphics technical support line for the current patch number for a particular system.

Software Not Affected

As a result of hardware design and software fixes, none of the following kinds of software are affected by the problem:

- Code using PIO to manage a device.

  The IO4 problem cannot be triggered by PIO, either at the user or kernel level.
- User-level code using the udmalib library or the dslib SCSI library.

  These libraries for user-level DMA contain the fix, or use kernel functions that are fixed.
- User-level code based on user-level interrupts (ULI) or external interrupts.

  These facilities are not relevant to the IO4 problem.

Among kernel-level drivers, only drivers that directly program DMA can be affected. STREAMS drivers are not affected; nor are pseudo-device drivers; nor are drivers that use only PIO and memory mapping. Drivers that are not used on Challenge-architecture machines are not affected; for example an EISA-bus driver cannot be affected.

SCSI drivers that use the host adapter interface (see “Host Adapter Facilities” on page 354) are also not affected. Silicon Graphics host adapter drivers contain the fix. Host adapter drivers from third parties may need to be fixed, but this does not affect drivers that rely on the host adapter interface.

Drivers that do only block-mode I/O for the filesystem, and do not implement a character I/O interface (or do not support the character I/O interface using DMA) are not affected. This is because the filesystem always requests I/O in cache-line-sized multiples to buffers that are cache-aligned.

Fixing the IO4 Problem

A kernel-level device driver for a device that uses DMA in a Challenge-architecture system probably needs to make one change to guard against the IO4 problem.

In order to preclude any chance of data corruption, drivers that are affected must ensure that the IO4 flushes its cache following any DMA write to memory (input from a device). This is done by calling a new kernel function, **io4_flush_cache()**, in the interrupt routine immediately following completion of any DMA.

The prototype of **io4_flush_cache()** is

```
int io4_flush_cache(caddr_t any_PIO_mapaddr);
```

The argument to the function is any value returned by **pio_mapaddr()** that is related to the device doing the DMA. The kernel uses this address to locate the IO4 involved. The returned value is 0 when the operation is successful (or was not needed). It is 1 when the argument is not a valid address returned by **pio_mapaddr()**.

The function should be called immediately after the completion of a DMA input to memory. Typically the device produces an interrupt at the end of a DMA, and the function can be called from the interrupt hander. However, some devices can complete more than one DMA transaction per interrupt, and **io4_flush_cache()** should be called when each DMA completes. Put another way, if it is possible that a data transfer completed after an interrupt, then the driver should call **io4_flush_cache()** before marking the transaction as complete.

The **io4_flush_cache()** function does nothing and returns immediately in a machine that has only one IO4 board, and in a machine in which all IO4 boards have the hardware fix.

The kernel's VME interrupt handler calls **io4_flush_cache()** once on each VME interrupt. Thus a VME device driver only needs to call **io4_flush_cache()** in the event that it handles the completion of more than DMA transaction per interrupt. For example, a VME-based network driver that handles multiple packets per interrupt should call **io4_flush_cache()** once for each packet that completes.

Since this problem only affects Challenge/Onyx systems (including POWER Challenge, POWER Onyx, and POWER Challenge R10000), the software fix can and should be conditionally compiled on the compiler variable EVEREST, which is set by */var/sysgen/Makefile.kernio* for the affected machines (see "Using /var/sysgen/Makefile.kernio" on page 228).

The following is a skeletal example of fix code for a hypothetical driver:

```
#ifdef EVEREST
extern void io4_flush_cache(void* anyPIOMapAddr);
#endif
caddr_t some_PIO_map_addr;
hypothetical_edtinit(...)
{
...
   some_PIO_map_addr = pio_mapaddr(my_piomap, some_dev_addr)
...
}
hypothetical_intr(...)
{
...
#ifdef EVEREST
   io4_flush_cache(some_PIO_map_addr);
#endif
...
}
```

For another example, see the code of the example VME device driver under "Sample VME Device Driver" on page 334.

Glossary

ABI

Application Binary Interface, a defined interface that includes an *API*, but adds the further promise that a compiled object file will be portable; no recompilation will be required to move to any supported platform.

API

Application Programming Interface, a defined interface through which services can be obtained. A typical API is implemented as a set of callable functions and header files that define the data structures and specific values that the functions accept or return. The promise behind an API is that a program that compiles and works correctly will continue to compile and work correctly in any supported environment (however, recompilation may be required when porting or changing versions). See *ABI*.

big-endian

The hardware design in which the most significant bits of a multi-byte integer are stored in the byte with the lowest address. Big-endian is the default storage order in MIPS processors. Opposed to *little-endian*.

block

As a verb, to suspend execution of a process. *See* sleep.

block device

A device such as magnetic tape or a disk drive, that naturally transfers data in blocks of fixed size. Opposed to *character device*.

block device driver

Driver for a block device. A block device's driver is not allowed to support the ioctl(), read() or write() entry points, but does have a strategy() entry point. *See* character device driver.

bus-watching cache

A *cache memory* that is aware of bus activity and, when the I/O system performs a DMA write into physical memory or another CPU in a multiprocessor system modifies *virtual memory*, automatically invalidates any copy of the same data then in the cache. This hardware function eliminates the need for explicit data cache write back or invalidation by software.

cache coherency

The problem of ensuring that all cached copies of data are true reflections of the data in memory. The usual solution is to ensure that, when one copy is changed, all other copies are automatically marked as invalid so that they will not be used.

cache line

The unit of data when data is loaded into a *cache memory*. Typically 128 bytes in current CPU models.

cache memory

High-speed memory closely attached to a CPU, containing a copy of the most recently used memory data. When the CPU's request for instructions or data can be satisfied from the cache, the CPU can run at full rated speed. In a multiprocessor or when DMA is allowed, a *bus-watching cache* is needed.

character device

A device such as a terminal or printer that transfers data as a stream of bytes, or a device that can be treated in this way under some circumstances. For example, a disk (normally a *block device*) can be treated as a character device for purposes of reading diagnostic information.

character device driver

The kernel-level device driver for a *character device* transfers data in bytes between the device and the user program. A *STREAMS driver* works with a character driver. Note that a *block device* such as magnetic tape or disk drives can also support character access through a character driver. Each disk device, for example, is represented as two different device special files, one managed by a *block device driver* and one by a character device driver.

close

Relinquish access to a resource. The user process invokes the **close**() system call when it is finished with a device, but the system does not necessarily execute your *drv***close**() entry point for that device.

data structure

Contiguous memory used to hold an ordered collection of fields of different types. Any *API* usually defines several data structures. The most common data structure in the *DDI/DKI* is the *buf_t*.

DDI/DKI

Device Driver Interface/Device Kernel Interface; the formal API that defines the services provided to a device driver by the kernel, and the rules for using those services. DDI/DKI is the term used in the UNIX System V documentation. The IRIX version of the DDI/DKI is close to, but not perfectly compatible with, the System V interface.

deadlock

The condition in which two or more processes are blocked, each waiting for a lock held by the other. Deadlock is prevented by the rule that a driver upper-half entry point is not allowed to hold a lock while sleeping.

devflag

A public global flag word that characterizes the abilities of a device driver, including the flags D_MP, D_WBACK and D_OLD.

device driver

A software module that manages access to a hardware device, taking the device in and out of service, setting hardware parameters, transmitting data between memory and the device, sometimes scheduling multiple uses of the device on behalf of multiple processes, and handling I/O errors.

direct memory access

When a device reads or writes in memory, asynchronously and without specific intervention by a CPU. In order to perform DMA, the device or its attachment must have some means of storing a memory address and incrementing it, usually through *mapping registers*. The device writes to physical memory and in so doing can invalidate *cache memory*; a *bus-watching cache* compensates.

device number

Each *device special file* is identified by a pair of numbers: the *major device number* identifies the device driver that manages the device, and the *minor device number* identifies the device to the driver.

device special file

A filename in the */dev* directory that represents a hardware device. A device special file does not specify data on disk, but rather identifies a particular hardware unit and the device driver that handles it. The *inode* of the file contains the *device number* as well as permissions and ownership data.

downstream

The direction of STREAMS messages flowing through a write queue from the user process to the driver.

EISA bus

Enhanced Industry Standard Architecture, a bus interface supported by certain Silicon Graphics systems.

EISA Product Identifier (ID)

The four-byte product identifier returned by an EISA expansion board.

file handle

An integer returned by the **open**() kernel function to represent the state of an open file. When the file handle is passed in subsequent kernel services, the kernel can retrieve information about the file, for example, when the file is a *device special file*, the file handle can be associated with the major and minor *device number.*

gigabyte

See kilobyte.

GIO bus

Graphics I/O bus, a bus interface used on Indigo, Indigo^2, and Indy workstations.

I/O operations

Services that provide access to shared input/output devices and to the global data structures that describe their status. I/O operations open and close files and devices, read data from and write data to devices, set the state of devices, and read and write system data structures.

inode

The UNIX and IRIX disk object that represents the existence of a file. The inode records the owner and group IDs and permissions. For regular disk files, the inode distinguishes files from directories and has other data that can be set with chmod. For device special files, the inode contains the major and minor device numbers and distinguishes block from character files.

inter-process communication

System calls that allow a process to send information to another process. There are several ways of sending information to another process: signals, pipes, shared memory, message queues, semaphores, streams, or sockets.

interrupt

A hardware signal that causes a CPU to set aside normal processing and begin execution of an interrupt handler. An interrupt is parameterized by the type of bus and the *interrupt level*, and possibly with an *interrupt vector* number. The kernel uses this information to select the interrupt handler for that device.

interrupt level

A number that characterizes the source of an *interrupt*. The VME bus provides for seven interrupt levels. Other buses have different schemes.

interrupt priority level

The relative priority at which a bus or device requests that the CPU call an interrupt process. Interrupts at a higher level are taken first. The interrupt handler for an interrupt can only be preempted on its CPU by an interrupt handler for an interrupt of higher level.

interrupt vector

A number that characterizes the specific device that caused an *interrupt*. Most VME bus devices have a specific vector number set by hardware, but some can have their vector set by software.

ioctl

Control a character device. Character device drivers may include a "special function" entry point, *pfx***ioct**().

IRQ

Interrupt Request Input, a hardware signal that initiates an interrupt.

k0

Virtual address range within the kernel address space that is cached but not mapped by translation look-aside buffers. Also referred to as kseg0.

k1

Virtual address range within the kernel address space that is neither cached nor mapped. Also called kseg1.

k2

Virtual address range within the kernel address space that can be both cached and mapped by translation look-aside buffers. Also called kseg2.

kernel level

The level of privilege at which code in the IRIX kernel runs. The kernel has a private address space, not acceptable to processes at *user-level*, and has sole access to physical memory.

kilobyte (KB)

1,024 bytes, a unit chosen because it is both an integer power of 2 (2^{10}) and close to 1,000, the basic scale multiple of engineering quantities. Thus 1,024 KB, 2^{20}, is 1 megabyte (MB) and close to 1e6; 1,024 MB, 2^{30}, is 1 gigabyte (GB) and close to 1e9; 1,024 GB, 2^{40}, is 1 terabyte (TB) and close to 1e12. In the MIPS architecture using 32-bit addressing, the user segment spans 2 GB. Using 64-bit addressing, both the user segment and the range of physical addresses span 1 TB.

kseg*n*

See k0, k1, k2.

little-endian

The hardware design in which the least significant bits of a multi-byte integer are stored in the byte with the lowest address. Little-endian order is the normal order in Intel processors, and optional in MIPS processors. Opposed to *big-endian*. (These terms are from Swift's *Gulliver's Travels*, in which the citizens of Lilliput and Blefescu are divided by the burning question of whether one's breakfast egg should be opened at the little or the big end.)

lock

A data object that represents the exclusive right to use a resource. A lock can be implemented as a *semaphore* (q.v.) with a count of 1, but because of the frequency of use of locks, they have been given distinct software support (see LOCK(D3)).

major device number

A number that specifies which device driver manages the device represented by a *device special file*. In IRIX 6.2, a major number has at most 9 bits of precision (0-511). Numbers 60-79 are used for OEM drivers. See also *minor device number.*

map

In general, to translate from one set of symbols to another. Particularly, translate one range of memory addresses to the addresses for the corresponding space in another system. The *virtual memory* hardware maps the process address space onto pages of physical memory. The *mapping registers* in a *DMA* device map bus addresses to physical memory corresponding to a buffer. The mmap(2) system call maps part of process address space onto the contents of a file.

mapping registers

Registers in a DMA device or its bus attachment that store the address translation data so that the device can access a buffer in physical memory.

megabyte

See kilobyte.

minor device number

A number that, encoded in a *device special file*, identifies a single hardware unit among the units managed by one device driver. Sometimes used to encode device management options as well. In IRIX 6.2, a minor number may have up to 18 bits of precision. *See also* major device number.

mmapped device driver

A driver that supports mapping hardware registers into process address space, permitting a user process to access device data as if it were in memory.

module

A STREAMS module consists of two related queue structures, one for upstream messages and one for downstream messages. One or more modules may be pushed onto a stream between the stream head and the driver, usually to implement and isolate a communication protocol or a line discipline.

open

Gain access to a device. The kernel calls the *pfx***open**() entry when the user process issues an **open**() system call.

page

A block of virtual or physical memory, of a size set by the operating system and residing on a page-size address boundary. The page size is 4,096 (2^{12}) bytes when in 32-bit mode; the page size in 64-bit mode can range from 2^{12} to 2^{20} at the operating system's choice (see the getpagesize(2) reference page).

PIO

Programmed I/O, meaning access to a VME device by mapping device registers into process address space, and transferring data by storing and loading single bytes or words.

poll

Poll entry point for a non-stream character driver. A character device driver may include a *drv***poll**() entry point so that users can use **select**(2) or **poll**(2) to poll the file descriptors opened on such devices.

prefix

Driver prefix. The name of the driver must be the first characters of its standard entry point names; the combined names are used to dynamically link the driver into the kernel. Specified in the *master.d* file for the driver. Throughout this manual, the prefix *pfx* represents the name of the device driver, as in *pfx***open**(), *pfx***ioctl**().

primary cache

The *cache memory* most closely attached to the CPU execution unit, usually in the processor chip.

primitives

Fundamental operations from which more complex operations can be constructed.

priority inheritance

An implementation technique that prevents *priority inversion* when a process of lower priority holds a mutual exclusion *lock* and a process of higher priority is blocked waiting for the lock. The process holding the lock “inherits” or acquires the priority of the highest-priority waiting process in order to expedite its release of the lock. IRIX supports priority inheritance for mutual exclusion locks only.

priority inversion

The effect that occurs when a low-priority process holds a *lock* that a process of higher priority needs. The lower priority process runs and the higher priority process waits, inverting the intended priorities. *See* priority inheritance.

process control

System calls that allow a process to control its own execution. A process can allocate memory, lock itself in memory, set its scheduling priorities, wait for events, execute a new program, or create a new process.

protocol stack

A software subsystem that manages the flow of data on a communications channel according to the rules of a particular protocol, for example the TCP/IP protocol. Called a “stack” because it is typically designed as a hierarchy of layers, each supporting the one above and using the one below.

pseudo-device

Software that uses the facilites of the *DDI/DKI* to provide specialized access to data, without using any actual hardware device. Pseudo-devices can provide access to system data structures that are unavailable at the user-level. For example, the fsctl driver gives superuser access to filesystem data (see fsctl(7)) and the inode monitor pseudo-device allows access to file activity (see imon(7)).

read

Read data from a device. The kernel executes the *pfx***read**() entry point whenever a user process calls the **read**() system call.

scatter/gather

An I/O operation in which what to the device is a contiguous range of data is distributed across multiple pages that may not be contiguous in physical memory. On input to memory, the device scatters the data into the different pages; on output, the device gathers data from the pages.

SCSI

Small Computer System Interface, the bus architecture commonly used to attach disk drives and other block devices.

SCSI driver interface

A collection of machine-independent input/output controls, functions, and data structures, that provides a standard interface for writing a SCSI driver.

semaphore

A data object that represents the right to use a limited resource, used for synchronization and communication between asynchronous processes. A semaphore contains a count that represents the quantity of available resource (typically 1). The **P** operation (mnemonic: de*P*lete) decrements the count and, if the count goes negative, causes the caller to wait (see psema(D3X), cpsema(D3X)). The **V** operation (mnemonic: re*V*ive) increments the count and releases any waiting process (see vsema(D3X), cvsema(D3X)). *See also* lock.

signals

Software interrupts used to communicate between processes. Specific signal numbers can be handled or blocked. Device drivers sometimes use signals to report events to user processes. Device drivers that can wait have to be sensitive to the possibility that a signal could arrive.

sleep

Suspend process execution pending occurrence of an event. The term “block” is also used.

socket

A software structure that represents one endpoint in a two-way communications link. Created by socket(2).

spl

Set priority level, a function that was formerly part of the *DDI/DKI*, and used to lock or allow interrupts on a processor. It is not possible to use spl effectively in a multiprocessor system, so it has been superceded by more sophisticated means of synchronization such as the *lock* and *semaphore*.

strategy

In general, the plan or policy for arbitrating between multiple, concurrent requests for the use of a device. Specifically in disk device drivers, the policy for scheduling multiple, concurrent disk block-read and block-write requests.

STREAM

A linked list of kernel data structures that provide a full-duplex data path between a user process and a device. Streams are supported by the STREAMS facilities in UNIX System V Release 3 and later.

STREAM head

The stream head, which is inserted by the STREAMS subsystem, processes STREAMS-related system calls and performs data transfers between user space and kernel space. It is the component of a stream closest to the user process. Every stream has a stream head.

STREAMS

A kernel subsystem used to build a stream, which is a modular, full-duplex data path between a device and a user process. In IRIX 5.x and later, the TCP/IP stack sits on top of the STREAMS stack. The Transport Layer Interface (TLI) is fully supported.

STREAMS driver

A software module that implements one stage of a STREAM. A STREAMS driver can be “pushed on” or “popped off” any *STREAM*.

TCP/IP

Transmission Control Protocol/Internet Protocol.

terabyte

See *kilobyte (KB)*.

TFP

The internal code name for the MIPS R8000 processor, used in some Silicon Graphics publications.

TLI

Transport Interface Layer.

user-level

The privilege level of the system at which user-initiated programs run. A user-level process can access the contents of one address space, and can access files and devices only by calling kernel functions. Contrast to *kernel level*.

unmap

Disconnect a memory-mapped device from user process space, breaking the association set by mapping it.

VME bus

VERSA Module Eurocard bus, a bus architecture supported by the Silicon Graphics Challenge and Onyx systems.

VME-bus adapter

A hardware conduit that translates host CPU operations to VME-bus operations and decodes some VME-bus operations to translate them to the host side.

virtual memory

Memory contents that appear to be in contiguous addresses, but are actually mapped to different physical memory locations by hardware action of the translation lookaside buffer (TLB) and page tables managed by the IRIX kernel. The kernel can exploit virtual memory to give each process its own address space, and to load many more processes than physical memory can support.

virtual page number

The most significant bits of a virtual address, which select a *page* of memory. The processor hardware looks for the VPN in the TLB; if the VPN is found, it is translated to a physical page address. If it is not found, the processor traps to an exception routine.

volatile

Subject to change. The volatile keyword informs the compiler that a variable could change value at any time (because it is mapped to a hardware register, or because it is shared with other, concurrent processes) and so should always be loaded before use.

wakeup

Resume suspended process execution.

write

Write data to a device. The kernel executes the *pfx***read**() or *pfx***write**() entry points whenever a user process calls the **read**() or **write**() system calls.

Index

Numerics

32-bit address space
  See address space, 32-bit
64-bit address space
  See address space, 64-bit
64-bit mode 24

A

address exception 9
address space
  32-bit 14–18
    embedding in 64-bit 20
    kseg0 17
    kseg1 18
    kseg2 17
    kuseg 16
    segments of 15
    virtual mapping 16
  64-bit 18–24
    cache-controlled 22
    segments of 18–24
    sign extension 20
    virtual mapping 20
    xkseg 22
    xksseg 22
    xkuseg 21
  bus virtual 12
  data transfer between 192
  device address 4
  kernel 17, 22
    map to user 25
  locking in memory 125
  memory address 5
  physical 4–5, 199
  supervisor 22
  user process 16, 21
addressing 3–28
alternate console 247
ASSERT macro 252
audio not covered 77
Audio/Serial Option (ASO) 45
authorized binary interface (ABI) 170

B

*bdevsw*table 137
block device 45
  combined with character 55, 147
  driver must be MP-aware 141
  used when mounting filesystem 277
  versus character 33
buffer (buf_t)
  See data types, *buf_t*
bus adapter
  translates addresses 12
bus virtual address 12

C

cache 13–14, 581
  64-bit access 22
  alignment of buffers 201
  coherency 14
  control functions 200
  device access always uncached 10
  primary 7
  secondary 7
cache algorithm 23
cdevsw table 137
Challenge/Onyx
  directing interrupts 312
  DMA engine in 303
  IO4 board 313, 581–584
  limit on VME DMA space 331
  no uncached memory 27
  VME address windows 329
  VME bus address mapping in 306
  VME bus numbers 315
  VME design constraints 321
  VME hardware 313–323
character device 45
  combined with block 55, 147
  used with *mkfs* 276
  versus block 34
COFF file format not supported 228
command
  See IRIX commands
compiler options
  32-bit 230
  64-bit 231
  for loadable driver 238
  for network driver 397
compiler variables 229
configuration files 38–40
  /dev/MAKEDEV 227, 241
  /etc/inittab 247
  /etc/rc2.d 36
  /etc/rc2/S23autoconfig 239
  /usr/cpu/sysgen/IPnnboot 233
  /usr/lib/X11/input/config 40
  /var/sysgen/boot 39, 232
  /var/sysgen/Makefile.kernio 228
  /var/sysgen/master.d 38, 226, 232, 233–235
    dependencies 234
    example 278
    format 233, 238
    stubs 235
    variables 235
  /var/sysgen/master.d/mem 26
  */var/sysgen/mtune/** 39
  /var/sysgen/system 39, 232
    example 278
  /var/sysgen/system/irix.sm 63, 67
    for debugging 245
    for SCSI 356
  */var/sysgen/system/irix.sm*VME devices 311
configuration flags 234
configuring a driver
  loadable 237–240
  nonloadable 232–237
CPU 5–14
  device access 10
  IP26 27
  memory access by 6
  model number from inventory 31
  processors in 5
  type numbers 5
  watchpoint registers 259
Crimson
  VME bus address mapping in 306

D

D_MP flag 141, 174
D_MT flag 142

D_OLD flag 142, 146
D_WBACK flag 141
Data Link Provider Interface (DLPI) 391
data transfer 192–195
data types
  summary table 565
  buf_t 154, 183–185
    BP_ISMAPPED 184
    displaying 268
    for syncronization 175
    functions 199
    interrupt handling 166
    management 217
  cred_t 148, 203
  dev_t 35, 147, 180
  struct dsconf 88
  struct dsreq 82, 88
    ds_flags 83
    ds_msg 87
    ds_ret 85
    ds_status 87
  edt_t 145
  iovec_t 182
  lock_t 185, 206
  major_t 34, 180
  minor_t 35, 180
  mrlock_t 185, 212
  mutex_t 185, 208
  struct pollhead 155
  proc_t (not available) 203
  struct scsi_request 363, 368
  struct scsi_target_info 360
  sema_t 185
  sleep_t 210
  sv_t 185, 220
  uio_t 152, 182, 194
  __userabi_t 171
  vhandl_t 159, 196
debugging kernel 243–248
device access 10
device address 4
device number
  See major device number, minor device number
device special file 32–37
  /dev/dsk 36
  /dev/ei 114, 124
  /dev/kmem 25
  /dev/mem 25, 26
  /dev/mmem 26
  */dev/scsi/** 78–81
  */dev/tty** 45
  */dev/vme/** 125
  as normal file 33
  defining 227–228
  EISA mapping 68
  for user-level interrupt 124
  inode contents 33
  multiple names for 37
  name format 36, 79
  VME mapping 63
device special file*/dev/kmem* 26
digital media not covered 77
DIrect Memory Access (DMA)
  cache control 581
Direct Memory Access (DMA) 11, 52–53
  buffer alignment for 201
  cache control 200
  DMA engine for VME bus 303
  EISA bus slave 446
  EISA bus-master 444
  GIO bus 521
  IO4 hardware problem 581
  maximum size 202
  setting up 198–202
  user-level 70–76
  user-level SCSI 85
  VME bus 302, 305
disk volume header 244, 275
driver
  compiling 229–231, 397

configuring 232–240
debugging 243–269
examples
EISA 447–508
GIO bus 528–538
network 401–423
RAM drive 271–294
SCSI bus 370–374
VME 334–347
flag constant 140–142, 237, 542
initialization 143–145
lower half 54
prefix 136, 232
in master.d 38
process context 202
registration 239
types
GIO bus 511–538
types of xxxv, 41–57
block 45
character 45
EISA bus 427–508
kernel-level xxxv, 26, 45–57
layered 55
loadable 57
network 389–423
process-level xxxv
pseudo-device 51
SCSI bus 369–370
STREAMS xxxv, 46
upper half 54
in multiprocessor 172, 173
user-level 25, 41–45
See also entry points
See also loadable driver

driver debugging
alternate console 247
breakpoints 257
circular buffer output 250
lock metering 246
memory display 260
multiprocessor 253
setsym use 247
stopping during bootstrap 254
symbol lookup 256
symbols 245
symmon use 252
system log output 249

driver operations 46–53
DMA 52
ioctl 48
mmap 50
open 46
read 49
write 49

dslib library 90–103
function summary 90
data transfer options 85
doscsireq() 93
ds_ctostr() 95
ds_vtostr() 95
dsclose() 92
dsopen() 92
filldsreq() 93
fillg0cmd() 94
fillg1cmd() 94
inquiry12() 96
modeselect15() 96
modesense1a() 97
read08() 98
readcapacity25() 99
readextended28() 98
releaseunit17() 100
requestsense03() 100
reserveunit16() 100
senddiagnostic1d() 101
testunitready00() 102
write0a() 102
writeextended2a() 102

dsreq driver 78
data transfer options 85

DS_ABORT 89
DS_CONF 88
DS_RESET 90
exclusive open 81
flags 83
return codes 85
scatter/gather 85
struct dsconf 88
struct dsreq 82–88
ds_flags 83
ds_msg 87
ds_ret 85
ds_status 87

E

EISA bus 427–508
address mapping 434
address spaces 430
allocate DMA channel 444
allocate IRQ 442–443
byte order 431
card slots 434
configuring 435–438
DMA to bus master 444–445
example driver 447
interrupts 430, 435
kernel services 439–447
locked cycles 431
mapping into user process 42
overview 428–433
PIO bandwidth 69
PIO mapping 439–441
product identifier ??–433
product indentifier 431–??
request arbitration 430
user-level PIO 67–70
ELF object format 228
entry points
summary table 138, 564
close 149–150, 159, 544
devflag 140–142, 237
edtinit 144, 239, 517
halt 168
info 542
init 144, 239, 543
interrupt 163–166, 519
ioctl 150–151, 170
map 159–161
mmap 161–162
mversion 237
open 146–149, 543
mode flag 148
type flag 147
poll 155–157
and interrupts 166
print 169
read 152–153
size 149, 169
start 145, 239, 543
strategy 154
and interrupts 166
called from read or write 153
design models 217
unload 159, 167–168, 240
unmap 162
usage 139
write 152–153
example driver 271, 334, 370, 401, 447, 528
execution model 170–171
external interrupt 43, 114–119
generate 114
input is level-triggered 115
pulse widths 116
set pulse widths 117
user-level handler 127

F

fixed PIO map 328
fmodsw table 137
function
  See IRIX functions, kernel functions

G

GIO bus 511–538
  address space mapping 513
  configuring 514
  edtinit entry point 517
  example driver 528–538
  form factors 512
  interrupt handler 519
  kernel services 515–517
  memory parity checking with 526
  varieties of 512

H

hardware inventory 29–32
  adding entries to 32
  contents 30
  hinv displays 30
  network driver use 396
  software interface to 31
header files
  summary table 185
  dslib.h 90
  for network drivers 393
  sgidefs.h 25
  sys/cmnerr.h 249
  sys/debug.h 252
  sys/file.h 148
  sys/immu.h 197
  sys/major.h 34
  sys/open.h 147
  sys/param.h 183
  sys/poll.h 155
  sys/region.h 161
  sys/scsi.h 355
  sys/sema.h 185
  sys/sysmacros.h 34, 35, 197
  sys/types.h 25, 34, 35, 180
  sys/uio.h 182
  sys/var.h 39

I

idbg debugger 245–247, 262–269
  command line use 263
  command syntax 264–269
  configuring in kernel 246
  display I/O status 267
  display process data 265
  interactive mode 262
  invoking 262
  loading 262
  lock meter display 267
  log file output 263
  memory display 265
ide PROM monitor 244
include file
  See header files
INCLUDE statement 144, 236, 239
initialization 143–145
inode 33, 47
interrupt 54
  and strategy entry point 166
  associating to a driver 163
  concurrent with processing 172
  enabled during initialization 143
  latency 165
  on multiprocessor 165
  See also user-level interrupt (ULI)

inventory
  See hardware inventory
IO4 board 313
  multiple DMA problem 581–584
IP26 CPU 27
IPL statement 312
IRIX commands
  autoconfig 232, 236, 247
  dvhtool 244–245
  hinv 30
    and MAKEDEV 36, 227
    for CPU type 6
  install 37, 227, 241
  lboot 39
    builds switch tables 137
    driver prefix with 136
    loads SCSI driver 356
  mkfs 276
  mknod 37, 227, 241
  ml 57, 241
  mount 48, 146, 277
  nvram 248
  prtvtoc 275
  setsym 247
  systune 39, 240, 250
    max DMA size 202
    switch table size 137
  umount 149
  uname 6
  versions 244
IRIX functions
  close() 149
  endinvent() 31
  getinvent() 31
  getpagesize() 21
  ioctl() 43, 44, 48, 127
  kmem_alloc() 54
  mmap() 26, 50, 158–159
    EISA PIO 68
    VME PIO 64
  mpin() 125
  munmap() 162
  open() 46, 146
    with dsreq driver 81
  pio_badaddr() 327
  plock() 125
  poll() 155–156
  read() 49, 52
  setinvent() 31
  syslog() 249
  test_and_set() 129
  ULI_block_intr() 129
  ULI_register_ei() 127
  ULI_register_vme() 127
  ULI_sleep() 124, 128
  ULI_wakeup() 128
  write() 49, 52

J

jag (SCSI-to-VME adapter) 356
jag (SCSI-toVME) adapter 80

K

kernel address space
  driver runs in 26
  mapping to user space 25
kernel execution model 170
kernel functions
  summary table 566
  add_to_inventory() 32
  badaddr() 195
  bcopy() 193
  biodone() 166, 218
  bioerror() 166
  biowait() 218
  bp_mapin() 200

brelse() 190
bzero() 193
cmn_err() 249–251
  buffer output 250
  system log output 249
copyin() 151, 193
copyout() 151, 193
cvsema() 177
dki_dcache_inval() 201
dki_dcache_wb() 27, 201
dma_map() 331, 445
dma_mapaddr() 331, 445
dma_mapalloc() 330, 445
drv_getparm() 203
drv_priv() 148, 203
drvhztousec() 215
drvusectohz() 215
eisa_dmachan_alloc() 444
eisa_ivec_alloc() 442
fasthzto() 215
flushbus() 201
fubyte() 194
geteblk() 190
getemajor() 34, 181
geteminor() 181, 360
getinvent() 6
getnextpg() 200
getrbuf() 190
initnsema() 177
initnsema_mutex() (not supported) 223
ip26_enable_ucmem() 28
ip26_return_ucmem() 28
itimeout() 156, 215
kern_malloc() (obsolete) 187
kmem_alloc() 17, 187
kmem_zalloc() 189
makedevice() 181
phalloc() 155, 189
phfree() 167, 189
physiock() 141, 153
pio_baddr() 441
pio_bcopyin() 328, 441
pio_bcopyout() 328, 441
pio_map_alloc() 447
pio_mapaddr() 328
pio_mapalloc() 326, 440
pio_mapfree() 167
pio_wbadaddr() 328
pollwakeup() 156, 166
pptophys() 200
printf() 251
psema() 177, 223
ptob() 21
rmalloc() 191
rmallocmap() 191
rmfree() 191
setgioconfig() 516
setgiovector() 515
sleep() 219
splhi()
  denigrated 213
  meaningless 172
splnet()
  ineffective 397
splvme()
  useless 176
subyte() 194
timeout() 215
uiomove() 194
uiophysio() 153
untimeout() 215
userabi() 170
v_getaddr() 196
v_gethandle() 196
v_mapphys() 160, 196
vme_ivec_alloc() 333
vme_ivec_free() 334
vme_ivec_set() 333

vsema() 177, 223
vt_gethandle() 161, 163
wakeup() 219
kernel mode of processor 8
kernel panic
address exception 9
moving data 192
kernel switch tables 137
kernel-level driver xxxv, 45–57, 135–177
structure of 136

L

layered driver 55
lboot
See IRIX commands
libc reentrant version 123
loadable driver 57
and switch table 137
autoregister 143
compiler options 238
configuring 237
initialization 143
loading 239
master.d 238
mversion entry 237
not in miniroot 57
registration 239
unloading 240
loading a driver 239
lock metering support 246, 267
locking
See mutual exclusion
locking memory 125
lower half of driver 54

M

major device number 34, 180
available numbers 34
block vs. character 34
dynamic allocation 241
external versus internal 181
for STREAMS clone 552, 553
in */dev/scsi* 80
in inode 33
in master.d 38, 233
in variables in master.d 235
indexes switch table 137
input to open 47
range of 34
selecting 226
/dev/MAKEDEV 35–37, 78, 180, 227, 241
adding to */dev/scsi* 81
master.d configuration files
See configuration files,*/var/sysgen/master.d*
memory 3–28
memory address
cached 17
physical 5, 259, 434
uncached 18
memory allocation 187–191
memory display 260
memory mapping 25–26, 158–163
miniroot
no loadable drivers 57
minor device number 35, 180
encoding 35
external versus internal 181
for STREAMS clone driver 552, 553
in */dev/scsi* 80
in inode 33
input to open 47, 147
selecting 227

multiprocessor
  block driver must support 141
  converting to 176
  driver design for 171–177, 547
  driver flag D_MP 141, 174
  drivers for 56
  interrupt handling on 165
  network drivers in 396–??
  nonMP driver on CPU 0 141, 174
  splhi useless in 172
  synchronizing upper-half code 173
  uniprocessor assumptions invalid 171
  uniprocessor drivers use CPU 0 56
  using symmon in 253
mutex locks 207
mutual exclusion 204, 205–214
  basic locks 206–207
  in multiprocessor drivers 172
  in network driver 398–400
  mutex locks 207
  priority inheritance 208
  reader/writer locks 211
  semaphore 223
  sleep locks 209

N

names of devices 33, 36, 79
network 389
  based on 4.3BSD 392
  driver interfaces 392–396
  example driver 401
  header files 393
  multiprocessor considerations 396
  overview 390
  STREAMS protocol stack 391
network driver
  debugging 269
  must be MP-aware 141
Network File System (NFS) 391

P

page size
  I/O 197
  macros 197
  memory 21, 197
parity check with GIO 526
pipe semantics 549
prefix 38, 136, 232
primary cache 7
priority inheritance 208
priority level functions 213
privilege checking 203
process 202–203
  display data about 265
  handle of 203
  sending signal to 203
  table of in kernel 265
process-level driver xxxv
processor
  kernel mode 8
  types 5
  user mode 8
Programmed I/O (PIO) 10, 61
  address maps for 325–329
  EISA bus 67–70, 439
  GIO bus 519
  VME bus 62–66, 302, 305
pseudo-device driver 51
putbuf circular buffer 250, 265

R

RAM drive 271
raw device
  See character device
reader/writer locks 211
reentrant C library 123
registration of loadable driver 239

S

sash standalone shell 244
SCSI bus 351–386
  adapter error codes 378
  adapter number 79, 359
  adapter type number 359, 376
  command
    Inquiry 96, 360
    Mode Select 96
    Mode Sense 97
    Read 98
    Read Capacity 99
    Request Sense 100
    Reserve Unit 100
    Send Diagnostic 101
    Test Unit Ready 102
    Write 102
  display request structure 267
  driver 369–370
  error messages 377–386
  example driver 370–374
  hardware support overview 352
  host adapter 353
    functions of 357
    intialization 375
    number of 355
    overview 375
    purpose 354
    scsi_abort() 368
    scsi_alloc() 361
    scsi_command() 363
    scsi_free() 362
    scsi_info() 360
    scsi_reset() 369
    vectors to 358, 376
  kernel overview 353
  LUN 79, 356
  message string tables 378
  sense codes 379
  target ID 79
  target number 356
  user-level access 43, 77–103
  See also dsreq driver
secondary cache 7
sector unit macros 197
semaphore 222–224
  for mutual exclusion 223
  for waiting 224
sign extension of 32-bit addresses 20
signal 203
SIGSEGV 9
Silicon Graphics
  developer program xxxvii
  FTP server xxxvii
  VME bus hardware 300
  WWW server xxxvii
64-bit address space
  See address space, 64-bit
64-bit entries *see*Numbers
sleep locks 209
socket interface 391
stray VME interrupt 334
STREAMS 541–561
  function summary 554
  clone driver 552–553
  close entry point 544
  debugging 268
  display data structures 268

driver xxxv
extended poll support 549
module_info structure 542
multiprocessor design 547
multithreaded monitor 547
open entry point 543
put functions 544
service scheduling 550
srv functions 545
streamtab structure 542
supplied drivers 550
STREAMS protocol stack 391
structure of driver 136
switch table 137
symmon debugger 244–245, 252–261
breakpoints 257
command syntax 255–256
how invoked 253
in multiprocessor 253
in uniprocessor 253
invoking at bootstrap 254
memory display 260
prompt 253
symbol lookup 256
virtual memory commands 259
watchpoint register use 259
synchronization variable 220
SysAD bus parity checks 526
sysgen files
See configuration files
system console
alternate 247
system log display 249
systune
See IRIX commands

T

terminal as console 247
The 269
32-bit entries *see* Numbers
tick 215
time unit functions 215
TLI interface 391
Translate Lookaside Buffer (TLB) 7
Translation Lookaside Buffer (TLB) 8
maps kernel space 22
maps kuseg 17
number of entries in 9

U

udmalib 70–76
dma_allocbuf() 71
dma_mkparms() 72
dma_start() 73
uncached memory access
32-bit 18
64-bit 22
do not map 160
IP26 27
none in Challenge 27
uniprocessor
converting driver 176
using symmon 253
unloading a driver 240
upper half of driver 54
upper half of of driver 172, 173
USE statement 144, 236
user mode of processor 8

user-level DMA 70–76
user-level driver 41–45
user-level interrupt (ULI) 44, 121–132
  and debugging 123
  external interrupt with 127
  initializing 124
  interrupt handler function 122–124
  registration 126
  restrictions on handler 122
  ULI_block_intr() function 129
  ULI_register_ei() function 127
  ULI_register_vme() function 127
  ULI_sleep() function 124, 128
  ULI_wakeup() function 128
  VME interrupt with 127
user-level process 41

V

variables in master.d 235
VECTOR statement 163, 236, 239
  edtinit entry point 144
  EISA kernel driver 436
  EISA PIO 67
  for SCSI host adapter 356
  GIO bus 514
  use of ctlr= 147
  use of probe= 437
  VME devices 311
  VME PIO 63
vfssw table 137
virtual memory 7, 8–10
  32-bit mapping 16
  64-bit mapping 20
  debug display of 259
  page size 21
virtual page number (VPN)
  32-bit 16
VME bus 297–347
  adapter number 80
  bus address spaces 299, 304–309
    mapping 304
  bus cycles 300
  bus numbers in Challenge 315
  configuring 310
  DMA engine 303
  DMA to
    address maps 330–331
    addresses 308
    addressing in Challenge 308
    addressing in Crimson 308–309
  example driver 334
  hardware
    Challenge 313–323
    design constraints 321
    DMA cycle 302
    interrupt priority 321
    overview 300–303
    PIO cycle 302
    relation to system bus 301
  history 298
  interrupt levels 300
  interrupt vector 332
  jag adapter 80
  kernel services 325–347
  mapping into user process 42
  master device 299
  PIO to
    address maps 325–329
    addresses 305
    addressing in Challenge 306
    addressing in Crimson 306
    bandwidth 66
    fixed, unfixed maps 328

slave device 299
stray interrupt cause 334
user-level DMA 43, 70–76
user-level DMA bandwidth 73
user-level interrupt handler for 127
user-level PIO 62–66
VME Cache Controller (VMECC) 315–317
VMEbus Channel Adapter Module (VCAM) board 313, 315
volatile keyword 123
volume header 244, 275

W

waiting 204, 214–222
for a general event 218
for an interrupt 217
for memory 216
semaphore 224
synchronization variables 220
time units 215
timed events 215

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-0911-060.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 415-965-0964
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043-1389