Writing Device Drivers
Ce logiciel et la documentation qui l’accompagne sont protégés par les lois sur la propriété intellectuelle. Ils sont concédés sous licence et soumis à des restrictions d’utilisation et de divulgation. Sauf disposition de votre contrat de licence ou de la loi, vous ne pouvez pas copier, reproduire, traduire, diffuser, modifier, breveter, transmettre, distribuer, exposer, exécuter, publier ou afficher le logiciel, même partiellement, sous quelque forme et par quelque procédé que ce soit. Par ailleurs, il est interdit de procéder à toute ingénierie inverse du logiciel, de le désassembler ou de le décompiler, excepté à des fins d’interopérabilité avec des logiciels tiers ou tel que prescrit par la loi.

Les informations fournies dans ce document sont susceptibles de modification sans préavis. Par ailleurs, Oracle Corporation ne garantit pas qu’elles soient exemptes d’erreurs et vous invite, à lui en faire part par écrit.

Si ce logiciel, ou la documentation qui l’accompagne, est concédé sous licence au Gouvernement des États-Unis, ou à toute entité qui délivre la licence de ce logiciel ou l’utilise pour le compte du Gouvernement des États-Unis, la notice suivante s’applique:

U.S. GOVERNMENT END USERS. Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are “commercial computer software” pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

Ce logiciel ou matériel a été développé pour un usage général dans le cadre d’applications de gestion des informations. Ce logiciel ou matériel n’est pas conçu ni n’est destiné à être utilisé dans des applications pouvant causer des dommages corporels. Si vous utilisez ce logiciel ou matériel dans le cadre d’applications dangereuses, il est de votre responsabilité de prendre toutes les mesures de securité, de sauvegarde, de redondance et autres mesures nécessaires à son utilisation dans des conditions optimales de sécurité. Oracle Corporation et ses affiliés déclinent toute responsabilité quant aux dommages causés par l’utilisation de ce logiciel ou matériel pour ce type d’applications.

Oracle et Java sont des marques déposées d’Oracle Corporation et/ou de ses affiliés. Tout autre nom mentionné peut correspondre à des marques appartenant à d’autres propriétaires qu’Oracle.

Intel et Intel Xeon sont des marques ou des marques déposées d’Intel Corporation. Toutes les marques SPARC sont utilisées sous licence et sont des marques ou des marques déposées d’SPARC International, Inc. AMD, Opteron, le logo AMD et le logo AMD Opteron sont des marques ou des marques déposées d’Advanced Micro Devices. UNIX est une marque déposée d’The Open Group.

Ce logiciel ou matériel et la documentation qui l’accompagne peuvent fournir des informations ou des liens donnant accès à des contenus, des produits et des services émanant de tiers. Oracle Corporation et ses affiliés déclinent toute responsabilité ou garantie expresse quant aux contenus, produits ou services émanant de tiers. En aucun cas, Oracle Corporation et ses affiliés ne sauraient être tenus pour responsables des pertes subies, des coûts occasionnés ou des dommages causés par l’accès à des contenus, produits ou services tiers, ou à leur utilisation.
## Contents

**Preface** ...................................................................................................................................................31

**Part I  Designing Device Drivers for the Oracle Solaris Platform** ............................................................37

1 **Overview of Oracle Solaris Device Drivers** ..........................................................................................39
   Device Driver Basics ..................................................................................................................39
   What Is a Device Driver? ...........................................................................................................39
   What Is a Device Driver Entry Point? .........................................................................................................40
   Device Driver Entry Points .......................................................................................................................41
      Entry Points Common to All Drivers ......................................................................................................41
      Entry Points for Block Device Drivers ................................................................................................44
      Entry Points for Character Device Drivers ............................................................................................45
      Entry Points for STREAMS Device Drivers .........................................................................................46
      Entry Points for Memory Mapped Devices ............................................................................................47
      Entry Points for Network Device Drivers ...............................................................................................48
      Entry Points for SCSI HBA Drivers ........................................................................................................48
      Entry Points for PC Card Drivers ..........................................................................................................49
   Considerations in Device Driver Design ..................................................................................................50
      DDI/DKI Facilities .................................................................................................................................50
      Driver Context ......................................................................................................................................52
      Returning Errors .................................................................................................................................53
      Dynamic Memory Allocation .................................................................................................................53
      Hotplugging .........................................................................................................................................54

2 **Oracle Solaris Kernel and Device Tree** ..............................................................................................55
   What Is the Kernel? .................................................................................................................................55
   Multithreaded Execution Environment ....................................................................................................57
Contents

Virtual Memory ........................................................................................................................... 57
Devices as Special Files ........................................................................................................... 57
DDI/DKI Interfaces ..................................................................................................................... 58
Overview of the Device Tree .................................................................................................... 59
Device Tree Components ....................................................................................................... 59
Displaying the Device Tree .................................................................................................. 60
Binding a Driver to a Device ................................................................................................. 62

3 Multithreading ....................................................................................................................... 67
Locking Primitives .................................................................................................................. 67
Storage Classes of Driver Data ............................................................................................. 67
Mutual-Exclusion Locks ......................................................................................................... 68
Readers/Writer Locks ............................................................................................................. 69
Semaphores ............................................................................................................................ 69
Thread Synchronization ......................................................................................................... 70
Condition Variables in Thread Synchronization .................................................................. 70
cv_wait() and cv_timedwait() Functions ............................................................................ 72
cv_wait_sig() Function ........................................................................................................... 73
cv_timedwait_sig() Function .................................................................................................. 73
Choosing a Locking Scheme .................................................................................................. 74
Potential Locking Pitfalls ....................................................................................................... 74
Threads Unable to Receive Signals ........................................................................................ 75

4 Properties ............................................................................................................................. 77
Device Properties .................................................................................................................. 77
Device Property Names ......................................................................................................... 78
Creating and Updating Properties ......................................................................................... 78
Looking Up Properties .......................................................................................................... 78
Changes to the driver.conf File ............................................................................................. 80
prop_op() Entry Point ............................................................................................................. 81

5 Managing Events and Queueing Tasks ............................................................................... 83
Managing Events .................................................................................................................... 83
Introduction to Events ........................................................................................................... 83
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using ddi_log_sys_event() to Log Events</td>
<td>85</td>
</tr>
<tr>
<td>Defining Event Attributes</td>
<td>86</td>
</tr>
<tr>
<td>Queueing Tasks</td>
<td>89</td>
</tr>
<tr>
<td>Introduction to Task Queues</td>
<td>89</td>
</tr>
<tr>
<td>Task Queue Interfaces</td>
<td>90</td>
</tr>
<tr>
<td>Observing Task Queues</td>
<td>90</td>
</tr>
<tr>
<td>6 Driver Autoconfiguration</td>
<td>93</td>
</tr>
<tr>
<td>Driver Loading and Unloading</td>
<td>93</td>
</tr>
<tr>
<td>Data Structures Required for Drivers</td>
<td>94</td>
</tr>
<tr>
<td>modlinkage Structure</td>
<td>95</td>
</tr>
<tr>
<td>modldrv Structure</td>
<td>95</td>
</tr>
<tr>
<td>dev_ops Structure</td>
<td>95</td>
</tr>
<tr>
<td>cb_ops Structure</td>
<td>96</td>
</tr>
<tr>
<td>Loadable Driver Interfaces</td>
<td>97</td>
</tr>
<tr>
<td>_init() Example</td>
<td>98</td>
</tr>
<tr>
<td>_fini() Example</td>
<td>99</td>
</tr>
<tr>
<td>_info() Example</td>
<td>100</td>
</tr>
<tr>
<td>Device Configuration Concepts</td>
<td>100</td>
</tr>
<tr>
<td>Device Instances and Instance Numbers</td>
<td>101</td>
</tr>
<tr>
<td>Minor Nodes and Minor Numbers</td>
<td>101</td>
</tr>
<tr>
<td>probe() Entry Point</td>
<td>101</td>
</tr>
<tr>
<td>attach() Entry Point</td>
<td>104</td>
</tr>
<tr>
<td>detach() Entry Point</td>
<td>109</td>
</tr>
<tr>
<td>getinfo() Entry Point</td>
<td>110</td>
</tr>
<tr>
<td>Using Device IDs</td>
<td>112</td>
</tr>
<tr>
<td>Registering Device IDs</td>
<td>112</td>
</tr>
<tr>
<td>Unregistering Device IDs</td>
<td>113</td>
</tr>
<tr>
<td>7 Device Access: Programmed I/O</td>
<td>115</td>
</tr>
<tr>
<td>Device Memory</td>
<td>115</td>
</tr>
<tr>
<td>Managing Differences in Device and Host Endianness</td>
<td>116</td>
</tr>
<tr>
<td>Managing Data Ordering Requirements</td>
<td>116</td>
</tr>
<tr>
<td>ddi_device_acc_attr Structure</td>
<td>116</td>
</tr>
<tr>
<td>Mapping Device Memory</td>
<td>117</td>
</tr>
</tbody>
</table>
8 **Interrupt Handlers** ........................................................................................................... 121
   Interrupt Handler Overview ............................................................................................. 121
   Device Interrupts ............................................................................................................. 122
      High-Level Interrupts .................................................................................................... 122
      Legacy Interrupts .......................................................................................................... 123
      Standard and Extended Message-Signaled Interrupts .................................................. 123
      Software Interrupts ....................................................................................................... 124
   DDI Interrupt Functions .................................................................................................. 125
      Interrupt Capability Functions ...................................................................................... 125
      Interrupt Initialization and Destruction Functions ..................................................... 125
      Priority Management Functions .................................................................................. 126
      Soft Interrupt Functions .............................................................................................. 126
      Interrupt Function Examples ...................................................................................... 127
   Registering Interrupts ...................................................................................................... 128
      Registering Legacy Interrupts ....................................................................................... 128
      Registering MSI Interrupts ........................................................................................... 131
   Interrupt Resource Management .................................................................................... 134
      The Interrupt Resource Management Feature ........................................................... 134
      Callback Interfaces ...................................................................................................... 135
      Interrupt Request Interfaces ........................................................................................ 137
      Example Implementation of Interrupt Resource Management .................................... 139
   Interrupt Handler Functionality ..................................................................................... 145
   Handling High-Level Interrupts ....................................................................................... 147
      High-Level Mutexes ..................................................................................................... 147
      High-Level Interrupt Handling Example ..................................................................... 147

9 **Direct Memory Access (DMA)** .................................................................................... 153
   DMA Model ..................................................................................................................... 153
   Types of Device DMA .................................................................................................. 154
      Bus-Master DMA .......................................................................................................... 154
      Third-Party DMA ....................................................................................................... 154
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>First-Party DMA</td>
<td>155</td>
</tr>
<tr>
<td>Types of Host Platform DMA</td>
<td>155</td>
</tr>
<tr>
<td>DMA Software Components: Handles, Windows, and Cookies</td>
<td>155</td>
</tr>
<tr>
<td>DMA Operations</td>
<td>156</td>
</tr>
<tr>
<td>Performing Bus-Master DMA Transfers</td>
<td>156</td>
</tr>
<tr>
<td>Performing First-Party DMA Transfers</td>
<td>157</td>
</tr>
<tr>
<td>Performing Third-Party DMA Transfers</td>
<td>157</td>
</tr>
<tr>
<td>DMA Attributes</td>
<td>157</td>
</tr>
<tr>
<td>Managing DMA Resources</td>
<td>161</td>
</tr>
<tr>
<td>Object Locking</td>
<td>161</td>
</tr>
<tr>
<td>Allocating a DMA Handle</td>
<td>161</td>
</tr>
<tr>
<td>Allocating DMA Resources</td>
<td>162</td>
</tr>
<tr>
<td>Determining Maximum Burst Sizes</td>
<td>164</td>
</tr>
<tr>
<td>Allocating Private DMA Buffers</td>
<td>165</td>
</tr>
<tr>
<td>Handling Resource Allocation Failures</td>
<td>167</td>
</tr>
<tr>
<td>Programming the DMA Engine</td>
<td>167</td>
</tr>
<tr>
<td>Freeing the DMA Engine</td>
<td>168</td>
</tr>
<tr>
<td>Freeing the DMA Handle</td>
<td>169</td>
</tr>
<tr>
<td>Canceling DMA Callbacks</td>
<td>169</td>
</tr>
<tr>
<td>Synchronizing Memory Objects</td>
<td>171</td>
</tr>
<tr>
<td>DMA Windows</td>
<td>173</td>
</tr>
</tbody>
</table>

10 Mapping Device and Kernel Memory ........................................... 177
   Memory Mapping Overview                                          | 177  |
   Exporting the Mapping                                             | 177  |
      The segmap(9E) Entry Point                                      | 177  |
      The devmap(9E) Entry Point                                      | 179  |
   Associating Device Memory With User Mappings                      | 181  |
   Associating Kernel Memory With User Mappings                      | 182  |
      Allocating Kernel Memory for User Access                       | 183  |
      Exporting Kernel Memory to Applications                        | 185  |
      Freeing Kernel Memory Exported for User Access                 | 186  |

11 Device Context Management ..................................................... 187
   Introduction to Device Context                                      | 187  |
## Contents

What Is a Device Context? ................................................................. 187  
Context Management Model ............................................................. 187  
Context Management Operation ....................................................... 189  
\texttt{devmap\_callback\_ctl} Structure ............................................. 189  
Enter Points for Device Context Management .............................. 190  
Associating User Mappings With Driver Notifications ................... 197  
Managing Mapping Accesses ......................................................... 198  

### 12 Power Management ................................................................. 201  
Power Management Framework ..................................................... 201  
Device Power Management ........................................................... 202  
System Power Management ........................................................... 202  
Device Power Management Model ................................................ 203  
Power Management Components .................................................. 203  
Power Management States ............................................................. 204  
Power Levels .................................................................................. 204  
Power Management Dependencies ................................................ 205  
Automatic Power Management for Devices ................................. 206  
Device Power Management Interfaces .......................................... 206  
\texttt{power()} Entry Point ........................................................... 208  
System Power Management Model ................................................ 210  
Autoshutdown Threshold ............................................................... 211  
Busy State ..................................................................................... 211  
Hardware State ............................................................................ 211  
Automatic Power Management for Systems ................................... 211  
Enter Points Used by System Power Management .......................... 211  
Power Management Device Access Example ................................. 215  
Power Management Flow of Control ............................................. 216  
Changes to Power Management Interfaces .................................... 218  

### 13 Hardening Oracle Solaris Drivers ........................................... 221  
Oracle Fault Management Architecture I/O Fault Services ............... 221  
What Is Predictive Self-Healing? .................................................... 222  
Oracle Solaris Fault Manager ......................................................... 222  
Error Handling ............................................................................. 225
14 Layered Driver Interface (LDI) ......................................................................................................... 235
  LDI Overview ......................................................................................................................................... 235
  Kernel Interfaces ...................................................................................................................................... 236
    Layered Identifiers – Kernel Device Consumers .............................................................................. 236
    Layered Driver Handles – Target Devices ...................................................................................... 237
    LDI Kernel Interfaces Example ....................................................................................................... 241
  User Interfaces ....................................................................................................................................... 251
    Device Information Library Interfaces ............................................................................................ 251
    Print System Configuration Command Interfaces ........................................................................ 253
    Device User Command Interfaces ................................................................................................... 256

Part II Designing Specific Kinds of Device Drivers ................................................................................. 259

15 Drivers for Character Devices ............................................................................................................ 261
  Overview of the Character Driver Structure ....................................................................................... 261
  Character Device Autoconfiguration .................................................................................................... 263
  Device Access (Character Drivers) .......................................................................................................... 264
    open() Entry Point (Character Drivers) ............................................................................................ 264
    close() Entry Point (Character Drivers) ............................................................................................ 266
  I/O Request Handling ............................................................................................................................ 266
    User Addresses ..................................................................................................................................... 266
    Vectored I/O .......................................................................................................................................... 267
    Differences Between Synchronous and Asynchronous I/O .................................................................. 269
  Data Transfer Methods .......................................................................................................................... 269
  Mapping Device Memory ....................................................................................................................... 275
  Multiplexing I/O on File Descriptors ................................................................................................... 276
  Miscellaneous I/O Control ....................................................................................................................... 278
    ioctl() Entry Point (Character Drivers) .............................................................................................. 278
    I/O Control Support for 64-Bit Capable Device Drivers ..................................................................... 280
    Handling copyout() Overflow ............................................................................................................. 282
  32-bit and 64-bit Data Structure Macros ............................................................................................... 283
    How Do the Structure Macros Work? .................................................................................................. 284
    When to Use Structure Macros .......................................................................................................... 284
    Declaring and Initializing Structure Handles ..................................................................................... 284
    Operations on Structure Handles ....................................................................................................... 285
### 16 Drivers for Block Devices

- Block Driver Structure Overview ...........................................................................287
- File I/O .....................................................................................................................288
- Block Device Autoconfiguration .............................................................................289
- Controlling Device Access ....................................................................................291
  - `open()` Entry Point (Block Drivers) .................................................................291
  - `close()` Entry Point (Block Drivers) .................................................................292
  - `strategy()` Entry Point ....................................................................................293
  - `buf` Structure ....................................................................................................293
- Synchronous Data Transfers (Block Drivers) ..........................................................295
- Asynchronous Data Transfers (Block Drivers) .........................................................299
  - Checking for Invalid `buf` Requests ...................................................................299
  - Enqueuing the Request .......................................................................................299
  - Starting the First Transfer ..................................................................................300
  - Handling the Interrupting Device .....................................................................302
- `dump()` and `print()` Entry Points ...................................................................303
  - `dump()` Entry Point (Block Drivers) .................................................................303
  - `print()` Entry Point (Block Drivers) .................................................................303
- Disk Device Drivers ...............................................................................................304
  - Disk `ioctl`s .........................................................................................................304
  - Disk Performance ...............................................................................................304

### 17 SCSI Target Drivers

- Introduction to Target Drivers ................................................................................307
- Sun Common SCSI Architecture Overview ............................................................307
  - General Flow of Control .....................................................................................309
- SCSA Functions ......................................................................................................310
- Hardware Configuration File ..................................................................................311
- Declarations and Data Structures ........................................................................311
  - `scsi_device` Structure .....................................................................................312
  - `scsi_pkt` Structure (Target Drivers) .................................................................313
- Autoconfiguration for SCSI Target Drivers ...........................................................315
  - `probe()` Entry Point (SCSI Target Drivers) ....................................................315
attach() Entry Point (SCSI Target Drivers) ................................................................. 317
detach() Entry Point (SCSI Target Drivers) ................................................................. 319
getinfo() Entry Point (SCSI Target Drivers) ............................................................... 320
Resource Allocation ..................................................................................................................320
scsi_init_pkt() Function .................................................................................................320
scsi_sync_pkt() Function .................................................................................................322
scsi_destroy_pkt() Function ............................................................................................322
scsi Alloc_consistent_buf() Function ...............................................................................322
scsi free_consistent_buf() Function ................................................................................322
Building and Transporting a Command ........................................................................... 323
Building a Command ......................................................................................................... 323
Setting Target Capabilities ............................................................................................... 324
Transporting a Command ................................................................................................. 324
Command Completion ....................................................................................................... 325
Reuse of Packets ................................................................................................................ 326
Auto-Request Sense Mode ............................................................................................... 327
Dump Handling .................................................................................................................. 328
SCSI Options ......................................................................................................................329

18 SCSI Host Bus Adapter Drivers .....................................................................................331
Introduction to Host Bus Adapter Drivers .......................................................................331
SCSI Interface .................................................................................................................. 332
SCSA HBA Interfaces ....................................................................................................... 334
SCSA HBA Entry Point Summary .................................................................................... 334
SCSA HBA Data Structures ...............................................................................................335
Per-Target Instance Data .................................................................................................. 341
Transport Structure Cloning ........................................................................................... 342
SCSA HBA Functions ....................................................................................................... 343
HBA Driver Dependency and Configuration Issues ....................................................... 344
Declarations and Structures ............................................................................................. 344
Entry Points for Module Initialization ............................................................................. 345
Autoconfiguration Entry Points ...................................................................................... 347
Entry Points for SCSA HBA Drivers ............................................................................... 350
Target Driver Instance Initialization .............................................................................. 351
Resource Allocation ........................................................................................................ 353
Contents

Command Transport .................................................................................................................. 362
Capability Management ............................................................................................................. 368
Abort and Reset Management ................................................................................................. 373
Dynamic Reconfiguration ........................................................................................................ 375
SCSI HBA Driver Specific Issues .......................................................................................... 376
Installing HBA Drivers ........................................................................................................... 376
HBA Configuration Properties ............................................................................................... 376
x86 Target Driver Configuration Properties ......................................................................... 377
Support for Queuing ............................................................................................................... 379

19 Drivers for Network Devices .............................................................................................. 381
GLDv3 Network Device Driver Framework ............................................................................ 381
GLDv3 MAC Registration ....................................................................................................... 382
GLDv3 Capabilities .................................................................................................................. 386
GLDv3 Data Paths ..................................................................................................................... 393
GLDv3 State Change Notifications ....................................................................................... 397
GLDv3 Network Statistics ....................................................................................................... 397
GLDv3 Properties .................................................................................................................... 399
Summary of GLDv3 Interfaces ............................................................................................... 399
GLDv2 Network Device Driver Framework ............................................................................ 403
GLDv2 Device Support ............................................................................................................ 405
GLDv2 DLPI Providers ............................................................................................................. 406
GLDv2 DLPI Primitives ............................................................................................................ 407
GLDv2 I/O Control Functions ................................................................................................. 408
GLDv2 Driver Requirements ................................................................................................. 409
GLDv2 Network Statistics ....................................................................................................... 410
GLDv2 Declarations and Data Structures ............................................................................. 414
GLDv2 Function Arguments ................................................................................................. 418
GLDv2 Entry Points ................................................................................................................. 419
GLDv2 Return Values .............................................................................................................. 423
GLDv2 Service Routines ......................................................................................................... 423

20 USB Drivers ....................................................................................................................... 427
USB in the Oracle Solaris Environment .................................................................................. 427
USBA 2.0 Framework ............................................................................................................. 427
USB Client Drivers ................................................................. 428
Binding Client Drivers ............................................................ 430
How USB Devices Appear to the System ............................... 430
USB Devices and the Oracle Solaris Device Tree .................. 430
Compatible Device Names ..................................................... 430
Devices With Multiple Interfaces ......................................... 432
Devices With Interface-Association Descriptors .................... 433
Checking Device Driver Bindings ........................................... 434
Basic Device Access .............................................................. 435
Before the Client Driver Is Attached ..................................... 435
The Descriptor Tree ............................................................... 435
Registering Drivers to Gain Device Access ............................ 437
Device Communication ......................................................... 438
USB Endpoints .................................................................... 438
The Default Pipe .................................................................. 439
Pipe States ........................................................................... 439
Opening Pipes ...................................................................... 440
Closing Pipes ........................................................................ 440
Data Transfer ....................................................................... 440
Flushing Pipes ...................................................................... 447
Device State Management .................................................... 448
Hotplugging USB Devices .................................................... 449
Power Management ............................................................. 451
Serialization ......................................................................... 455
Utility Functions .................................................................... 456
Device Configuration Facilities ............................................. 456
Other Utility Functions .......................................................... 458
Sample USB Device Drivers .................................................. 459

21 SR-IOV Drivers ................................................................. 461
Introduction to SR-IOV .......................................................... 461
Benefits of SR-IOV ............................................................... 463
Supported Platforms ............................................................. 463
Glossary .............................................................................. 464
Overview of SR-IOV Device Driver ....................................... 464
Part III  Building a Device Driver

22  Compiling, Loading, Packaging, and Testing Drivers

Driver Development Summary ................................................................. 487
Driver Code Layout .................................................................................. 488
Header Files ............................................................................................. 488
Source Files ............................................................................................... 489
Configuration Files .................................................................................... 489
Preparing for Driver Installation ................................................................. 490
Compiling and Linking the Driver .............................................................. 491
Module Dependencies .............................................................................. 492
Writing a Hardware Configuration File .................................................... 492
## Contents

1. Installing, Updating, and Removing Drivers ................................................................. 492
   - Copying the Driver to a Module Directory ............................................................... 493
   - Installing Drivers with `add_drv` ........................................................................... 494
   - Updating Driver Information ................................................................................. 494
   - Removing the Driver ............................................................................................. 495

2. Loading and Unloading Drivers .................................................................................. 495

3. Driver Packaging ...................................................................................................... 495

4. Criteria for Testing Drivers ....................................................................................... 496
   - Configuration Testing ............................................................................................. 496
   - Functionality Testing .............................................................................................. 496
   - Error Handling ........................................................................................................ 497

5. Testing Loading and Unloading .................................................................................. 497

6. Stress, Performance, and Interoperability Testing ..................................................... 497

7. DDI/DKI Compliance Testing .................................................................................... 498

8. Installation and Packaging Testing ............................................................................ 498

9. Testing Specific Types of Drivers .............................................................................. 498

10. Testing Drivers ......................................................................................................... 503

11. Enable the Deadman Feature to Avoid a Hard Hang ............................................... 503

12. Testing With a Serial Connection ............................................................................. 504

13. Setting Up Test Modules .......................................................................................... 506

14. Avoiding Data Loss on a Test System ...................................................................... 509

15. Recovering the Device Directory ............................................................................. 510

16. Debugging Tools ...................................................................................................... 511

17. Postmortem Debugging ............................................................................................ 512

18. Using the `kmdb` Kernel Debugger .......................................................................... 512

19. Using the `mdb` Modular Debugger ......................................................................... 515

20. Useful Debugging Tasks With `kmdb` and `mdb` ..................................................... 516

21. Tuning Drivers .......................................................................................................... 524

22. Kernel Statistics ....................................................................................................... 524

23. DTrace for Dynamic Instrumentation ...................................................................... 530

24. Recommended Coding Practices .............................................................................. 531

   - Debugging Preparation Techniques ..................................................................... 531
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use a Unique Prefix to Avoid Kernel Symbol Collisions</td>
<td>531</td>
</tr>
<tr>
<td>Use *cmn_err() to Log Driver Activity</td>
<td>532</td>
</tr>
<tr>
<td>Use *ASSERT() to Catch Invalid Assumptions</td>
<td>532</td>
</tr>
<tr>
<td>Use *mutex_owned() to Validate and Document Locking Requirements</td>
<td>533</td>
</tr>
<tr>
<td>Use Conditional Compilation to Toggle Costly Debugging Features</td>
<td>533</td>
</tr>
<tr>
<td>Declaring a Variable Volatile</td>
<td>534</td>
</tr>
<tr>
<td>Serviceability</td>
<td>536</td>
</tr>
<tr>
<td>Periodic Health Checks</td>
<td>536</td>
</tr>
</tbody>
</table>

**Part IV Appendixes**  

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Hardware Overview</td>
<td>539</td>
</tr>
<tr>
<td>SPARC Processor Issues</td>
<td>539</td>
</tr>
<tr>
<td>SPARC Data Alignment</td>
<td>540</td>
</tr>
<tr>
<td>Member Alignment in SPARC Structures</td>
<td>540</td>
</tr>
<tr>
<td>SPARC Byte Ordering</td>
<td>540</td>
</tr>
<tr>
<td>SPARC Register Windows</td>
<td>541</td>
</tr>
<tr>
<td>SPARC Multiply and Divide Instructions</td>
<td>541</td>
</tr>
<tr>
<td>x86 Processor Issues</td>
<td>541</td>
</tr>
<tr>
<td>x86 Byte Ordering</td>
<td>541</td>
</tr>
<tr>
<td>x86 Architecture Manuals</td>
<td>542</td>
</tr>
<tr>
<td>Endianness</td>
<td>542</td>
</tr>
<tr>
<td>Store Buffers</td>
<td>543</td>
</tr>
<tr>
<td>System Memory Model</td>
<td>543</td>
</tr>
<tr>
<td>Total Store Ordering (TSO)</td>
<td>544</td>
</tr>
<tr>
<td>Partial Store Ordering (PSO)</td>
<td>544</td>
</tr>
<tr>
<td>Bus Architectures</td>
<td>544</td>
</tr>
<tr>
<td>Device Identification</td>
<td>544</td>
</tr>
<tr>
<td>Supported Interrupt Types</td>
<td>545</td>
</tr>
<tr>
<td>Bus Specifics</td>
<td>545</td>
</tr>
<tr>
<td>PCI Local Bus</td>
<td>545</td>
</tr>
<tr>
<td>PCI Address Domain</td>
<td>546</td>
</tr>
<tr>
<td>PCI Express</td>
<td>548</td>
</tr>
<tr>
<td>SBus</td>
<td>549</td>
</tr>
<tr>
<td>Device Issues</td>
<td>550</td>
</tr>
</tbody>
</table>
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timing-Critical Sections</td>
<td>551</td>
</tr>
<tr>
<td>Delays</td>
<td>551</td>
</tr>
<tr>
<td>Internal Sequencing Logic</td>
<td>551</td>
</tr>
<tr>
<td>Interrupt Issues</td>
<td>551</td>
</tr>
<tr>
<td>PROM on SPARC Machines</td>
<td>552</td>
</tr>
<tr>
<td>Open Boot PROM 3</td>
<td>552</td>
</tr>
<tr>
<td>Reading and Writing</td>
<td>555</td>
</tr>
</tbody>
</table>

## B Summary of Oracle Solaris DDI/DKI Services

- Module Functions ........................................................................................................ 558
- Device Information Tree Node (dev_info_t) Functions ........................................... 558
- Device (dev_t) Functions ............................................................................................ 558
- Property Functions ....................................................................................................... 559
- Device Software State Functions .................................................................................. 560
- Memory Allocation and Deallocation Functions .......................................................... 560
- Kernel Thread Control and Synchronization Functions ............................................... 561
- Task Queue Management Functions ................................................................................ 562
- Interrupt Functions ....................................................................................................... 563
- Programmed I/O Functions ............................................................................................ 565
- Direct Memory Access (DMA) Functions ......................................................................... 571
- User Space Access Functions ......................................................................................... 573
- User Process Event Functions ......................................................................................... 574
- User Process Information Functions .............................................................................. 574
- User Application Kernel and Device Access Functions ................................................ 575
- Time-Related Functions ................................................................................................. 576
- Power Management Functions ......................................................................................... 577
- Fault Management Functions .......................................................................................... 577
- Kernel Statistics Functions ........................................................................................... 578
- Kernel Logging and Printing Functions ......................................................................... 579
- Buffered I/O Functions .................................................................................................. 579
- Virtual Memory Functions .............................................................................................. 580
- Device ID Functions ....................................................................................................... 580
- SCSI Functions ............................................................................................................... 581
- Resource Map Management Functions .............................................................................. 583
- System Global State ........................................................................................................ 583
Utility Functions .......................................................................................................................... 583

C Making a Device Driver 64-Bit Ready .......................................................................................... 585
Introduction to 64-Bit Driver Design ......................................................................................... 585
General Conversion Steps ........................................................................................................... 586
  Use Fixed-Width Types for Hardware Registers .................................................................. 587
  Use Fixed-Width Common Access Functions ...................................................................... 587
  Check and Extend Use of Derived Types ............................................................................. 588
  Check Changed Fields in DDI Data Structures .................................................................. 588
  Check Changed Arguments of DDI Functions ..................................................................... 589
  Modify Routines That Handle Data Sharing ...................................................................... 591
  Check Structures with 64-bit Long Data Types on x86-Based Platforms ......................... 592
Well Known ioctl Interfaces ...................................................................................................... 593
Device Sizes ............................................................................................................................... 594

D Console Frame Buffer Drivers .................................................................................................. 595
Oracle Solaris Consoles and the Kernel Terminal Emulator ....................................................... 595
  x86 Platform Console Communication ................................................................................. 595
  SPARC Platform Console Communication .......................................................................... 596
Console Visual I/O Interfaces ..................................................................................................... 597
  I/O Control Interfaces .............................................................................................................. 597
  Polled I/O Interfaces ............................................................................................................... 598
  Video Mode Change Callback Interface .............................................................................. 598
Implementing the Visual I/O Interfaces in Console Frame Buffer Drivers .............................. 599
  VIS_DEVINIT ......................................................................................................................... 599
  VIS_DEFINI ............................................................................................................................ 601
  VIS_CONSDISPLAY ............................................................................................................... 602
  VIS_CONSCOPY ...................................................................................................................... 603
  VIS_CONSCURSOR ................................................................................................................... 603
  VIS_PUTCMAP ........................................................................................................................ 604
  VIS_GETCMAP ........................................................................................................................ 605
Implementing Polled I/O in Console Frame Buffer Drivers ......................................................... 605
Frame Buffer Specific Configuration Module ................................................................................ 606
  The X Window System Frame Buffer Specific DDX Module .............................................. 606
Developing, Testing, and Debugging Console Frame Buffer Drivers ....................................... 606
Figures

FIGURE 2–1 Oracle Solaris Kernel ..........................................................................................................................56
FIGURE 2–2 Example Device Tree ..........................................................................................................................60
FIGURE 2–3 Device Node Names .........................................................................................................................63
FIGURE 2–4 Specific Driver Node Binding ............................................................................................................64
FIGURE 2–5 Generic Driver Node Binding ............................................................................................................65
FIGURE 5–1 Event Plumbing ....................................................................................................................................84
FIGURE 6–1 Module Loading and Autoconfiguration Entry Points ........................................................................94
FIGURE 9–1 CPU and System I/O Caches ............................................................................................................171
FIGURE 11–1 Device Context Management .........................................................................................................188
FIGURE 11–2 Device Context Switched to User Process A ..................................................................................189
FIGURE 12–1 Power Management Conceptual State Diagram ...............................................................................217
FIGURE 15–1 Character Driver Roadmap ............................................................................................................262
FIGURE 16–1 Block Driver Roadmap ...................................................................................................................288
FIGURE 17–1 SCSA Block Diagram ....................................................................................................................309
FIGURE 18–1 SCSA Interface ...............................................................................................................................333
FIGURE 18–2 Transport Layer Flow ....................................................................................................................334
FIGURE 18–3 HBA Transport Structures .............................................................................................................341
FIGURE 18–4 Cloning Transport Operation ..........................................................................................................343
FIGURE 18–5 scsi_pkt(9S) Structure Pointers ....................................................................................................354
FIGURE 20–1 Oracle Solaris USB Architecture .................................................................................................428
FIGURE 20–2 Driver and Controller Interfaces ..................................................................................................429
FIGURE 20–3 A Hierarchical USB Descriptor Tree ............................................................................................436
FIGURE 20–4 USB Device State Machine ...........................................................................................................448
FIGURE 20–5 USB Power Management .............................................................................................................453
FIGURE 21–1 SR-IOV Technology .......................................................................................................................462
FIGURE 21–2 High-Level View of Sparc OVM Configuration .............................................................................469
FIGURE A–1 Byte Ordering Required for Host Bus Dependency ........................................................................542
FIGURE A–2 Data Ordering Host Bus Dependency ............................................................................................543
Figures

FIGURE A–3 Machine Block Diagram ............................................................. 546
FIGURE A–4 Base Address Registers for Memory and I/O ...................... 547
Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE 1-1</td>
<td>Entry Points for All Driver Types</td>
<td>42</td>
</tr>
<tr>
<td>TABLE 1-2</td>
<td>Additional Entry Points for Block Drivers</td>
<td>44</td>
</tr>
<tr>
<td>TABLE 1-3</td>
<td>Additional Entry Points for Character Drivers</td>
<td>45</td>
</tr>
<tr>
<td>TABLE 1-4</td>
<td>Entry Points for STREAMS Drivers</td>
<td>47</td>
</tr>
<tr>
<td>TABLE 1-5</td>
<td>Entry Points for Character Drivers That Use devmap for Memory Mapping</td>
<td>47</td>
</tr>
<tr>
<td>TABLE 1-6</td>
<td>Additional Entry Points for SCSI HBA Drivers</td>
<td>48</td>
</tr>
<tr>
<td>TABLE 1-7</td>
<td>Entry Points for PC Card Drivers Only</td>
<td>49</td>
</tr>
<tr>
<td>TABLE 4-1</td>
<td>Property Interface Uses</td>
<td>79</td>
</tr>
<tr>
<td>TABLE 5-1</td>
<td>Functions for Using Name-Value Pairs</td>
<td>88</td>
</tr>
<tr>
<td>TABLE 6-1</td>
<td>Possible Node Types</td>
<td>106</td>
</tr>
<tr>
<td>TABLE 8-1</td>
<td>Callback Support Interfaces</td>
<td>135</td>
</tr>
<tr>
<td>TABLE 8-2</td>
<td>Interrupt Vector Request Interfaces</td>
<td>137</td>
</tr>
<tr>
<td>TABLE 9-1</td>
<td>Resource Allocation Handling</td>
<td>167</td>
</tr>
<tr>
<td>TABLE 12-1</td>
<td>Power Management Interfaces</td>
<td>218</td>
</tr>
<tr>
<td>TABLE 17-1</td>
<td>Standard SCSA Functions</td>
<td>310</td>
</tr>
<tr>
<td>TABLE 18-1</td>
<td>SCSA HBA Entry Point Summary</td>
<td>334</td>
</tr>
<tr>
<td>TABLE 18-2</td>
<td>SCSA HBA Functions</td>
<td>343</td>
</tr>
<tr>
<td>TABLE 18-3</td>
<td>SCSA Entry Points</td>
<td>351</td>
</tr>
<tr>
<td>TABLE 19-1</td>
<td>GLDv3 Interfaces</td>
<td>400</td>
</tr>
<tr>
<td>TABLE 20-1</td>
<td>Request Initialization</td>
<td>442</td>
</tr>
<tr>
<td>TABLE 20-2</td>
<td>Request Transfer Setup</td>
<td>443</td>
</tr>
<tr>
<td>TABLE 21-1</td>
<td>Configuration Parameters Definition</td>
<td>466</td>
</tr>
<tr>
<td>TABLE 21-2</td>
<td>Interfaces for SR-IOV Drivers</td>
<td>470</td>
</tr>
<tr>
<td>TABLE 22-1</td>
<td>Compiler Options for SPARC and x86 64-bit Architectures</td>
<td>491</td>
</tr>
<tr>
<td>TABLE 23-1</td>
<td>kmdb Macros</td>
<td>514</td>
</tr>
<tr>
<td>TABLE 23-2</td>
<td>Ethernet MII/GMII Physical Layer Interface Kernel Statistics</td>
<td>527</td>
</tr>
<tr>
<td>TABLE A-1</td>
<td>Device Physical Space in the Ultra 2</td>
<td>549</td>
</tr>
<tr>
<td>TABLE A-2</td>
<td>Ultra 2 SBus Address Bits</td>
<td>550</td>
</tr>
<tr>
<td>TABLE B-1</td>
<td>Deprecated Property Functions ................................................................. 559</td>
<td></td>
</tr>
<tr>
<td>TABLE B-2</td>
<td>Deprecated Memory Allocation and Deallocation Functions ....................... 561</td>
<td></td>
</tr>
<tr>
<td>TABLE B-3</td>
<td>Deprecated Interrupt Functions ................................................................. 564</td>
<td></td>
</tr>
<tr>
<td>TABLE B-4</td>
<td>Deprecated Programmed I/O Functions ....................................................... 568</td>
<td></td>
</tr>
<tr>
<td>TABLE B-5</td>
<td>Deprecated Direct Memory Access (DMA) Functions .................................... 572</td>
<td></td>
</tr>
<tr>
<td>TABLE B-6</td>
<td>Deprecated User Space Access Functions ................................................... 574</td>
<td></td>
</tr>
<tr>
<td>TABLE B-7</td>
<td>Deprecated User Process Information Functions ......................................... 575</td>
<td></td>
</tr>
<tr>
<td>TABLE B-8</td>
<td>Deprecated User Application Kernel and Device Access Functions ............. 575</td>
<td></td>
</tr>
<tr>
<td>TABLE B-9</td>
<td>Deprecated Time-Related Functions ........................................................... 576</td>
<td></td>
</tr>
<tr>
<td>TABLE B-10</td>
<td>Deprecated Power Management Functions ................................................ 577</td>
<td></td>
</tr>
<tr>
<td>TABLE B-11</td>
<td>Deprecated Virtual Memory Functions ...................................................... 580</td>
<td></td>
</tr>
<tr>
<td>TABLE B-12</td>
<td>Deprecated SCSI Functions ....................................................................... 582</td>
<td></td>
</tr>
<tr>
<td>TABLE C-1</td>
<td>Comparison of ILP32 and LP64 Data Types .................................................. 585</td>
<td></td>
</tr>
</tbody>
</table>
Examples

EXAMPLE 3–1   Using Mutexes and Condition Variables ................................................................. 71
EXAMPLE 3–2   Using cv_timedwait() ............................................................................................. 72
EXAMPLE 3–3   Using cv_waitSig() .............................................................................................. 73
EXAMPLE 4–1   Driver Check for Locally Configured Timeout Value ........................................ 81
EXAMPLE 4–2   prop_op() Routine ................................................................................................. 82
EXAMPLE 5–1   Calling ddi_log_sysEvent() .................................................................................... 86
EXAMPLE 5–2   Creating and Populating a Name-Value Pair List .............................................. 87
EXAMPLE 6–1   Loadable Interface Section ..................................................................................... 97
EXAMPLE 6–2   _init() Function ................................................................................................... 98
EXAMPLE 6–3   probe(9E) Routine ................................................................................................ 102
EXAMPLE 6–4   probe(9E) Routine Using ddi_poke8(9F) .............................................................. 103
EXAMPLE 6–5   Typical attach() Entry Point .............................................................................. 107
EXAMPLE 6–6   Typical detach() Entry Point .............................................................................. 110
EXAMPLE 6–7   Typical getinfo() Entry Point ............................................................................ 111
EXAMPLE 7–1   Mapping Setup .................................................................................................. 118
EXAMPLE 7–2   Mapping Setup: Buffer ........................................................................................ 119
EXAMPLE 8–1   Changing Soft Interrupt Priority ......................................................................... 127
EXAMPLE 8–2   Checking for Pending Interrupts ........................................................................ 127
EXAMPLE 8–3   Setting Interrupt Masks ...................................................................................... 127
EXAMPLE 8–4   Clearing Interrupt Masks .................................................................................... 128
EXAMPLE 8–5   Registering a Legacy Interrupt .......................................................................... 129
EXAMPLE 8–6   Removing a Legacy Interrupt ............................................................................ 130
EXAMPLE 8–7   Registering a Set of MSI Interrupts .................................................................... 131
EXAMPLE 8–8   Removing MSI Interrupts .................................................................................. 133
EXAMPLE 8–9   Interrupt Example .............................................................................................. 146
EXAMPLE 8–10  Handling High-Level Interrupts With attach() .............................................. 148
EXAMPLE 8–11  High-level Interrupt Routine ............................................................................. 149
EXAMPLE 8–12  Low-Level Soft Interrupt Routine .................................................................... 150
<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXAMPLE 9–1</td>
<td>DMA Callback Example</td>
<td>164</td>
</tr>
<tr>
<td>EXAMPLE 9–2</td>
<td>Determining Burst Size</td>
<td>165</td>
</tr>
<tr>
<td>EXAMPLE 9–3</td>
<td>Using ddi_dma_mem_alloc(9F)</td>
<td>166</td>
</tr>
<tr>
<td>EXAMPLE 9–4</td>
<td>ddi_dma_cookie(9S) Example</td>
<td>168</td>
</tr>
<tr>
<td>EXAMPLE 9–5</td>
<td>Freeing DMA Resources</td>
<td>168</td>
</tr>
<tr>
<td>EXAMPLE 9–6</td>
<td>Canceling DMA Callbacks</td>
<td>169</td>
</tr>
<tr>
<td>EXAMPLE 9–7</td>
<td>Setting Up DMA Windows</td>
<td>173</td>
</tr>
<tr>
<td>EXAMPLE 9–8</td>
<td>Interrupt Handler Using DMA Windows</td>
<td>174</td>
</tr>
<tr>
<td>EXAMPLE 10–1</td>
<td>segmap(9E) Routine</td>
<td>178</td>
</tr>
<tr>
<td>EXAMPLE 10–2</td>
<td>Using the segmap() Function to Change the Address Returned by the mmap() Call</td>
<td>179</td>
</tr>
<tr>
<td>EXAMPLE 10–3</td>
<td>Using the devmap_devmem_setup() Routine</td>
<td>182</td>
</tr>
<tr>
<td>EXAMPLE 10–4</td>
<td>Using the ddi_umem_alloc() Routine</td>
<td>184</td>
</tr>
<tr>
<td>EXAMPLE 10–5</td>
<td>devmap_umem_setup(9F) Routine</td>
<td>186</td>
</tr>
<tr>
<td>EXAMPLE 11–1</td>
<td>Using the devmap() Routine</td>
<td>191</td>
</tr>
<tr>
<td>EXAMPLE 11–2</td>
<td>Using the devmap_access() Routine</td>
<td>192</td>
</tr>
<tr>
<td>EXAMPLE 11–3</td>
<td>Using the devmap_contextmgmt() Routine</td>
<td>193</td>
</tr>
<tr>
<td>EXAMPLE 11–4</td>
<td>Using the devmap_dup() Routine</td>
<td>195</td>
</tr>
<tr>
<td>EXAMPLE 11–5</td>
<td>Using the devmap_unmap() Routine</td>
<td>196</td>
</tr>
<tr>
<td>EXAMPLE 11–6</td>
<td>devmap(9E) Entry With Context Management Support</td>
<td>198</td>
</tr>
<tr>
<td>EXAMPLE 12–1</td>
<td>Sample pm-component Entry</td>
<td>204</td>
</tr>
<tr>
<td>EXAMPLE 12–2</td>
<td>attach(9E) Routine With pm-components Property</td>
<td>204</td>
</tr>
<tr>
<td>EXAMPLE 12–3</td>
<td>Multiple Component pm-components Entry</td>
<td>205</td>
</tr>
<tr>
<td>EXAMPLE 12–4</td>
<td>Using the power() Routine for a Single-Component Device</td>
<td>208</td>
</tr>
<tr>
<td>EXAMPLE 12–5</td>
<td>power(9E) Routine for Multiple-Component Device</td>
<td>209</td>
</tr>
<tr>
<td>EXAMPLE 12–6</td>
<td>detach(9E) Routine Implementing DDI_SUSPEND</td>
<td>213</td>
</tr>
<tr>
<td>EXAMPLE 12–7</td>
<td>attach(9E) Routine Implementing DDI_RESUME</td>
<td>214</td>
</tr>
<tr>
<td>EXAMPLE 12–8</td>
<td>Device Access</td>
<td>215</td>
</tr>
<tr>
<td>EXAMPLE 12–9</td>
<td>Device Operation Completion</td>
<td>216</td>
</tr>
<tr>
<td>EXAMPLE 14–1</td>
<td>Configuration File</td>
<td>241</td>
</tr>
<tr>
<td>EXAMPLE 14–2</td>
<td>Driver Source File</td>
<td>242</td>
</tr>
<tr>
<td>EXAMPLE 14–3</td>
<td>Write a Short Message to the Layered Device</td>
<td>250</td>
</tr>
<tr>
<td>EXAMPLE 14–4</td>
<td>Write a Longer Message to the Layered Device</td>
<td>250</td>
</tr>
<tr>
<td>EXAMPLE 14–5</td>
<td>Change the Target Device</td>
<td>251</td>
</tr>
<tr>
<td>EXAMPLE 14–6</td>
<td>Device Usage Information</td>
<td>254</td>
</tr>
<tr>
<td>EXAMPLE 14–7</td>
<td>Ancestor Node Usage Information</td>
<td>254</td>
</tr>
</tbody>
</table>
EXAMPLE 14–8  Child Node Usage Information ..................................................254
EXAMPLE 14–9  Layering and Device Minor Node Information – Keyboard ........254
EXAMPLE 14–10 Layering and Device Minor Node Information – Network Device ....255
EXAMPLE 14–11 Consumers of Underlying Device Nodes ..............................256
EXAMPLE 14–12 Consumer of the Keyboard Device .......................................257
EXAMPLE 15–1  Character Driver attach() Routine ........................................263
EXAMPLE 15–2  Character Driver open(9E) Routine .......................................265
EXAMPLE 15–3  Ramdisk read(9E) Routine Using uiomove(9F) ....................270
EXAMPLE 15–4  Programmed I/O write(9E) Routine Using uwritec(9F) ...........270
EXAMPLE 15–5  read(9E) and write(9E) Routines Using physio(9F) .................271
EXAMPLE 15–6  a read(9E) and a write(9E) Routines Using a physio(9F) ...........272
EXAMPLE 15–7  minphys(9F) Routine .................................................................273
EXAMPLE 15–8  strategy(9E) Routine .................................................................274
EXAMPLE 15–9  Interrupt Routine .................................................................275
EXAMPLE 15–10 chpoll(9E) Routine .................................................................277
EXAMPLE 15–11 Interrupt Routine Supporting chpoll(9E) ..............................278
EXAMPLE 15–12 ioctl(9E) Routine .................................................................279
EXAMPLE 15–13 Using ioctl(9E) .................................................................280
EXAMPLE 15–14 ioctl(9E) Routine to Support 32-bit Applications and 64-bit Applications ...281
EXAMPLE 15–15 Handling copyout(9F) Overflow ........................................282
EXAMPLE 15–16 Using Data Structure Macros to Move Data .......................283
EXAMPLE 16–1  Block Driver attach() Routine .............................................289
EXAMPLE 16–2  Block Driver open(9E) Routine .............................................291
EXAMPLE 16–3  Block Device close(9E) Routine ............................................292
EXAMPLE 16–4  Synchronous Interrupt Routine for Block Drivers ..................297
EXAMPLE 16–5  Enqueuing Data Transfer Requests for Block Drivers ..............300
EXAMPLE 16–6  Starting the First Data Request for a Block Driver ..................301
EXAMPLE 16–7  Block Driver Routine for Asynchronous Interrupts ...............302
EXAMPLE 17–1  SCSI Target Driver probe(9E) Routine .................................315
EXAMPLE 17–2  SCSI Target Driver attach(9E) Routine ..................................317
EXAMPLE 17–3  SCSI Target Driver detach(9E) Routine .................................319
EXAMPLE 17–4  Alternative SCSI Target Driver getinfo() Code Fragment .........320
EXAMPLE 17–5  Completion Routine for a SCSI Driver .................................325
EXAMPLE 17–6  Enabling Auto-Request Sense Mode ......................................327
EXAMPLE 17–7  dump(9E) Routine .................................................................328
EXAMPLE 18–1  Module Initialization for SCSI HBA ......................................346
<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example 18–2</td>
<td>HBA Driver Initialization of a SCSI Packet Structure</td>
<td>354</td>
</tr>
<tr>
<td>Example 18–3</td>
<td>HBA Driver Allocation of DMA Resources</td>
<td>357</td>
</tr>
<tr>
<td>Example 18–4</td>
<td>DMA Resource Reallocation for HBA Drivers</td>
<td>359</td>
</tr>
<tr>
<td>Example 18–5</td>
<td>HBA Driver <code>tran_destroy_pkt(9E)</code> Entry Point</td>
<td>360</td>
</tr>
<tr>
<td>Example 18–6</td>
<td>HBA Driver <code>tran_sync_pkt(9E)</code> Entry Point</td>
<td>361</td>
</tr>
<tr>
<td>Example 18–7</td>
<td>HBA Driver <code>tran_dmafree(9E)</code> Entry Point</td>
<td>362</td>
</tr>
<tr>
<td>Example 18–8</td>
<td>HBA Driver <code>tran_start(9E)</code> Entry Point</td>
<td>363</td>
</tr>
<tr>
<td>Example 18–9</td>
<td>HBA Driver Interrupt Handler</td>
<td>365</td>
</tr>
<tr>
<td>Example 18–10</td>
<td>HBA Driver <code>tran_getcap(9E)</code> Entry Point</td>
<td>368</td>
</tr>
<tr>
<td>Example 18–11</td>
<td>HBA Driver <code>tran_setcap(9E)</code> Entry Point</td>
<td>371</td>
</tr>
<tr>
<td>Example 18–12</td>
<td>HBA Driver <code>tran_reset_notify(9E)</code> Entry Point</td>
<td>374</td>
</tr>
<tr>
<td>Example 19–1</td>
<td><code>mac_init_ops()</code> and <code>mac_fini_ops()</code> Functions</td>
<td>383</td>
</tr>
<tr>
<td>Example 19–2</td>
<td><code>mac_alloc()</code>, <code>mac_register()</code>, and <code>mac_free()</code> Functions and</td>
<td>383</td>
</tr>
<tr>
<td></td>
<td><code>mac_register</code> Structure</td>
<td></td>
</tr>
<tr>
<td>Example 19–3</td>
<td><code>mac_unregister()</code> Function</td>
<td>384</td>
</tr>
<tr>
<td>Example 19–4</td>
<td><code>mac_callbacks</code> Structure</td>
<td>385</td>
</tr>
<tr>
<td>Example 19–5</td>
<td><code>mc_getcapab()</code> Entry Point</td>
<td>386</td>
</tr>
<tr>
<td>Example 19–6</td>
<td><code>mc_tx()</code> and <code>mri_tx()</code> Entry Point</td>
<td>394</td>
</tr>
<tr>
<td>Example 19–7</td>
<td><code>mc_getstat()</code> Entry Point</td>
<td>398</td>
</tr>
<tr>
<td>Example 20–1</td>
<td>USB Mouse Compatible Device Names</td>
<td>431</td>
</tr>
<tr>
<td>Example 20–2</td>
<td>Compatible Device Names Shown by the Print Configuration Command</td>
<td>431</td>
</tr>
<tr>
<td>Example 20–3</td>
<td>USB Audio Compatible Device Names</td>
<td>432</td>
</tr>
<tr>
<td>Example 20–4</td>
<td>USB Video Interface Association Compatible Names</td>
<td>433</td>
</tr>
<tr>
<td>Example 21–1</td>
<td>Setting Device Configuration Parameters</td>
<td>468</td>
</tr>
<tr>
<td>Example 21–2</td>
<td>SR-IOV <code>pci_param_get(9F)</code> Routine</td>
<td>472</td>
</tr>
<tr>
<td>Example 23–1</td>
<td>Setting <code>input-device</code> and <code>output-device</code> With Boot PROM Commands</td>
<td>505</td>
</tr>
<tr>
<td>Example 23–2</td>
<td>Setting <code>input-device</code> and <code>output-device</code> With the <code>eeprom</code> Command</td>
<td>506</td>
</tr>
<tr>
<td>Example 23–3</td>
<td>Using <code>modinfo</code> to Confirm a Loaded Driver</td>
<td>507</td>
</tr>
<tr>
<td>Example 23–4</td>
<td>Recovering a Damaged Device Directory</td>
<td>510</td>
</tr>
<tr>
<td>Example 23–5</td>
<td>Setting Standard Breakpoints in <code>kmdb</code></td>
<td>513</td>
</tr>
<tr>
<td>Example 23–6</td>
<td>Setting Deferred Breakpoints in <code>kmdb</code></td>
<td>513</td>
</tr>
<tr>
<td>Example 23–7</td>
<td>Invoking <code>mdb</code> on a Crash Dump</td>
<td>516</td>
</tr>
<tr>
<td>Example 23–8</td>
<td>Invoking <code>mdb</code> on a Running Kernel</td>
<td>516</td>
</tr>
<tr>
<td>Example 23–9</td>
<td>Reading All Registers on a SPARC Processor With <code>kmdb</code></td>
<td>517</td>
</tr>
<tr>
<td>Example 23–10</td>
<td>Reading and Writing Registers on an x86 Machine With <code>kmdb</code></td>
<td>517</td>
</tr>
<tr>
<td>Example 23–11</td>
<td>Inspecting the Registers of a Different Processor</td>
<td>518</td>
</tr>
</tbody>
</table>
### Examples

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>23–12</td>
<td>Retrieving the Value of an Individual Register From a Specified Processor</td>
</tr>
<tr>
<td>23–13</td>
<td>Displaying Kernel Data Structures With a Debugger</td>
</tr>
<tr>
<td>23–14</td>
<td>Displaying the Size of a Kernel Data Structure</td>
</tr>
<tr>
<td>23–15</td>
<td>Displaying the Offset to a Kernel Data Structure</td>
</tr>
<tr>
<td>23–16</td>
<td>Displaying the Relative Addresses of a Kernel Data Structure</td>
</tr>
<tr>
<td>23–17</td>
<td>Displaying the Absolute Addresses of a Kernel Data Structure</td>
</tr>
<tr>
<td>23–18</td>
<td>Using the <code>::prtconf</code> Dcmd</td>
</tr>
<tr>
<td>23–19</td>
<td>Displaying Device Information for an Individual Node</td>
</tr>
<tr>
<td>23–20</td>
<td>Using the <code>::prtconf</code> Dcmd in Verbose Mode</td>
</tr>
<tr>
<td>23–21</td>
<td>Using the <code>::devbindings</code> Dcmd to Locate Driver Instances</td>
</tr>
<tr>
<td>23–22</td>
<td>Modifying a Kernel Variable With a Debugger</td>
</tr>
</tbody>
</table>
Writing Device Drivers provides information on developing drivers for character-oriented devices, block-oriented devices, network devices, SCSI target and HBA devices, and USB devices for the Oracle Solaris Operating System (Oracle Solaris OS). This book discusses how to develop multithreaded reentrant device drivers for all architectures that conform to the Oracle Solaris OS DDI/DKI (Device Driver Interface, Driver-Kernel Interface). A common driver programming approach is described that enables drivers to be written without concern for platform-specific issues such as endianness and data ordering.

Additional topics include hardening Oracle Solaris drivers; power management; driver autoconfiguration; programmed I/O; Direct Memory Access (DMA); device context management; compilation, installation, and testing drivers; debugging drivers; and porting Oracle Solaris drivers to a 64-bit environment.

Note – This Oracle Solaris release supports systems that use the SPARC and x86 families of processor architectures. The supported systems appear in the Oracle Solaris OS: Hardware Compatibility Lists. This document cites any implementation differences between the platform types.

In this document, these x86 related terms mean the following:
- x86 refers to the larger family of 64-bit and 32-bit x86 compatible products.
- x64 relates specifically to 64-bit x86 compatible CPUs.
- "32-bit x86" points out specific 32-bit information about x86 based systems.

For supported systems, see the Oracle Solaris OS: Hardware Compatibility Lists.

Who Should Use This Book

This book is written for UNIX programmers who are familiar with UNIX device drivers. Overview information is provided, but the book is not intended to serve as a general tutorial on device drivers.
Note – The Oracle Solaris operating system (Oracle Solaris OS) runs on both SPARC and x86 architectures. The Oracle Solaris OS also runs on both 64-bit and 32-bit address spaces. The information in this document applies to all platforms and address spaces unless specifically noted.

How This Book Is Organized

This book is organized into the following chapters:

- **Chapter 1, “Overview of Oracle Solaris Device Drivers,”** provides an introduction to device drivers and associated entry points on the Oracle Solaris platform. The entry points for each device driver type are presented in tables.
- **Chapter 2, “Oracle Solaris Kernel and Device Tree,”** provides an overview of the Oracle Solaris kernel with an explanation of how devices are represented as nodes in a device tree.
- **Chapter 3, “Multithreading,”** describes the aspects of the Oracle Solaris multithreaded kernel that are relevant for device driver developers.
- **Chapter 4, “Properties,”** describes the set of interfaces for using device properties.
- **Chapter 5, “Managing Events and Queueing Tasks,”** describes how device drivers log events and how to use task queues to perform a task at a later time.
- **Chapter 6, “Driver Autoconfiguration,”** explains the support that a driver must provide for autoconfiguration.
- **Chapter 7, “Device Access: Programmed I/O,”** describes the interfaces and methodologies for drivers to read or write to device memory.
- **Chapter 8, “Interrupt Handlers,”** describes the mechanisms for handling interrupts. These mechanisms include allocating, registering, servicing, and removing interrupts.
- **Chapter 9, “Direct Memory Access (DMA),”** describes direct memory access (DMA) and the DMA interfaces.
- **Chapter 10, “Mapping Device and Kernel Memory,”** describes interfaces for managing device and kernel memory.
- **Chapter 11, “Device Context Management,”** describes the set of interfaces that enable device drivers to manage user access to devices.
- **Chapter 12, “Power Management,”** explains the interfaces for Power Management, a framework for managing power consumption.
- **Chapter 13, “Hardening Oracle Solaris Drivers,”** describes how to integrate fault management capabilities into I/O device drivers, how to incorporate defensive programming practices, and how to use the driver hardening test harness.
- **Chapter 14, “Layered Driver Interface (LDI),”** describes the LDI, which enables kernel modules to access other devices in the system.

Chapter 16, “Drivers for Block Devices,” describes drivers for a block-oriented devices.

Chapter 17, “SCSI Target Drivers,” outlines the Sun Common SCSI Architecture (SCSA) and the requirements for SCSI target drivers.

Chapter 18, “SCSI Host Bus Adapter Drivers,” explains how to apply SCSA to SCSI Host Bus Adapter (HBA) drivers.

Chapter 19, “Drivers for Network Devices,” describes the Generic LAN driver (GLD). The GLDv3 framework is a function calls-based interface of MAC plugins and MAC driver service routines and structures.

Chapter 20, “USB Drivers,” describes how to write a client USB device driver using the USBA 2.0 framework.

Chapter 21, “SR-IOV Drivers,” describes the requirements to write a SR-IOV device driver.

Chapter 22, “Compiling, Loading, Packaging, and Testing Drivers,” provides information on compiling, linking, and installing a driver.

Chapter 23, “Debugging, Testing, and Tuning Device Drivers,” describes techniques for debugging, testing, and tuning drivers.

Chapter 24, “Recommended Coding Practices,” describes the recommended coding practices for writing drivers.

Appendix A, “Hardware Overview,” discusses multi-platform hardware issues for device drivers.

Appendix B, “Summary of Oracle Solaris DDI/DKI Services,” provides tables of kernel functions for device drivers. Deprecated functions are indicated as well.

Appendix C, “Making a Device Driver 64-Bit Ready,” provides guidelines for updating a device driver to run in a 64-bit environment.

Appendix D, “Console Frame Buffer Drivers,” describes how to add the necessary interfaces to a frame buffer driver to enable the driver to interact with the Oracle Solaris kernel terminal emulator.

Related Books and Papers

For detailed reference information about the device driver interfaces, see the section 9 man pages. Section 9E, Intro(9E), describes DDI/DKI (Device Driver Interface, Driver-Kernel Interface) driver entry points. Section 9F, Intro(9F), describes DDI/DKI kernel functions. Sections 9P and 9S, Intro(9S), describe DDI/DKI properties and data structures.

For information on hardware and other driver-related issues, see the following books:

- Device Driver Tutorial
- Oracle Solaris Modular Debugger Guide
Preface

- Oracle Solaris 11.1 Dynamic Tracing Guide
- Multithreaded Programming Guide
- STREAMS Programming Guide
The following book might also be useful:


## Access to Oracle Support


## Typographic Conventions

The following table describes the typographic conventions that are used in this book.

<table>
<thead>
<tr>
<th>Typeface</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
</table>
| *AaBbCc123* | The names of commands, files, and directories, and onscreen computer output | Edit your `.login` file.  
Use `ls -a` to list all files.  
`machine_name% you have mail.` |
| *AaBbCc123* | What you type, contrasted with onscreen computer output | `machine_name% su`  
`Password:` |
| *aabbcc123* | Placeholder: replace with a real name or value | The command to remove a file is `rm filename`. |
| *AaBbCc123* | Book titles, new terms, and terms to be emphasized | Read Chapter 6 in the *User’s Guide*.  
A *cache* is a copy that is stored locally.  
Do *not* save the file.  
**Note:** Some emphasized items appear bold online. |
Shell Prompts in Command Examples

The following table shows UNIX system prompts and superuser prompts for shells that are included in the Oracle Solaris OS. In command examples, the shell prompt indicates whether the command should be executed by a regular user or a user with privileges.

<table>
<thead>
<tr>
<th>Shell</th>
<th>Prompt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bash shell, Korn shell, and Bourne shell</td>
<td>$</td>
</tr>
<tr>
<td>Bash shell, Korn shell, and Bourne shell for superuser</td>
<td>#</td>
</tr>
<tr>
<td>C shell</td>
<td>machine_name%</td>
</tr>
<tr>
<td>C shell for superuser</td>
<td>machine_name#</td>
</tr>
</tbody>
</table>
Designing Device Drivers for the Oracle Solaris Platform

The first part of this manual provides general information for developing device drivers on the Oracle Solaris platform. This part includes the following chapters:

- **Chapter 1, “Overview of Oracle Solaris Device Drivers,”** provides an introduction to device drivers and associated entry points on the Oracle Solaris platform. The entry points for each device driver type are presented in tables.
- **Chapter 2, “Oracle Solaris Kernel and Device Tree,”** provides an overview of the Oracle Solaris kernel with an explanation of how devices are represented as nodes in a device tree.
- **Chapter 3, “Multithreading,”** describes the aspects of the Oracle Solaris multithreaded kernel that are relevant for device driver developers.
- **Chapter 4, “Properties,”** describes the set of interfaces for using device properties.
- **Chapter 5, “Managing Events and Queueing Tasks,”** describes how device drivers log events and how to use task queues to perform a task at a later time.
- **Chapter 6, “Driver Autoconfiguration,”** explains the support that a driver must provide for autoconfiguration.
- **Chapter 7, “Device Access: Programmed I/O,”** describes the interfaces and methodologies for drivers to read or write to device memory.
Chapter 8, “Interrupt Handlers,” describes the mechanisms for handling interrupts. These mechanisms include allocating, registering, servicing, and removing interrupts.

Chapter 9, “Direct Memory Access (DMA),” describes direct memory access (DMA) and the DMA interfaces.


Chapter 11, “Device Context Management,” describes the set of interfaces that enable device drivers to manage user access to devices.

Chapter 12, “Power Management,” explains the interfaces for the Power Management feature, a framework for managing power consumption.

Chapter 13, “Hardening Oracle Solaris Drivers,” describes how to integrate fault management capabilities into I/O device drivers, how to incorporate defensive programming practices, and how to use the driver hardening test harness.

Chapter 14, “Layered Driver Interface (LDI),” describes the LDI, which enables kernel modules to access other devices in the system.
Overview of Oracle Solaris Device Drivers

This chapter gives an overview of Oracle Solaris device drivers. The chapter provides information on the following subjects:
- “Device Driver Basics” on page 39
- “Device Driver Entry Points” on page 41
- “Considerations in Device Driver Design” on page 50

Device Driver Basics

This section introduces you to device drivers and their entry points on the Oracle Solaris platform.

What Is a Device Driver?

A device driver is a kernel module that is responsible for managing the low-level I/O operations of a hardware device. Device drivers are written with standard interfaces that the kernel can call to interface with a device. Device drivers can also be software-only, emulating a device that exists only in software, such as RAM disks, buses, and pseudo-terminals.

A device driver contains all the device-specific code necessary to communicate with a device. This code includes a standard set of interfaces to the rest of the system. This interface shields the kernel from device specifics just as the system call interface protects application programs from platform specifics. Application programs and the rest of the kernel need little, if any, device-specific code to address the device. In this way, device drivers make the system more portable and easier to maintain.

When the Oracle Solaris operating system (Oracle Solaris OS) is initialized, devices identify themselves and are organized into the device tree, a hierarchy of devices. In effect, the device tree is a hardware model for the kernel. An individual device driver is represented as a node in the tree with no children. This type of node is referred to as a leaf driver. A driver that provides
services to other drivers is called a bus nexus driver and is shown as a node with children. As part of the boot process, physical devices are mapped to drivers in the tree so that the drivers can be located when needed. For more information on how the Oracle Solaris OS accommodates devices, see Chapter 2, “Oracle Solaris Kernel and Device Tree.”

Device drivers are classified by how they handle I/O. Device drivers fall into three broad categories:

- **Block device drivers** – For cases where handling I/O data as asynchronous chunks is appropriate. Typically, block drivers are used to manage devices with physically addressable storage media, such as disks.

- **Character device drivers** – For devices that perform I/O on a continuous flow of bytes.

**Note** – A driver can be both block and character at the same time if you set up two different interfaces to the file system. See “Devices as Special Files” on page 57.

Included in the character category are drivers that use the STREAMS model (see below), programmed I/O, direct memory access, SCSI buses, USB, and other network I/O.

- **STREAMS device drivers** – Subset of character drivers that uses the streamio(7I) set of routines for character I/O within the kernel.

### What Is a Device Driver Entry Point?

An *entry point* is a function within a device driver that can be called by an external entity to get access to some driver functionality or to operate a device. Each device driver provides a standard set of functions as entry points. For the complete list of entry points for all driver types, see the Int ro(9E) man page. The Oracle Solaris kernel uses entry points for these general task areas:

- **Loading and unloading the driver**

- **Autoconfiguring the device** – Autoconfiguration is the process of loading a device driver’s code and static data into memory so that the driver is registered with the system.

- **Providing I/O services for the driver**

Drivers for different types of devices have different sets of entry points according to the kinds of operations the devices perform. A driver for a memory-mapped character-oriented device, for example, supports a devmap(9E) entry point, while a block driver does not support this entry.

Use a prefix based on the name of your driver to give driver functions unique names. Typically, this prefix is the name of the driver, such as xx_open() for the open(9E) routine of driver xx. See “Use a Unique Prefix to Avoid Kernel Symbol Collisions” on page 531 for more information. In subsequent examples in this book, xx is used as the driver prefix.
Device Driver Entry Points

This section provides lists of entry points for the following categories:

- "Entry Points Common to All Drivers" on page 41
- "Entry Points for Block Device Drivers" on page 44
- "Entry Points for Character Device Drivers" on page 45
- "Entry Points for STREAMS Device Drivers" on page 46
- "Entry Points for Memory Mapped Devices" on page 47
- "Entry Points for Network Device Drivers" on page 48
- "Entry Points for SCSI HBA Drivers" on page 48
- "Entry Points for PC Card Drivers" on page 49

Entry Points Common to All Drivers

Some operations can be performed by any type of driver, such as the functions that are required for module loading and for the required autoconfiguration entry points. This section discusses types of entry points that are common to all drivers. The common entry points are listed in "Summary of Common Entry Points" on page 42 with links to man pages and other relevant discussions.

Device Access Entry Points

Drivers for character and block devices export the `cb_ops(9S)` structure, which defines the driver entry points for block device access and character device access. Both types of drivers are required to support the `open(9E)` and `close(9E)` entry points. Block drivers are required to support `strategy(9E)`, while character drivers can choose to implement whatever mix of `read(9E), write(9E), ioctl(9E), mmap(9E),` or `devmap(9E)` entry points is appropriate for the type of device. Character drivers can also support a polling interface through `chpoll(9E)`. Asynchronous I/O is supported through `aread(9E)` and `awrite(9E)` for block drivers and those drivers that can use both block and character file systems.

Loadable Module Entry Points

All drivers are required to implement the loadable module entry points `init(9E), fini(9E), and info(9E)` to load, unload, and report information about the driver module.

Drivers should allocate and initialize any global resources in `init(9E)`. Drivers should release their resources in `fini(9E).

Note – In the Oracle Solaris OS, only the loadable module routines must be visible outside the driver object module. Other routines can have the storage class static.
Autoconfiguration Entry Points

Drivers are required to implement the attach(9E), detach(9E), and getinfo(9E) entry points for device autoconfiguration. Drivers can also implement the optional entry point probe(9E) in cases where devices do not identify themselves during boot-up, such as SCSI target devices. See Chapter 6, “Driver Autoconfiguration,” for more information on these routines.

Kernel Statistics Entry Points

The Oracle Solaris platform provides a rich set of interfaces to maintain and export kernel-level statistics, also known as kstats. Drivers are free to use these interfaces to export driver and device statistics that can be used by user applications to observe the internal state of the driver. Two entry points are provided for working with kernel statistics:

- ks_snapshot(9E) captures kstats at a specific time.
- ks_update(9E) can be used to update kstat data at will. ks_update() is useful in situations where a device is set up to track kernel data but extracting that data is time-consuming.

For further information, see the kstat_create(9F) and kstat(9S) man pages. See also “Kernel Statistics” on page 524.

Power Management Entry Point

Drivers for hardware devices that provide Power Management functionality can support the optional power(9E) entry point. See Chapter 12, “Power Management,” for details about this entry point.

System Quiesce Entry Point

A driver that manages devices must implement the quiesce(9E) entry point. Drivers that do not manage devices can set the devo_quiesce field in the dev_ops structure to ddi_quiesce_not_needed(). The quiesce() function can be called only when the system is single-threaded at high PIL (priority interrupt level) with preemption disabled. Therefore, this function must not be blocked. If a device has a defined reset state configuration, the driver should return that device to that reset state as part of the quiesce operation. An example of this case is Fast Reboot, where firmware is bypassed when booting to a new operating system image.

Summary of Common Entry Points

The following table lists entry points that can be used by all types of drivers.

<table>
<thead>
<tr>
<th>Category / Entry Point</th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cb_ops Entry Points</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Device Driver Entry Points

#### Table 1–1 Entry Points for All Driver Types (Continued)

<table>
<thead>
<tr>
<th>Category / EntryPoint</th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>open(9E)</td>
<td>Required</td>
<td>Gets access to a device. Additional information:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ &quot;open() Entry Point (Character Drivers)&quot; on page 264</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ &quot;open() Entry Point (Block Drivers)&quot; on page 291</td>
</tr>
<tr>
<td>close(9E)</td>
<td>Required</td>
<td>Gives up access to a device. The version of close() for STREAMS drivers has a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>different signature than character and block drivers. Additional information:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ &quot;close() Entry Point (Character Drivers)&quot; on page 266</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ &quot;close() Entry Point (Block Drivers)&quot; on page 292</td>
</tr>
</tbody>
</table>

#### Loadable Module Entry Points

| _init(9E) | Required | Initializes a loadable module. Additional information:                       |
|           |         |  ■ "Loadable Driver Interfaces" on page 97                                   |
| _fini(9E) | Required | Prepares a loadable module for unloading. Required for all driver types.     |
|           |         |  Additional information: "Loadable Driver Interfaces" on page 97            |
| _info(9E) | Required | Returns information about a loadable module. Additional information:        |
|           |         |  "Loadable Driver Interfaces" on page 97                                    |

#### Autoconfiguration Entry Points

| attach(9E) | Required | Adds a device to the system as part of initialization. Also used to resume   |
|            |         |  a system that has been suspended. Additional information:                  |
|            |         |  ■ "attach() Entry Point" on page 104                                        |
| detach(9E) | Required | Detaches a device from the system. Also, used to suspend a device temporarily. |
|            |         |  Additional information: "detach() Entry Point" on page 109                 |
| getinfo(9E)| Required | Gets device information that is specific to the driver, such as the mapping  |
|            |         |  between a device number and the corresponding instance. Additional         |
|            |         |  information:                                                                |
|            |         |  ■ "getinfo() Entry Point" on page 110                                       |
|            |         |  ■ "getinfo() Entry Point (SCSI Target Drivers)" on page 320.                |
| probe(9E)  | See Description | Determines if a non-self-identifying device is present. Required for a device |
|            |         |  that cannot identify itself. Additional information:                      |
|            |         |  ■ "probe() Entry Point" on page 101                                         |
|            |         |  ■ "probe() Entry Point (SCSI Target Drivers)" on page 315                  |

#### Kernel Statistics Entry Points

| ks_snapshot(9E) | Optional | Takes a snapshot of kstat(9S) data. Additional information:                 |
|                |         |  "Kernel Statistics" on page 524                                            |
| ks_update(9E)  | Optional | Updates kstat(9S) data dynamically. Additional information:                 |
|                |         |  "Kernel Statistics" on page 524                                            |

#### Power Management Entry Point
Device Driver Entry Points

TABLE 1–1 Entry Points for All Driver Types  (Continued)

<table>
<thead>
<tr>
<th>Category / Entry Point</th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>power(9E)</td>
<td>Required</td>
<td>Sets the power level of a device. If not used, set to NULL. Additional information: &quot;power() Entry Point&quot; on page 208</td>
</tr>
</tbody>
</table>

**System Quiesce Entry Point**

quiesce(9E) See Description Quiesces a device so that the device no longer generates interrupts or modifies or accesses memory.

**Miscellaneous Entry Points**

prop_op(9E) See Description Reports driver property information. Required unless ddi_prop_op(9F) is substituted. Additional information:
- "Creating and Updating Properties" on page 78
- "prop_op() Entry Point" on page 81

dump(9E) See Description Dumps memory to a device during system failure. Required for any device that is to be used as the dump device during a panic. Additional information:
- "dump() Entry Point (Block Drivers)" on page 303
- "Dump Handling" on page 328

identify(9E) Obsolete Do not use this entry point. Assign nulldev(9F) to this entry point in the dev_ops structure.

Entry Points for Block Device Drivers

Devices that support a file system are known as block devices. Drivers written for these devices are known as block device drivers. Block device drivers take a file system request, in the form of a buf(9S) structure, and issue the I/O operations to the disk to transfer the specified block. The main interface to the file system is the strategy(9E) routine. See Chapter 16, “Drivers for Block Devices,” for more information.

A block device driver can also provide a character driver interface to enable utility programs to bypass the file system and to access the device directly. This device access is commonly referred to as the raw interface to a block device.

The following table lists additional entry points that can be used by block device drivers. See also “Entry Points Common to All Drivers” on page 41.

TABLE 1–2 Additional Entry Points for Block Drivers

<table>
<thead>
<tr>
<th>Entry Point</th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
</table>
| aread(9E)   | Optional | Performs an asynchronous read. Drivers that do not support an aread() entry point should use the nodev(9F) error return function. Additional information:
- "Differences Between Synchronous and Asynchronous I/O" on page 269
- "DMA Transfers (Asynchronous)" on page 272 |
TABLE 1–2  Additional Entry Points for Block Drivers  (Continued)

<table>
<thead>
<tr>
<th>Entry Point</th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>awrite(9E)</td>
<td>Optional</td>
<td>Performs an asynchronous write. Drivers that do not support an awrite() entry point should use the nodev(9F) error return function. Additional information: ■ “Differences Between Synchronous and Asynchronous I/O” on page 269 ■ “DMA Transfers (Asynchronous)” on page 272</td>
</tr>
<tr>
<td>print(9E)</td>
<td>Required</td>
<td>Displays a driver message on the system console. Additional information: “print() Entry Point (Block Drivers)” on page 303</td>
</tr>
<tr>
<td>strategy(9E)</td>
<td>Required</td>
<td>Perform block I/O. Additional information: ■ “Canceling DMA Callbacks” on page 169 ■ “DMA Transfers (Synchronous)” on page 271 ■ “strategy() Entry Point” on page 274 ■ “DMA Transfers (Asynchronous)” on page 272 ■ “General Flow of Control” on page 309 ■ “x86 Target Driver Configuration Properties” on page 377</td>
</tr>
</tbody>
</table>

### Entry Points for Character Device Drivers

Character device drivers normally perform I/O in a byte stream. Examples of devices that use character drivers include tape drives and serial ports. Character device drivers can also provide additional interfaces not present in block drivers, such as I/O control (ioctl) commands, memory mapping, and device polling. See Chapter 15, "Drivers for Character Devices," for more information.

The main task of any device driver is to perform I/O, and many character device drivers do what is called byte-stream or character I/O. The driver transfers data to and from the device without using a specific device address. This type of transfer is in contrast to block device drivers, where part of the file system request identifies a specific location on the device.

The read(9E) and write(9E) entry points handle byte-stream I/O for standard character drivers. See "I/O Request Handling" on page 266 for more information.

The following table lists additional entry points that can be used by character device drivers. For other entry points, see "Entry Points Common to All Drivers" on page 41.

TABLE 1–3  Additional Entry Points for Character Drivers

<table>
<thead>
<tr>
<th>Entry Point</th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chpoll(9E)</td>
<td>Optional</td>
<td>Polls events for a non-STREAMS character driver. Additional information: “Multiplexing I/O on File Descriptors” on page 276</td>
</tr>
</tbody>
</table>
**Device Driver Entry Points**

<table>
<thead>
<tr>
<th>Entry Point</th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ioctl (9E)</td>
<td>Optional</td>
<td>Performs a range of I/O commands for character drivers. <code>ioctl()</code> routines must make sure that user data is copied into or out of the kernel address space explicitly using <code>copyin(9F)</code>, <code>copyout(9F)</code>, <code>ddi_copyin(9F)</code>, and <code>ddi_copyout(9F)</code>, as appropriate. Additional information:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ &quot;ioctl() Entry Point (Character Drivers)&quot; on page 278</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ &quot;Well Known ioctl Interfaces&quot; on page 593</td>
</tr>
<tr>
<td>read (9E)</td>
<td>Required</td>
<td>Reads data from a device. Additional information:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ &quot;Vectored I/O&quot; on page 267</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ &quot;Differences Between Synchronous and Asynchronous I/O&quot; on page 269</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ &quot;Programmed I/O Transfers&quot; on page 269</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ &quot;DMA Transfers (Synchronous)&quot; on page 271</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ &quot;General Flow of Control&quot; on page 309</td>
</tr>
<tr>
<td>segmap (9E)</td>
<td>Optional</td>
<td>Maps device memory into user space. Additional information:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ &quot;Exporting the Mapping&quot; on page 177</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ &quot;Allocating Kernel Memory for User Access&quot; on page 183</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ &quot;Associating User Mappings With Driver Notifications&quot; on page 197</td>
</tr>
<tr>
<td>write (9E)</td>
<td>Required</td>
<td>Writes data to a device. Additional information:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ &quot;Device Access Functions&quot; on page 118</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ &quot;Vectored I/O&quot; on page 267</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ &quot;Differences Between Synchronous and Asynchronous I/O&quot; on page 269</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ &quot;Programmed I/O Transfers&quot; on page 269</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ &quot;DMA Transfers (Synchronous)&quot; on page 271</td>
</tr>
<tr>
<td></td>
<td></td>
<td>■ &quot;General Flow of Control&quot; on page 309</td>
</tr>
</tbody>
</table>

**Entry Points for STREAMS Device Drivers**

STREAMS is a separate programming model for writing a character driver. Devices that receive data asynchronously, such as terminal and network devices, are suited to a STREAMS implementation. STREAMS device drivers must provide the loading and autoconfiguration support described in Chapter 6, “Driver Autoconfiguration.” See the [STREAMS Programming Guide](#) for additional information on how to write STREAMS drivers.

The following table lists additional entry points that can be used by STREAMS device drivers. For other entry points, see “Entry Points Common to All Drivers” on page 41 and “Entry Points for Character Device Drivers” on page 45.
### Entry Points for STREAMS Drivers

<table>
<thead>
<tr>
<th>Entry Point</th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>put(9E)</td>
<td>See Description</td>
<td>Coordinates the passing of messages from one queue to the next queue in a stream. Required, except for the side of the driver that reads data. Additional information: <em>STREAMS Programming Guide</em></td>
</tr>
<tr>
<td>srv(9E)</td>
<td>Required</td>
<td>Manipulate messages in a queue. Additional information: <em>STREAMS Programming Guide</em></td>
</tr>
</tbody>
</table>

### Entry Points for Memory Mapped Devices

For certain devices, such as frame buffers, providing application programs with direct access to device memory is more efficient than byte-stream I/O. Applications can map device memory into their address spaces using the `mmap(2)` system call. To support memory mapping, device drivers implement `segmap(9E)` and `devmap(9E)` entry points. For information on `devmap(9E)`, see Chapter 10, "Mapping Device and Kernel Memory." For information on `segmap(9E)`, see Chapter 15, "Drivers for Character Devices."

Drivers that define the `devmap(9E)` entry point usually do not define `read(9E)` and `write(9E)` entry points, because application programs perform I/O directly to the devices after calling `mmap(2)`.

The following table lists additional entry points that can be used by character device drivers that use the devmap framework to perform memory mapping. For other entry points, see "Entry Points Common to All Drivers" on page 41 and "Entry Points for Character Device Drivers" on page 45.

<table>
<thead>
<tr>
<th>Entry Point</th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>devmap(9E)</td>
<td>Required</td>
<td>Validates and translates virtual mapping for a memory-mapped device. Additional information: &quot;Exporting the Mapping&quot; on page 177</td>
</tr>
<tr>
<td>devmap_access(9E)</td>
<td>Optional</td>
<td>Notifies drivers when an access is made to a mapping with validation or protection problems. Additional information: &quot;devmap_access() Entry Point&quot; on page 191</td>
</tr>
<tr>
<td>devmap_contextmgt(9E)</td>
<td>Required</td>
<td>Performs device context switching on a mapping. Additional information: &quot;devmap_contextmgt() Entry Point&quot; on page 193</td>
</tr>
<tr>
<td>devmap_dup(9E)</td>
<td>Optional</td>
<td>Duplicates a device mapping. Additional information: &quot;devmap_dup() Entry Point&quot; on page 194</td>
</tr>
<tr>
<td>devmap_map(9E)</td>
<td>Optional</td>
<td>Creates a device mapping. Additional information: &quot;devmap_map() Entry Point&quot; on page 190</td>
</tr>
</tbody>
</table>
**Entry Points for Character Drivers That Use devmap for Memory Mapping**

(Continued)

<table>
<thead>
<tr>
<th>Entry Point</th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>devmap_unmap(9E)</td>
<td>Optional</td>
<td>Cancels a device mapping. Additional information: &quot;devmap_unmap() Entry Point&quot; on page 195</td>
</tr>
</tbody>
</table>

**Entry Points for Network Device Drivers**

See Table 19–1 for a list of entry points for network device drivers that use the Generic LAN Driver version 3 (GLDv3) framework. For more information, see “GLDv3 Network Device Driver Framework” on page 381 and “GLDv3 MAC Registration Functions” on page 382 in Chapter 19, “Drivers for Network Devices.”

**Entry Points for SCSI HBA Drivers**

The following table lists additional entry points that can be used by SCSI HBA device drivers. For information on the SCSI HBA transport structure, see `scsi_hba_tran(9S)`. For other entry points, see “Entry Points Common to All Drivers” on page 41 and “Entry Points for Character Device Drivers” on page 45.

<table>
<thead>
<tr>
<th>Entry Point</th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tran_abort(9E)</td>
<td>Required</td>
<td>Aborts a specified SCSI command that has been transported to a SCSI Host Bus Adapter (HBA) driver. Additional information: &quot;tran_abort() Entry Point&quot; on page 373</td>
</tr>
<tr>
<td>tran_bus_reset(9E)</td>
<td>Optional</td>
<td>Resets a SCSI bus. Additional information: &quot;tran_bus_reset() Entry Point&quot; on page 374</td>
</tr>
<tr>
<td>tran_destroy_pkt(9E)</td>
<td>Required</td>
<td>Frees resources that are allocated for a SCSI packet. Additional information: &quot;tran_destroy_pkt() Entry Point&quot; on page 360</td>
</tr>
<tr>
<td>tran_dmafree(9E)</td>
<td>Required</td>
<td>Frees DMA resources that have been allocated for a SCSI packet. Additional information: &quot;tran_dmafree() Entry Point&quot; on page 361</td>
</tr>
<tr>
<td>tran_getcap(9E)</td>
<td>Required</td>
<td>Gets the current value of a specific capability that is provided by the HBA driver. Additional information: &quot;tran_getcap() Entry Point&quot; on page 368</td>
</tr>
<tr>
<td>tran_init_pkt(9E)</td>
<td>Required</td>
<td>Allocate and initialize resources for a SCSI packet. Additional information: &quot;Resource Allocation&quot; on page 353</td>
</tr>
<tr>
<td>tran_quiesce(9E)</td>
<td>Optional</td>
<td>Stop all activity on a SCSI bus, typically for dynamic reconfiguration. Additional information: &quot;Dynamic Reconfiguration&quot; on page 375</td>
</tr>
<tr>
<td>tran_reset(9E)</td>
<td>Required</td>
<td>Resets a SCSI bus or target device. Additional information: &quot;tran_reset() Entry Point&quot; on page 373</td>
</tr>
</tbody>
</table>
## Entry Points for PC Card Drivers

The following table lists additional entry points that can be used by PC Card device drivers. For other entry points, see “Entry Points Common to All Drivers” on page 41 and “Entry Points for Character Device Drivers” on page 45.

### Table 1-7 Entry Points for PC Card Drivers Only

<table>
<thead>
<tr>
<th>Entry Point</th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>csx_event_handler(9E)</td>
<td>Required</td>
<td>Handles events for a PC Card driver. The driver must call the csx_RegisterClient(9F) function explicitly to set the entry point instead of using a structure field like cb_ops.</td>
</tr>
</tbody>
</table>
Considerations in Device Driver Design

A device driver must be compatible with the Oracle Solaris OS, both as a consumer and provider of services. This section discusses the following issues, which should be considered in device driver design:

- “DDI/DKI Facilities” on page 50
- “Driver Context” on page 52
- “Returning Errors” on page 53
- “Dynamic Memory Allocation” on page 53
- “Hotplugging” on page 54

DDI/DKI Facilities

The Oracle Solaris DDI/DKI interfaces are provided for driver portability. With DDI/DKI, developers can write driver code in a standard fashion without having to worry about hardware or platform differences. This section describes aspects of the DDI/DKI interfaces.

Device IDs

The DDI interfaces enable drivers to provide a persistent, unique identifier for a device. The device ID can be used to identify or locate a device. The ID is independent of the device’s name or number (dev_t). Applications can use the functions defined in libdevid(3LIB) to read and manipulate the device IDs registered by the drivers.

Device Properties

The attributes of a device or device driver are specified by properties. A property is a name-value pair. The name is a string that identifies the property with an associated value. Properties can be defined by the FCode of a self-identifying device, by a hardware configuration file (see the driver.conf(4) man page), or by the driver itself using the ddi_prop_update(9F) family of routines.

Interrupt Handling

The DDI/DKI addresses the following aspects of device interrupt handling:

- Registering device interrupts with the system
- Removing device interrupts
- Dispatching interrupts to interrupt handlers

Device interrupt sources are contained in a property called interrupt, which is either provided by the PROM of a self-identifying device, in a hardware configuration file, or by the booting system on the x86 platform.
Callback Functions

Certain DDI mechanisms provide a callback mechanism. DDI functions provide a mechanism for scheduling a callback when a condition is met. Callback functions can be used for the following typical conditions:

- A transfer has completed
- A resource has become available
- A time-out period has expired

Callback functions are somewhat similar to entry points, for example, interrupt handlers. DDI functions that allow callbacks expect the callback function to perform certain tasks. In the case of DMA routines, a callback function must return a value indicating whether the callback function needs to be rescheduled in case of a failure.

Callback functions execute as a separate interrupt thread. Callbacks must handle all the usual multithreading issues.

**Note** – A driver must cancel all scheduled callback functions before detaching a device.

Software State Management

To assist device driver writers in allocating state structures, the DDI/DKI provides a set of memory management routines called the software state management routines, also known as the soft-state routines. These routines dynamically allocate, retrieve, and destroy memory items of a specified size, and hide the details of list management. An instance number is used to identify the desired memory item. This number is typically the instance number assigned by the system.

Routines are provided for the following tasks:

- Initialize a driver's soft-state list
- Allocate space for an instance of a driver's soft state
- Retrieve a pointer to an instance of a driver's soft state
- Free the memory for an instance of a driver's soft state
- Finish using a driver's soft-state list

See “Loadable Driver Interfaces” on page 97 for an example of how to use these routines.

Programmed I/O Device Access

Programmed I/O device access is the act of reading and writing of device registers or device memory by the host CPU. The Oracle Solaris DDI provides interfaces for mapping a device's registers or memory by the kernel as well as interfaces for reading and writing to device memory from the driver. These interfaces enable drivers to be developed that are platform and bus independent, by automatically managing any difference in device and host endianness as well as by enforcing any memory-store sequence requirements imposed by the device.
Direct Memory Access (DMA)

The Oracle Solaris platform defines a high-level, architecture-independent model for supporting DMA-capable devices. The Oracle Solaris DDI shields drivers from platform-specific details. This concept enables a common driver to run on multiple platforms and architectures.

Layered Driver Interfaces

The DDI/DKI provides a group of interfaces referred to as layered device interfaces (LDI). These interfaces enable a device to be accessed from within the Oracle Solaris kernel. This capability enables developers to write applications that observe kernel device usage. For example, both the `prtconf(1M)` and `fuser(1M)` commands use LDI to enable system administrators to track aspects of device usage. The LDI is covered in more detail in Chapter 14, “Layered Driver Interface (LDI).”

Driver Context

The driver context refers to the condition under which a driver is currently operating. The context limits the operations that a driver can perform. The driver context depends on the executing code that is invoked. Driver code executes in four contexts:

- **User context.** A driver entry point has user context when invoked by a user thread in a synchronous fashion. That is, the user thread waits for the system to return from the entry point that was invoked. For example, the `read(9E)` entry point of the driver has user context when invoked by a `read(2)` system call. In this case, the driver has access to the user area for copying data into and out of the user thread.

- **Kernel context.** A driver function has kernel context when invoked by some part of the kernel. In a block device driver, the `strategy(9E)` entry point can be called by the pageout daemon to write pages to the device. Because the page daemon has no relation to the current user thread, `strategy(9E)` has kernel context in this case.

- **Interrupt context.** Interrupt context is a more restrictive form of kernel context. Interrupt context is invoked as a result of the servicing of an interrupt. Driver interrupt routines operate in interrupt context with an associated interrupt level. Callback routines also operate in an interrupt context. See Chapter 8, “Interrupt Handlers,” for more information.

- **High-level interrupt context.** High-level interrupt context is a more restricted form of interrupt context. If `ddi_intr_hilevel(9F)` indicates that an interrupt is high level, the driver interrupt handler runs in high-level interrupt context. See Chapter 8, “Interrupt Handlers,” for more information.

The manual pages in section 9F document the allowable contexts for each function. For example, in kernel context the driver must not call `copyin(9F).`
Returning Errors

Device drivers do not usually print messages, except for unexpected errors such as data corruption. Instead, the driver entry points should return error codes so that the application can determine how to handle the error. Use the `cmn_err(9F)` function to write messages to a system log that can then be displayed on the console.

The format string specifier interpreted by `cmn_err(9F)` is similar to the `printf(3C)` format string specifier, with the addition of the format `%b`, which prints bit fields. The first character of the format string can have a special meaning. Calls to `cmn_err(9F)` also specify the message level, which indicates the severity label to be printed. See the `cmn_err(9F)` man page for more details.

The level `CE_PANIC` has the side effect of crashing the system. This level should be used only if the system is in such an unstable state that to continue would cause more problems. The level can also be used to get a system core dump when debugging. `CE_PANIC` should not be used in production device drivers.

Dynamic Memory Allocation

Device drivers must be prepared to simultaneously handle all attached devices that the drivers claim to drive. The number of devices that the driver handles should not be limited. All per-device information must be dynamically allocated.

```c
void *kmem_alloc(size_t size, int flag);
```

The standard kernel memory allocation routine is `kmem_alloc(9F)`. `kmem_alloc()` is similar to the C library routine `malloc(3C)`, with the addition of the `flag` argument. The `flag` argument can be either `KM_SLEEP` or `KM_NOSLEEP`, indicating whether the caller is willing to block if the requested size is not available. If `KM_NOSLEEP` is set and memory is not available, `kmem_alloc(9F)` returns `NULL`.

`kmem_zalloc(9F)` is similar to `kmem_alloc(9F)`, but also clears the contents of the allocated memory.

**Note** - Kernel memory is a limited resource, not pageable, and competes with user applications and the rest of the kernel for physical memory. Drivers that allocate a large amount of kernel memory can cause system performance to degrade.

```c
void kmem_free(void *cp, size_t size);
```

Memory allocated by `kmem_alloc(9F)` or by `kmem_zalloc(9F)` is returned to the system with `kmem_free(9F)`. `kmem_free()` is similar to the C library routine `free(3C)`, with the addition of the size argument. Drivers must keep track of the size of each allocated object in order to call `kmem_free(9F)` later.
Hotplugging

This manual does not highlight hotplugging information. If you follow the rules and suggestions for writing device drivers given in this book, your driver should be able to handle hotplugging. In particular, make sure that both autoconfiguration (see Chapter 6, “Driver Autoconfiguration”) and detach(9E) work correctly in your driver. In addition, if you are designing a driver that uses power management, you should follow the information given in Chapter 12, “Power Management.” SCSI HBA drivers might need to add a cb_ops structure to their dev_ops structure (see Chapter 18, “SCSI Host Bus Adapter Drivers”) to take advantage of hotplugging capabilities.

Previous versions of the Oracle Solaris OS required hotpluggable drivers to include a DT_HOTPLUG property, but this property is no longer required. Driver writers are free, however, to include and use the DT_HOTPLUG property as they see fit.
A device driver needs to work transparently as an integral part of the operating system. Understanding how the kernel works is a prerequisite for learning about device drivers. This chapter provides an overview of the Oracle Solaris kernel and device tree. For an overview of how device drivers work, see Chapter 1, “Overview of Oracle Solaris Device Drivers.”

This chapter provides information on the following subjects:

- “What Is the Kernel?” on page 55
- “Multithreaded Execution Environment” on page 57
- “Virtual Memory” on page 57
- “Devices as Special Files” on page 57
- “DDI/DKI Interfaces” on page 58
- “Device Tree Components” on page 59
- “Displaying the Device Tree” on page 60
- “Binding a Driver to a Device” on page 62

What Is the Kernel?

The Oracle Solaris kernel is a program that manages system resources. The kernel insulates applications from the system hardware and provides them with essential system services such as input/output (I/O) management, virtual memory, and scheduling. The kernel consists of object modules that are dynamically loaded into memory when needed.

The Oracle Solaris kernel can be divided logically into two parts: the first part, referred to as the kernel, manages file systems, scheduling, and virtual memory. The second part, referred to as the I/O subsystem, manages the physical components.

The kernel provides a set of interfaces for applications to use that are accessible through system calls. System calls are documented in section 2 of the Reference Manual Collection (see Intro(2)). Some system calls are used to invoke device drivers to perform I/O. Device drivers are loadable kernel modules that manage data transfers while insulating the rest of the kernel from...
the device hardware. To be compatible with the operating system, device drivers need to be able to accommodate such features as multithreading, virtual memory addressing, and both 32-bit and 64-bit operation.

The following figure illustrates the kernel. The kernel modules handle system calls from application programs. The I/O modules communicate with hardware.

**FIGURE 2-1  Oracle Solaris Kernel**

The kernel provides access to device drivers through the following features:

- **Device-to-driver mapping.** The kernel maintains the *device tree*. Each node in the tree represents a virtual or a physical device. The kernel binds each node to a driver by matching the device node name with the set of drivers installed in the system. The device is made accessible to applications only if there is a driver binding.
DDI/DKI interfaces. DDI/DKI (Device Driver Interface/Driver-Kernel Interface) interfaces standardize interactions between the driver and the kernel, the device hardware, and the boot/configuration software. These interfaces keep the driver independent from the kernel and improve the driver’s portability across successive releases of the operating system on a particular machine.

LDI. The LDI (Layered Driver Interface) is an extension of the DDI/DKI. The LDI enables a kernel module to access other devices in the system. The LDI also enables you to determine which devices are currently being used by the kernel. See Chapter 14, "Layered Driver Interface (LDI)."

Multithreaded Execution Environment

The Oracle Solaris kernel is multithreaded. On a multiprocessor machine, multiple kernel threads can be running kernel code, and can do so concurrently. Kernel threads can also be preempted by other kernel threads at any time.

The multithreading of the kernel imposes some additional restrictions on device drivers. For more information on multithreading considerations, see Chapter 3, "Multithreading." Device drivers must be coded to run as needed at the request of many different threads. For each thread, a driver must handle contention problems from overlapping I/O requests.

Virtual Memory

A complete overview of the Oracle Solaris virtual memory system is beyond the scope of this book, but two virtual memory terms of special importance are used when discussing device drivers: virtual address and address space.

- **Virtual address.** A virtual address is an address that is mapped by the memory management unit (MMU) to a physical hardware address. All addresses directly accessible by the driver are kernel virtual addresses. Kernel virtual addresses refer to the kernel address space.

- **Address space.** An address space is a set of virtual address segments. Each segment is a contiguous range of virtual addresses. Each user process has an address space called the user address space. The kernel has its own address space, called the kernel address space.

Devices as Special Files

Devices are represented in the file system by special files. In the Oracle Solaris OS, these files reside in the /devices directory hierarchy.

Special files can be of type block or character. The type indicates which kind of device driver operates the device. Drivers can be implemented to operate on both types. For example, disk drivers export a character interface for use by the `fsck(1)` and `mkfs(1)` utilities, and a block interface for use by the file system.
Associated with each special file is a device number (dev_t). A device number consists of a major number and a minor number. The major number identifies the device driver associated with the special file. The minor number is created and used by the device driver to further identify the special file. Usually, the minor number is an encoding that is used to identify which device instance the driver should access and which type of access should be performed. For example, the minor number can identify a tape device used for backup and can specify that the tape needs to be rewound when the backup operation is complete.

**DDI/DKI Interfaces**

In System V Release 4 (SVR4), the interface between device drivers and the rest of the UNIX kernel was standardized as the DDI/DKI. The DDI/DKI is documented in section 9 of the Reference Manual Collection. Section 9E documents driver entry points, section 9F documents driver-callable functions, and section 9S documents kernel data structures used by device drivers. See Intro(9E), Intro(9F), and Intro(9S).

The DDI/DKI is intended to standardize and document all interfaces between device drivers and the rest of the kernel. In addition, the DDI/DKI enables source and binary compatibility for drivers on any machine that runs the Oracle Solaris OS, regardless of the processor architecture, whether SPARC or x86. Drivers that use only kernel facilities that are part of the DDI/DKI are known as DDI/DKI-compliant device drivers.

The DDI/DKI enables you to write platform-independent device drivers for any machine that runs the Oracle Solaris OS. These binary-compatible drivers enable you to more easily integrate third-party hardware and software into any machine that runs the Oracle Solaris OS. The DDI/DKI is architecture independent, which enables the same driver to work across a diverse set of machine architectures.

Platform independence is accomplished by the design of DDI in the following areas:

- Dynamic loading and unloading of modules
- Power management
- Interrupt handling
- Accessing the device space from the kernel or a user process, that is, register mapping and memory mapping
- Accessing kernel or user process space from the device using DMA services
- Managing device properties
Overview of the Device Tree

Devices in the Oracle Solaris OS are represented as a tree of interconnected device information nodes. The device tree describes the configuration of loaded devices for a particular machine.

Device Tree Components

The system builds a tree structure that contains information about the devices connected to the machine at boot time. The device tree can also be modified by dynamic reconfiguration operations while the system is in normal operation. The tree begins at the root device node, which represents the platform.

Below the root node are the branches of the device tree. A branch consists of one or more bus nexus devices and a terminating leaf device.

A bus nexus device provides bus mapping and translation services to subordinate devices in the device tree. PCI - PCI bridges, PCMCIA adapters, and SCSI HBAs are all examples of nexus devices. The discussion of writing drivers for nexus devices is limited to the development of SCSI HBA drivers (see Chapter 18, “SCSI Host Bus Adapter Drivers”).

Leaf devices are typically peripheral devices such as disks, tapes, network adapters, frame buffers, and so forth. Leaf device drivers export the traditional character driver interfaces and block driver interfaces. The interfaces enable user processes to read data from and write data to either storage or communication devices.

The system goes through the following steps to build the tree:

1. The CPU is initialized and searches for firmware.
2. The main firmware (OpenBoot, Basic Input/Output System (BIOS), or Boot conf) initializes and creates the device tree with known or self-identifying hardware.
3. When the main firmware finds compatible firmware on a device, the main firmware initializes the device and retrieves the device's properties.
4. The firmware locates and boots the operating system.
5. The kernel starts at the root node of the tree, searches for a matching device driver, and binds that driver to the device.
6. If the device is a nexus, the kernel looks for child devices that have not been detected by the firmware. The kernel adds any child devices to the tree below the nexus node.
7. The kernel repeats the process from Step 5 until no further device nodes need to be created.

Each driver exports a device operations structure dev_ops(9S) to define the operations that the device driver can perform. The device operations structure contains function pointers for generic operations such as attach(9E), detach(9E), and getinfo(9E). The structure also contains a pointer to a set of operations specific to bus nexus drivers and a pointer to a set of operations specific to leaf drivers.
The tree structure creates a parent-child relationship between nodes. This parent-child relationship is the key to architectural independence. When a leaf or bus nexus driver requires a service that is architecturally dependent in nature, that driver requests its parent to provide the service. This approach enables drivers to function regardless of the architecture of the machine or the processor. A typical device tree is shown in the following figure.

**Figure 2-2  Example Device Tree**

The nexus nodes can have one or more children. The leaf nodes represent individual devices.

**Displaying the Device Tree**

The device tree can be displayed in three ways:

- The `libdevinfo` library provides interfaces to access the contents of the device tree programatically.
- The `prtconf(1M)` command displays the complete contents of the device tree.
- The `/devices` hierarchy is a representation of the device tree. Use the `ls(1)` command to view the hierarchy.

*Note – `/devices` displays only devices that have drivers configured into the system. The `prtconf(1M)` command shows all device nodes regardless of whether a driver for the device exists on the system.*
libdevinfo Library

The libdevinfo library provides interfaces for accessing all public device configuration data. See the \libdevinfo(3LIB) man page for a list of interfaces.

prtconf Command

The following excerpted \prtconf(1M) command example displays all the devices in the system.

Memory size: 128 Megabytes
System Peripherals (Software Nodes):

```
SUNW,Ultra-5 10
    packages (driver not attached)
    terminal-emulator (driver not attached)
    deblocker (driver not attached)
    obp-tftp (driver not attached)
    disk-label (driver not attached)
    SUNW,builtin-drivers (driver not attached)
    sun-keyboard (driver not attached)
    ufs-file-system (driver not attached)
        chosen (driver not attached)
        openprom (driver not attached)
        client-services (driver not attached)
        options, instance #0
        aliases (driver not attached)
        memory (driver not attached)
        virtual-memory (driver not attached)
    pci, instance #0
        pci, instance #0
            ebus, instance #0
                auxio (driver not attached)
                power, instance #0
                    SUNW,pll (driver not attached)
                    se, instance #0
                    su, instance #0
                    su, instance #1
                    eccp (driver not attached)
                    fdthree, instance #0
                    eeprom (driver not attached)
                    flashprom (driver not attached)
                    SUNW,CS4231 (driver not attached)
                    network, instance #0
                    SUNW,m64B (driver not attached)
                    ide, instance #0
                        disk (driver not attached)
                        cdrom (driver not attached)
                        dad, instance #0
                        sd, instance #15
    pci, instance #1
        pci, instance #0
            pcl880e,1000 (driver not attached)
                    SUNW,hme, instance #1
                    SUNW,isptwo, instance #0
                        sd (driver not attached)
                        st (driver not attached)
                        sd, instance #0 (driver not attached)
```
sd, instance #1 (driver not attached)
sd, instance #2 (driver not attached)
...
SUNW, UltraSPARC-IIi (driver not attached)
SUNW, ffb, instance #0
pseudo, instance #0

/devices Directory

The /devices hierarchy provides a namespace that represents the device tree. Following is an abbreviated listing of the /devices namespace. The sample output corresponds to the example device tree and "prtconf(1M)" output shown previously.

Binding a Driver to a Device

In addition to constructing the device tree, the kernel determines the drivers that are used to manage the devices.

Binding a driver to a device refers to the process by which the system selects a driver to manage a particular device. The binding name is the name that links a driver to a unique device node in the device information tree. For each device in the device tree, the system attempts to choose a driver from a list of installed drivers.
Each device node has an associated *name* property. This property can be assigned either from an external agent, such as the PROM, during system boot or from a *driver.conf* configuration file. In any case, the *name* property represents the node name assigned to a device in the device tree. The node name is the name visible in /devices and listed in the *prtconf(1M)* output.

A device node can have an associated *compatible* property as well. The *compatible* property contains an ordered list of one or more possible driver names or driver aliases for the device.

The system uses both the *compatible* and the *name* properties to select a driver for the device. The system first attempts to match the contents of the *compatible* property, if the *compatible* property exists, to a driver on the system. Beginning with the first driver name on the *compatible* property list, the system attempts to match the driver name to a known driver on the system. Each entry on the list is processed until the system either finds a match or reaches the end of the list.

If the contents of either the *name* property or the *compatible* property match a driver on the system, then that driver is bound to the device node. If no match is found, no driver is bound to the device node.

**Generic Device Names**

Some devices specify a *generic* device name as the value for the *name* property. Generic device names describe the function of a device without actually identifying a specific driver for the device. For example, a SCSI host bus adapter might have a generic device name of *scsi*. An Ethernet device might have a generic device name of *ethernet*. 
The `compatible` property enables the system to determine alternate driver names for devices with a generic device name, for example, `glm` for `scsi` HBA device drivers or `hme` for ethernet device drivers.

Devices with generic device names are required to supply a `compatible` property.

**Note** – For a complete description of generic device names, see the IEEE 1275 Open Firmware Boot Standard.

The following figure shows a device node with a specific device name. The driver binding name `SUNW,ffb` is the same name as the device node name.

**FIGURE 2-4** Specific Driver Node Binding

```plaintext
Device Node A

    name = SUNW,ffb
    binding name = SUNW,ffb

/devices/SUNW,ffb@le,0:ffb0
```

The following figure shows a device node with the generic device name `display`. The driver binding name `SUNW,ffb` is the first name on the `compatible` property driver list that matches a driver on the system driver list. In this case, `display` is a generic device name for frame buffers.
Overview of the Device Tree

Figure 2-5  Generic Driver Node Binding

Device Node B

```
name = display
compatible = fast_fb
             SUNW,ffb
             slow_fb
binding name = SUNW,ffb
```

/devices/display@le,0:ffb0
This chapter describes the locking primitives and thread synchronization mechanisms of the Oracle Solaris multithreaded kernel. You should design device drivers to take advantage of multithreading. This chapter provides information on the following subjects:

- “Locking Primitives” on page 67
- “Thread Synchronization” on page 70
- “Choosing a Locking Scheme” on page 74

### Locking Primitives

In traditional UNIX systems, every section of kernel code terminates either through an explicit call to `sleep(1)` to give up the processor or through a hardware interrupt. The Oracle Solaris OS operates differently. A kernel thread can be preempted at any time to run another thread. Because all kernel threads share kernel address space and often need to read and modify the same data, the kernel provides a number of locking primitives to prevent threads from corrupting shared data. These mechanisms include mutual exclusion locks, which are also known as mutexes, readers/writer locks, and semaphores.

### Storage Classes of Driver Data

The storage class of data is a guide to whether the driver might need to take explicit steps to control access to the data. The three data storage classes are:

- **Automatic (stack) data.** Every thread has a private stack, so drivers never need to lock automatic variables.
- **Global static data.** Global static data can be shared by any number of threads in the driver. The driver might need to lock this type of data at times.
- **Kernel heap data.** Any number of threads in the driver can share kernel heap data, such as data allocated by `kmem_alloc(9F)`. The driver needs to protect shared data at all times.
Mutual-Exclusion Locks

A mutual-exclusion lock, or mutex, is usually associated with a set of data and regulates access to that data. Mutexes provide a way to allow only one thread at a time access to that data. The mutex functions are:

- **mutex_destroy(9F)**: Releases any associated storage.
- **mutex_enter(9F)**: Acquires a mutex.
- **mutex_exit(9F)**: Releases a mutex.
- **mutex_init(9F)**: Initializes a mutex.
- **mutex_owned(9F)**: Tests to determine whether the mutex is held by the current thread. To be used in `ASSERT(9F)` only.
- **mutex_tryenter(9F)**: Acquires a mutex if available, but does not block.

### Setting Up Mutexes

Device drivers usually allocate a mutex for each driver data structure. The mutex is typically a field in the structure of type `mutex_t`. `mutex_init(9F)` is called to prepare the mutex for use. This call is usually made at `attach(9E)` time for per-device mutexes and `_init(9E)` time for global driver mutexes.

For example,

```c
struct xxstate *xsp; /* ... */
mutex_init(&xsp->mu, NULL, MUTEX_DRIVER, NULL);
/* ... */
```

For a more complete example of mutex initialization, see Chapter 6, "Driver Autoconfiguration."

The driver must destroy the mutex with `mutex_destroy(9F)` before being unloaded. Destroying the mutex is usually done at `detach(9E)` time for per-device mutexes and `_fini(9E)` time for global driver mutexes.

### Using Mutexes

Every section of the driver code that needs to read or write the shared data structure must do the following tasks:

- Acquire the mutex
- Access the data
- Release the mutex
The scope of a mutex, that is, the data the mutex protects, is entirely up to the programmer. A mutex protects a data structure only if every code path that accesses the data structure does so while holding the mutex.

**Readers/Writer Locks**

A readers/writer lock regulates access to a set of data. The readers/writer lock is so called because many threads can hold the lock simultaneously for reading, but only one thread can hold the lock for writing.

Most device drivers do not use readers/writer locks. These locks are slower than mutexes. The locks provide a performance gain only when they protect commonly read data that is not frequently written. In this case, contention for a mutex could become a bottleneck, so using a readers/writer lock might be more efficient. The readers/writer functions are summarized in the following table. See the `rwlock(9F)` man page for detailed information. The readers/writer lock functions are:

- `rw_destroy(9F)` Destroys a readers/writer lock
- `rw_downgrade(9F)` Downgrades a readers/writer lock holder from writer to reader
- `rw_enter(9F)` Acquires a readers/writer lock
- `rw_exit(9F)` Releases a readers/writer lock
- `rw_init(9F)` Initializes a readers/writer lock
- `rw_read_locked(9F)` Determines whether a readers/writer lock is held for read or write
- `rw_tryenter(9F)` Attempts to acquire a readers/writer lock without waiting
- `rw_tryupgrade(9F)` Attempts to upgrade readers/writer lock holder from reader to writer

**Semaphores**

Counting semaphores are available as an alternative primitive for managing threads within device drivers. See the `semaphore(9F)` man page for more information. The semaphore functions are:

- `sema_destroy(9F)` Destroys a semaphore.
- `sema_init(9F)` Initialize a semaphore.
- `sema_p(9F)` Decrement semaphore and possibly block.
- `sema_p_sig(9F)` Decrement semaphore but do not block if signal is pending. See “Threads Unable to Receive Signals” on page 75.
Thread Synchronization

In addition to protecting shared data, drivers often need to synchronize execution among multiple threads.

Condition Variables in Thread Synchronization

Condition variables are a standard form of thread synchronization. They are designed to be used with mutexes. The associated mutex is used to ensure that a condition can be checked atomically, and that the thread can block on the associated condition variable without missing either a change to the condition or a signal that the condition has changed.

The `condvar(9F)` functions are:

- `cv_broadcast(9F)`: Signals all threads waiting on the condition variable.
- `cv_destroy(9F)`: Destroys a condition variable.
- `cv_init(9F)`: Initializes a condition variable.
- `cv_signal(9F)`: Signals one thread waiting on the condition variable.
- `cv_timedwait(9F)`: Waits for condition, time-out, or signal. See “Threads Unable to Receive Signals” on page 75.
- `cv_timedwait_sig(9F)`: Waits for condition or time-out.
- `cv_wait(9F)`: Waits for condition.
- `cv_wait_sig(9F)`: Waits for condition or return zero on receipt of a signal. See “Threads Unable to Receive Signals” on page 75.

Initializing Condition Variables

Declare a condition variable of type `kcondvar_t` for each condition. Usually, the condition variables are declared in the driver’s soft-state structure. Use `cv_init(9F)` to initialize each condition variable. Similar to mutexes, condition variables are usually initialized at `attach(9E)` time. A typical example of initializing a condition variable is:

```c
cv_init(&xsp->cv, NULL, CV_DRIVER, NULL);
```
For a more complete example of condition variable initialization, see Chapter 6, “Driver Autoconfiguration.”

**Waiting for the Condition**

To use condition variables, follow these steps in the code path waiting for the condition:

1. Acquire the mutex guarding the condition.
2. Test the condition.
3. If the test results do not allow the thread to continue, use `cv_wait(9F)` to block the current thread on the condition. The `cv_wait(9F)` function releases the mutex before blocking the thread and reacquires the mutex before returning. On return from `cv_wait(9F)`, repeat the test.
4. After the test allows the thread to continue, set the condition to its new value. For example, set a device flag to busy.
5. Release the mutex.

**Signaling the Condition**

Follow these steps in the code path to signal the condition:

1. Acquire the mutex guarding the condition.
2. Set the condition.
3. Signal the blocked thread with `cv_broadcast(9F)`.
4. Release the mutex.

The following example uses a busy flag along with mutex and condition variables to force the `read(9E)` routine to wait until the device is no longer busy before starting a transfer.

**EXAMPLE 3–1 Using Mutexes and Condition Variables**

```c
static int
xxread(dev_t dev, struct uio *uiop, cred_t *credp)
{
    struct xxstate *xsp;
    /* ... */
    mutex_enter(&xsp->mu);
    while (xsp->busy)
        cv_wait(&xsp->cv, &xsp->mu);
    xsp->busy = 1;
    mutex_exit(&xsp->mu);
    /* perform the data access */
}

static uint_t
xxintr(caddr_t arg)
{
    struct xxstate *xsp = (struct xxstate *)arg;
    mutex_enter(&xsp->mu);
    /* ... */
}
```

Thread Synchronization
Using Mutexes and Condition Variables

xsp->busy = 0;
cv_broadcast(&xsp->cv);
mutex_exit(&xsp->mu);

EXAMPLE 3–1  Using Mutexes and Condition Variables  (Continued)

cv_wait() and cv_timedwait() Functions

If a thread is blocked on a condition with cv_wait(9F) and that condition does not occur, the thread would wait forever. To avoid that situation, use cv_timedwait(9F), which depends upon another thread to perform a wakeup. cv_timedwait() takes an absolute wait time as an argument. cv_timedwait() returns -1 if the time is reached and the event has not occurred. cv_timedwait() returns a positive value if the condition is met.

cv_timedwait(9F) requires an absolute wait time expressed in clock ticks since the system was last rebooted. The wait time can be determined by retrieving the current value with ddi_get_lbolt(9F). The driver usually has a maximum number of seconds or microseconds to wait, so this value is converted to clock ticks with drv_usectohz(9F) and added to the value from ddi_get_lbolt(9F).

The following example shows how to use cv_timedwait(9F) to wait up to five seconds to access the device before returning EIO to the caller.

EXAMPLE 3–2  Using cv_timedwait()

clock_t cur_ticks, to;
mutex_enter(&xsp->mu);
while (xsp->busy) {
    cur_ticks = ddi_get_lbolt();
    to = cur_ticks + drv_usectohz(5000000); /* 5 seconds from now */
    if (cv_timedwait(&xsp->cv, &xsp->mu, to) == -1) {
        /*
         * The timeout time 'to' was reached without the
         * condition being signaled.
         */
        /* tidy up and exit */
        mutex_exit(&xsp->mu);
        return (EIO);
    }
}
xsp->busy = 1;
mutex_exit(&xsp->mu);

Although device driver writers generally prefer to use cv_timedwait(9F) over cv_wait(9F), sometimes cv_wait(9F) is a better choice. For example, cv_wait(9F) is better if a driver is waiting on the following conditions:
Internal driver state changes, where such a state change might require some command to be executed, or a set amount of time to pass

- Something the driver needs to single-thread
- Some situation that is already managing a possible timeout, as when “A” depends on “B,” and “B” is using `cv_timedwait(9F)`

### cv_wait_sig() Function

A driver might be waiting for a condition that cannot occur or will not happen for a long time. In such cases, the user can send a signal to abort the thread. Depending on the driver design, the signal might not cause the driver to wake up.

`cv_wait_sig(9F)` allows a signal to unblock the thread. This capability enables the user to break out of potentially long waits by sending a signal to the thread with `kill(1)` or by typing the interrupt character. `cv_wait_sig(9F)` returns zero if it is returning because of a signal, or nonzero if the condition occurred. However, see “Threads Unable to Receive Signals” on page 75 for cases in which signals might not be received.

The following example shows how to use `cv_wait_sig(9F)` to allow a signal to unblock the thread.

```c
EXAMPLE3-3 Using cv_wait_sig()
mutex_enter(&xsp->mu);
while (xsp->busy) {
    if (cv_wait_sig(&xsp->cv, &xsp->mu) == 0) {
        /* Signaled while waiting for the condition */
        /* tidy up and exit */
        mutex_exit(&xsp->mu);
        return (EINTR);
    }
}
xsp->busy = 1;
mutex_exit(&xsp->mu);
```

### cv_timedwait_sig() Function

`cv_timedwait_sig(9F)` is similar to `cv_timedwait(9F)` and `cv_wait_sig(9F)`, except that `cv_timedwait_sig()` returns -1 without the condition being signaled after a timeout has been reached, or 0 if a signal (for example, `kill(2)`) is sent to the thread.

For both `cv_timedwait(9F)` and `cv_timedwait_sig(9F)`, time is measured in absolute clock ticks since the last system reboot.
Choosing a Locking Scheme

The locking scheme for most device drivers should be kept straightforward. Using additional locks allows more concurrency but increases overhead. Using fewer locks is less time consuming but allows less concurrency. Generally, use one mutex per data structure, a condition variable for each event or condition the driver must wait for, and a mutex for each major set of data global to the driver. Avoid holding mutexes for long periods of time. Use the following guidelines when choosing a locking scheme:

- Use the multithreading semantics of the entry point to your advantage.
- Make all entry points re-entrant. You can reduce the amount of shared data by changing a static variable to automatic.
- If your driver acquires multiple mutexes, acquire and release the mutexes in the same order in all code paths.
- Hold and release locks within the same functional space.
- Avoid holding driver mutexes when calling DDI interfaces that can block, for example, `kmem_alloc(9F)` with `KM_SLEEP`.

To look at lock usage, use `lockstat(1M)`. `lockstat(1M)` monitors all kernel lock events, gathers frequency and timing data about the events, and displays the data.

See the *Multithreaded Programming Guide* for more details on multithreaded operations.

Potential Locking Pitfalls

Mutexes are not re-entrant by the same thread. If you already own the mutex, attempting to claim this mutex a second time leads to the following panic:

`panic: recursive mutex_enter. mutex %x caller %x`

Releasing a mutex that the current thread does not hold causes this panic:

`panic: mutex_adaptive_exit: mutex not held by thread`

The following panic occurs only on uniprocessors:

`panic: lock_set: lock held and only one CPU`

The `lock_set` panic indicates that a spin mutex is held and will spin forever, because no other CPU can release this mutex. This situation can happen if the driver forgets to release the mutex on one code path or becomes blocked while holding the mutex.
A common cause of the lock_set panic occurs when a device with a high-level interrupt calls a routine that blocks, such as `cv_wait(9F)`. Another typical cause is a high-level handler grabbing an adaptive mutex by calling `mutex_enter(9F)`.

### Threads Unable to Receive Signals

The `sema_p_sig()`, `cv_wait_sig()`, and `cv_timedwait_sig()` functions can be awakened when the thread receives a signal. A problem can arise because some threads are unable to receive signals. For example, when `close(9E)` is called as a result of the application calling `close(2)`, signals can be received. However, when `close(9E)` is called from within the `exit(2)` processing that closes all open file descriptors, the thread cannot receive signals. When the thread cannot receive signals, `sema_p_sig()` behaves as `sema_p()`, `cv_wait_sig()` behaves as `cv_wait()`, and `cv_timedwait_sig()` behaves as `cv_timedwait()`.

Use caution to avoid sleeping forever on events that might never occur. Events that never occur create unkillable (defunct) threads and make the device unusable until the system is rebooted. Signals cannot be received by defunct processes.

To detect whether the current thread is able to receive a signal, use the `ddi_can_receive_sig(9F)` function. If the `ddi_can_receive_sig()` function returns `B_TRUE`, then the above functions can wake up on a received signal. If the `ddi_can_receive_sig()` function returns `B_FALSE`, then the above functions cannot wake up on a received signal. If the `ddi_can_receive_sig()` function returns `B_FALSE`, then the driver should use an alternate means, such as the `timeout(9F)` function, to reawaken.

One important case where this problem occurs is with serial ports. If the remote system asserts flow control and the `close(9E)` function blocks while attempting to drain the output data, a port can be stuck until the flow control condition is resolved or the system is rebooted. Such drivers should detect this case and set up a timer to abort the drain operation when the flow control condition persists for an excessive period of time.

This issue also affects the `qwait_siq(9F)` function, which is described in Chapter 7, "STREAMS Framework – Kernel Level," in STREAMS Programming Guide.
Properties

Properties are user-defined, name-value pair structures that are managed using the DDI/DKI interfaces. This chapter provides information on the following subjects:

- “Device Property Names” on page 78
- “Creating and Updating Properties” on page 78
- “Looking Up Properties” on page 78
- “prop_op() Entry Point” on page 81

Device Properties

Device attribute information can be represented by a name-value pair notation called a property.

For example, device registers and onboard memory can be represented by the reg property. The reg property is a software abstraction that describes device hardware registers. The value of the reg property encodes the device register address location and size. Drivers use the reg property to access device registers.

Another example is the interrupt property. An interrupt property represents the device interrupt. The value of the interrupt property encodes the device-interrupt PIN.

Five types of values can be assigned to properties:

- Byte array – Series of bytes of an arbitrary length
- Integer property – An integer value
- Integer array property – An array of integers
- String property – A null-terminated string
- String array property – A list of null-terminated strings

A property that has no value is considered to be a Boolean property. A Boolean property that exists is true. A Boolean value that does not exist is false.
Device Property Names

Strictly speaking, DDI/DKI software property names have no restrictions. Certain uses are recommended, however. The IEEE 1275-1994 Standard for Boot Firmware defines properties as follows:

A property is a human readable text string consisting of from 1 to 31 printable characters. Property names cannot contain upper case characters or the characters "/", ":", "[", "]" and "@". Property names beginning with the character "+" are reserved for use by future revisions of IEEE 1275-1994.

By convention, underscores are not used in property names. Use a hyphen (-) instead. By convention, property names ending with the question mark character (?) contain values that are strings, typically TRUE or FALSE, for example auto-boot?.

For a discussion of adding properties in driver configuration files, see the driver.conf(4) man page. The pm(9P) and pm-components(9P) man pages show how properties are used in power management. Read the sd(7D) man page as an example of how properties should be documented in device driver man pages.

Creating and Updating Properties

To create a property for a driver, or to update an existing property, use an interface from the DDI driver update interfaces such as ddi_prop_update_int(9F) or ddi_prop_update_string(9F) with the appropriate property type. See Table 4–1 for a list of available property interfaces. These interfaces are typically called from the driver's attach(9E) entry point. In the following example, ddi_prop_update_string() creates a string property called pm-hardware-state with a value of needs-suspend-resume.

```c
/* The following code is to tell cpr that this device
 * needs to be suspended and resumed. */
(void) ddi_prop_update_string(device, dip,
    "pm-hardware-state", "needs-suspend-resume");
```

In most cases, using a ddi_prop_update() routine is sufficient for updating a property. Sometimes, however, the overhead of updating a property value that is subject to frequent change can cause performance problems. See "prop_op() Entry Point" on page 81 for a description of using a local instance of a property value to avoid using ddi_prop_update().

Looking Up Properties

A driver can request a property from its parent, which in turn can ask its parent. The driver can control whether the request can go higher than its parent.
For example, the esp driver in the following example maintains an integer property called `targetx-sync-speed` for each target. The `x` in `targetx-sync-speed` represents the target number. The `prtconf(1M)` command displays driver properties in verbose mode. The following example shows a partial listing for the esp driver.

```
% prtconf -v
...  esp, instance #0
    Driver software properties:
    name <target2-sync-speed> length <4>
    value <0x00000fa0>.
...  
```

The following table provides a summary of the property interfaces.

| Table 4-1  Property Interface Uses |
|-----------|----------------------------------|
| **Family** | **Property Interfaces**          | **Description**                        |
| ddi_prop_lookup | ddi_prop_exists(9F) | Looks up a property and returns successfully if the property exists. Fails if the property does not exist. |
| | ddi_prop_get_int(9F) | Looks up and returns an integer property |
| | ddi_prop_get_int64(9F) | Looks up and returns a 64-bit integer property |
| | ddi_prop_lookup_int_array(9F) | Looks up and returns an integer array property |
| | ddi_prop_lookup_int64_array(9F) | Looks up and returns a 64-bit integer array property |
| | ddi_prop_lookup_string(9F) | Looks up and returns a string property |
| | ddi_prop_lookup_string_array(9F) | Looks up and returns a string array property |
| | ddi_prop_lookup_byte_array(9F) | Looks up and returns a byte array property |
| ddi_prop_update | ddi_prop_update_int(9F) | Updates or creates an integer property |
| | ddi_prop_update_int64(9F) | Updates or creates a single 64-bit integer property |
| | ddi_prop_update_int_array(9F) | Updates or creates an integer array property |
| | ddi_prop_update_string(9F) | Updates or creates a string property |
| | ddi_prop_update_string_array(9F) | Updates or creates a string array property |
| | ddi_prop_update_int64_array(9F) | Updates or creates a 64-bit integer array property |
| | ddi_prop_update_byte_array(9F) | Updates or creates a byte array property |
| ddi_prop_remove | ddi_prop_remove(9F) | Removes a property |
Whenever possible, use 64-bit versions of `int` property interfaces such as `ddi_prop_update_int64(9F)` instead of 32-bit versions such as `ddi_prop_update_int(9F)`.

### Changes to the `driver.conf` File

When a system running the Oracle Solaris OS is upgraded, new versions of drivers may be installed. During the upgrade process, the `driver.conf` file is also updated on the system. The `driver.conf` file is customized by both the vendor and the system administrator. During a system upgrade, the system’s previous configuration should continue to work with the new drivers, the vendor’s `driver.conf` file and with the administrator’s `driver.conf` file.

In the Oracle Solaris 11 release, driver writers have an option to provide a separate `driver.conf` file that will contain the vendor provided driver data. The new `driver.conf` file is stored in the `/etc/driver/drv` directory. This enables the system to retain any administrative changes made to the file. If a driver is found in both the configuration files, the system will merge the files and present a file with the combined properties. The format of the vendor’s `driver.conf` file is the same as the administrator’s driver configuration file.

The vendor and administrative configuration data can now be made available to the driver explicitly via new interfaces. This enables the driver writer to encode any merge logic directly in the driver rather than in the class action scripts or the pre-install scripts and post-install scripts. The customizations made to the administrative file are preserved and the driver can decide on the relevance of the new values to the old values.

In order for a driver to ensure that the above model works well, the driver developer should consider the following:

- Design the driver such that a set of disciplined, configurable options are available.
- Describe fully the driver’s options and the model in the driver documentation and man pages.
- If the driver changes its configuration options such that the administrator settings are either invalidated or superseded, the driver should ensure that previous administrative settings are honored. To lookup the previous configuration, the driver can use the `ddi_prop_lookup(9F)` interface with the property type set to either `DDI_PROP_VENDOR` or `DDI_PROP_ADMIN`.

For example, if a driver supports a timeout configuration in units of seconds and a new version of the driver now supports a finer timeout granularity in units of milliseconds. The new property should be named such that it can be distinguished from the previous property.
The driver should then look up the earlier property on the administrative list and if the property is found, the driver should continue to honor it. The following driver code illustrates the timeout example.

**EXAMPLE 4-1  Driver Check for Locally Configured Timeout Value**

```c
/* Has the timeout been locally configured using the prior option of timeout in units of seconds? */
if (ddi_prop_lookup_int(DDI_DEV_T_ANY, dip, DDI_PROP_ADMIN, "timeout", &ivalues, &n) == DDI_PROP_SUCCESS) {
  if (n != 1) {
    ddi_prop_free(ivalues);
    return (EINVAL);
  }
  /* yes - convert our working timeout accordingly */
  dip->ms_timeout = 1000 * ivalues[0];
  /* record the new parameter setting for confirmation */
  (void) ddi_prop_update_int(DDI_DEV_T_NONE, dip, "ms-timeout", dip->ms_timeout);
  ddi_prop_free(ivalues);
}
```

The `prtconf(1M)` command displays the driver properties and the new `-u` option can be used to display the original property value and the changed property value.

**prop_op() Entry Point**

The `prop_op(9E)` entry point is generally required for reporting device properties or driver properties to the system. If the driver does not need to create or manage its own properties, then the `ddi_prop_op(9F)` function can be used for this entry point.

`ddi_prop_op(9F)` can be used as the `prop_op(9E)` entry point for a device driver when `ddi_prop_op()` is defined in the driver's `cb_ops(9S)` structure. `ddi_prop_op()` enables a leaf device to search for and obtain property values from the device's property list.

If the driver has to maintain a property whose value changes frequently, you should define a driver-specific `prop_op()` routine within the `cb_ops` structure instead of calling `ddi_prop_op()`. This technique avoids the inefficiency of using `ddi_prop_update()` repeatedly. The driver should then maintain a copy of the property value either within its soft-state structure or in a driver variable.

The `prop_op(9E)` entry point reports the values of specific driver properties and device properties to the system. In many cases, the `ddi_prop_op(9F)` routine can be used as the driver's `prop_op()` entry point in the `cb_ops(9S)` structure. `ddi_prop_op()` performs all of the required processing. `ddi_prop_op()` is sufficient for drivers that do not require special processing when handling device property requests.
However, sometimes the driver must provide a `prop_op()` entry point. For example, if a driver maintains a property whose value changes frequently, updating the property with `ddi_prop_update(9F)` for each change is not efficient. Instead, the driver should maintain a shadow copy of the property in the instance’s soft state. The driver would then update the shadow copy when the value changes without using any of the `ddi_prop_update()` routines. The `prop_op()` entry point must intercept requests for this property and use one of the `ddi_prop_update()` routines to update the value of the property before passing the request to `ddi_prop_op()` to process the property request.

In the following example, `prop_op()` intercepts requests for the `temperature` property. The driver updates a variable in the state structure whenever the property changes. However, the property is updated only when a request is made. The driver then uses `ddi_prop_op()` to process the property request. If the property request is not specific to a device, the driver does not intercept the request. This situation is indicated when the value of the `dev` parameter is equal to `DDI_DEV_T_ANY`, the wildcard device number.

**EXAMPLE 4-2  prop_op() Routine**

```c
static int
xx_prop_op(dev_t dev, dev_info_t *dip, ddi_prop_op_t prop_op,
int flags, char *name, caddr_t valuep, int *lengthp)
{
    minor_t instance;
    struct xxstate *xsp;
    if (dev != DDI_DEV_T_ANY) {
        return (ddi_prop_op(dev, dip, prop_op, flags, name,
            valuep, lengthp));
    }

    instance = getminor(dev);
    xsp = ddi_get_soft_state(statep, instance);
    if (xsp == NULL)
        return (DDI_PROP_NOTFOUND);
    if (strcmp(name, "temperature") == 0) {
        ddi_prop_update_int(dev, dip, name, temperature);
    }
    /* other cases */
}
```
Managing Events and Queueing Tasks

Drivers use events to respond to state changes. This chapter provides the following information on events:

- "Introduction to Events" on page 83
- "Using ddi_log_sysevent() to Log Events" on page 85
- "Defining Event Attributes" on page 86

Drivers use task queues to manage resource dependencies between tasks. This chapter provides the following information about task queues:

- "Introduction to Task Queues" on page 89
- "Task Queue Interfaces" on page 90
- "Observing Task Queues" on page 90

Managing Events

A system often needs to respond to a condition change such as a user action or system request. For example, a device might issue a warning when a component begins to overheat, or might start a movie player when a DVD is inserted into a drive. Device drivers can use a special message called an event to inform the system that a change in state has taken place.

Introduction to Events

An event is a message that a device driver sends to interested entities to indicate that a change of state has taken place. Events are implemented in the Oracle Solaris OS as user-defined, name-value pair structures that are managed using the nvlist* functions. (See the nvlist_alloc(9F) man page. Events are organized by vendor, class, and subclass. For example, you could define a class for monitoring environmental conditions. An environmental class could have subclasses to indicate changes in temperature, fan status, and power.
When a change in state occurs, the device notifies the driver. The driver then uses the `ddi_log_sysevent(9F)` function to log this event in a queue called `sysevent`. The `sysevent` queue passes events to the user level for handling by either the `syseventd` daemon or `syseventconfd` daemon. These daemons send notifications to any applications that have subscribed for notification of the specified event.

Two methods for designers of user-level applications deal with events:

- An application can use the routines in `libsysevent(3LIB)` to subscribe with the `syseventd` daemon for notification when a specific event occurs.
- A developer can write a separate user-level application to respond to an event. This type of application needs to be registered with `syseventadm(1M)`. When `syseventconfd` encounters the specified event, the application is run and deals with the event accordingly.

This process is illustrated in the following figure.

**FIGURE 5-1  Event Plumbing**
Using `ddi_log_sysevent()` to Log Events

Device drivers use the `ddi_log_sysevent(9F)` interface to generate and log events with the system.

**ddi_log_sysevent() Syntax**

`ddi_log_sysevent()` uses the following syntax:

```c
int ddi_log_sysevent(dev_info_t *dip, char *vendor, char *class,
                      char *subclass, nvlist_t *attr-list, sysevent_id_t *eidp,
                      int sleep-flag);
```

where:

- **dip** A pointer to the `dev_info` node for this driver.
- **vendor** A pointer to a string that defines the driver's vendor. Third-party drivers should use their company's stock symbol or a similarly enduring identifier. Oracle-supplied drivers use `DDI_VENDOR_SUNW`.
- **class** A pointer to a string defining the event's class. `class` is a driver-specific value. An example of a class might be a string that represents a set of environmental conditions that affect a device. This value must be understood by the event consumer.
- **subclass** A driver-specific string that represents a subset of the `class` argument. For example, within a class that represents environmental conditions, an event subclass might refer to the device's temperature. This value must be intelligible to the event consumer.
- **attr-list** A pointer to an `nvlist_t` structure that lists name-value attributes associated with the event. Name-value attributes are driver-defined and can refer to a specific attribute or condition of the device.

For example, consider a device that reads both CD-ROMs and DVDs. That device could have an attribute with the name `disc_type` and the value equal to either `cd_rom` or `dvd`.

As with `class` and `subclass`, an event consumer must be able to interpret the name-value pairs.

For more information on name-value pairs and the `nvlist_t` structure, see "Defining Event Attributes" on page 86, as well as the `nvlist_alloc(9F)` man page.

If the event has no attributes, then this argument should be set to `NULL`. 
The address of a sysevent_id_t structure. The sysevent_id_t structure is used to provide a unique identification for the event. ddi_log_sysevent(9F) returns this structure with a system-provided event sequence number and time stamp. See the ddi_log_sysevent(9F) man page for more information on the sysevent_id_t structure.

A flag that indicates how the caller wants to handle the possibility of resources not being available. If sleep-flag is set to DDI_SLEEP, the driver blocks until the resources become available. With DDI_NOSLEEP, an allocation will not sleep and cannot be guaranteed to succeed. If DDI_ENOMEM is returned, the driver would need to retry the operation at a later time.

Even with DDI_SLEEP, other error returns are possible with this interface, such as system busy, the syseventd daemon not responding, or trying to log an event in interrupt context.

Sample Code for Logging Events

A device driver performs the following tasks to log events:

- Allocate memory for the attribute list using nvlist_alloc(9F)
- Add name-value pairs to the attribute list
- Use the ddi_log_sysevent(9F) function to log the event in the sysevent queue
- Call nvlist_free(9F) when the attribute list is no longer needed

The following example demonstrates how to use ddi_log_sysevent()...

```c
EXAMPLE5-1 Calling ddi_log_sysevent()

char *vendor_name = "DDI_VENDOR_JGJG"
char *my_class = "JGJG_event";
char *my_subclass = "JGJG_alert";

nvlist_t *nvl;
/* ... */
nvlist_alloc(&nvl, nvflag, kmflag);
/* ... */
(void) nvlist_add_byte_array(nvl, propname, (uchar_t *)propval, proplen + 1);
/* ... */
if (ddi_log_sysevent(dip, vendor_name, my_class,
                   my_subclass, nvl, NULL, DDI_SLEEP) != DDI_SUCCESS)
    cmn_err(CE_WARN, "error logging system event");
nvlist_free(nvl);
```

Defining Event Attributes

Event attributes are defined as a list of name-value pairs. The Oracle Solaris DDI provides routines and structures for storing information in name-value pairs. Name-value pairs are retained in an nvlist_t structure, which is opaque to the driver. The value for a name-value
pair can be a Boolean, an int, a byte, a string, an nvlist, or an array of these data types. An int can be defined as 16 bits, 32 bits, or 64 bits and can be signed or unsigned.

The steps in creating a list of name-value pairs are as follows.

1. Create an `nvlist_t` structure with `nvlist_alloc(9F)`.

   The `nvlist_alloc()` interface takes three arguments:
   - `nvlp` – Pointer to a pointer to an `nvlist_t` structure
   - `nvflag` – Flag to indicate the uniqueness of the names of the pairs. If this flag is set to `NV_UNIQUE_NAME_TYPE`, any existing pair that matches the name and type of a new pair is removed from the list. If the flag is set to `NV_UNIQUE_NAME`, then any existing pair with a duplicate name is removed, regardless of its type. Specifying `NV_UNIQUE_NAME_TYPE` allows a list to contain two or more pairs with the same name as long as their types are different, whereas with `NV_UNIQUE_NAME` only one instance of a pair name can be in the list. If the flag is not set, then no uniqueness checking is done and the consumer of the list is responsible for dealing with duplicates.
   - `kmflag` – Flag to indicate the allocation policy for kernel memory. If this argument is set to `KM_SLEEP`, then the driver blocks until the requested memory is available for allocation. `KM_SLEEP` allocations might sleep but are guaranteed to succeed. `KM_NOSLEEP` allocations are guaranteed not to sleep but might return NULL if no memory is currently available.

2. Populate the `nvlist` with name-value pairs. For example, to add a string, use `nvlist_add_string(9F)`. To add an array of 32-bit integers, use `nvlist_add_int32_array(9F)`. The `nvlist_add_boolean(9F)` man page contains a complete list of interfaces for adding pairs.

To deallocate a list, use `nvlist_free(9F)`.

The following code sample illustrates the creation of a name-value list.

**EXAMPLE 5-2  Creating and Populating a Name-Value Pair List**

```c
nvlist_t* create_nvlist()
{
    int err;
    char *str = "child";
    int32_t ints[] = {0, 1, 2};
    nvlist_t *nvl;

    err = nvlist_alloc(&nvl, NV_UNIQUE_NAME, 0); /* allocate list */
    if (err)
        return (NULL);
    if ((nvlist_add_string(nvl, "name", str) != 0) ||
        (nvlist_add_int32_array(nvl, "prop", ints, 3) != 0)) {
        nvlist_free(nvl);
        return (NULL);
    }
}```
EXAMPLE 5–2 Creating and Populating a Name-Value Pair List (Continued)

```c
return (nvl);
}
```

Drivers can retrieve the elements of an `nvlist` by using a lookup function for that type, such as `nvlist_lookup_int32_array(9F)`, which takes as an argument the name of the pair to be searched for.

**Note** – These interfaces work only if either `NV_UNIQUE_NAME` or `NV_UNIQUE_NAME_TYPE` is specified when `nvlist_alloc(9F)` is called. Otherwise, ENOTSUP is returned, because the list cannot contain multiple pairs with the same name.

A list of name-value list pairs can be placed in contiguous memory. This approach is useful for passing the list to an entity that has subscribed for notification. The first step is to get the size of the memory block that is needed for the list with `nvlist_size(9F)`. The next step is to pack the list into the buffer with `nvlist_pack(9F)`. The consumer receiving the buffer's content can unpack the buffer with `nvlist_unpack(9F)`. The functions for manipulating name-value pairs are available to both user-level and kernel-level developers. You can find identical man pages for these functions in both *man pages section 3: Library Interfaces and Headers* and in *man pages section 9: DDI and DKI Kernel Functions*. For a list of functions that operate on name-value pairs, see the following table.

**TABLE 5–1 Functions for Using Name-Value Pairs**

<table>
<thead>
<tr>
<th>Man Page</th>
<th>Purpose / Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>nvlist_add_boolean(9F)</td>
<td>Add name-value pairs to the list. Functions include: nvlist_add_boolean(), nvlist_add_boolean_value(), nvlist_add_byte(), nvlist_add_int8(), nvlist_add_uint8(), nvlist_add_int16(), nvlist_add_uint16(), nvlist_add_int32(), nvlist_add_uint32(), nvlist_add_int64(), nvlist_add_uint64(), nvlist_add_string(), nvlist_add_nvlist(), nvlist_add_nvpair(), nvlist_add_boolean_array(), nvlist_add_int8_array, nvlist_add_uint8_array(), nvlist_add_nvlist_array(), nvlist_add_byte_array(), nvlist_add_int16_array(), nvlist_add_uint16_array(), nvlist_add_int32_array(), nvlist_add_uint32_array(), nvlist_add_int64_array(), nvlist_add_uint64_array(), nvlist_add_string_array()</td>
</tr>
<tr>
<td>nvlist_alloc(9F)</td>
<td>Manipulate the name-value list buffer. Functions include: nvlist_alloc(), nvlist_free(), nvlist_size(), nvlist_pack(), nvlist_unpack(), nvlist_dup(), nvlist_merge()</td>
</tr>
</tbody>
</table>
Queueing Tasks

This section discusses how to use task queues to postpone processing of some tasks and delegate their execution to another kernel thread.

Introduction to Task Queues

A common operation in kernel programming is to schedule a task to be performed at a later time, by a different thread. The following examples give some reasons that you might want a different thread to perform a task at a later time:

- Your current code path is time critical. The additional task you want to perform is not time critical.
- The additional task might require grabbing a lock that another thread is currently holding.
- You cannot block in your current context. The additional task might need to block, for example to wait for memory.
- A condition is preventing your code path from completing, but your current code path cannot sleep or fail. You need to queue the current task to execute after the condition disappears.
- You need to launch multiple tasks in parallel.

### TABLE 5–1 Functions for Using Name-Value Pairs  (Continued)

<table>
<thead>
<tr>
<th>Man Page</th>
<th>Purpose / Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>nvlist_lookup_boolean(9F)</td>
<td>Search for name-value pairs. Functions include: nvlist_lookup_boolean(), nvlist_lookup_boolean_value(), nvlist_lookup_byte(), nvlist_lookup_int8(), nvlist_lookup_int16(), nvlist_lookup_int32(), nvlist_lookup_int64(), nvlist_lookup_uint8(), nvlist_lookup_uint16(), nvlist_lookup_uint32(), nvlist_lookup_uint64(), nvlist_lookup_string(), nvlist_lookup_nvlist(), nvlist_lookup_boolean_array, nvlist_lookup_byte_array(), nvlist_lookup_int8_array(), nvlist_lookup_int16_array(), nvlist_lookup_int32_array(), nvlist_lookup_int64_array(), nvlist_lookup_uint8_array(), nvlist_lookup_uint16_array(), nvlist_lookup_uint32_array(), nvlist_lookup_uint64_array(), nvlist_lookup_string_array(), nvlist_lookup_nvlist_array(), nvlist_lookup_pairs()</td>
</tr>
<tr>
<td>nvlist_next_nvpair(9F)</td>
<td>Get name-value pair data. Functions include: nvlist_next_nvpair(), nvpair_name(), nvpair_type()</td>
</tr>
<tr>
<td>nvlist_remove(9F)</td>
<td>Remove name-value pairs. Functions include: nv_remove(), nv_remove_all()</td>
</tr>
</tbody>
</table>
In each of these cases, a task is executed in a different context. A different context is usually a different kernel thread with a different set of locks held and possibly a different priority. Task queues provide a generic kernel API for scheduling asynchronous tasks.

A task queue is a list of tasks with one or more threads to service the list. If a task queue has a single service thread, all tasks are guaranteed to execute in the order in which they are added to the list. If a task queue has more than one service thread, the order in which the tasks will execute is not known.

**Note** – If the task queue has more than one service thread, make sure that the execution of one task does not depend on the execution of any other task. Dependencies between tasks can cause a deadlock to occur.

**Task Queue Interfaces**

The following DDI interfaces manage task queues. These interfaces are defined in the `sys/sunddi.h` header file. See the `taskq(9F)` man page for more information about these interfaces.

- `ddi_taskq_t` : Opaque handle
- `TASKQ_DEFAULTPRI` : System default priority
- `DDI_SLEEP` : Can block for memory
- `DDI_NOSLEEP` : Cannot block for memory
- `ddi_taskq_create()` : Create a task queue
- `ddi_taskq_destroy()` : Destroy a task queue
- `ddi_taskq_dispatch()` : Add a task to a task queue
- `ddi_taskq_wait()` : Wait for pending tasks to complete
- `ddi_taskq_suspend()` : Suspend a task queue
- `ddi_taskq_suspended()` : Check whether a task queue is suspended
- `ddi_taskq_resume()` : Resume a suspended task queue

**Observing Task Queues**

The typical usage in drivers is to create task queues at `attach(9E)`. Most `taskq_dispatch()` invocations are from interrupt context.
This section describes two techniques that you can use to monitor the system resources that are consumed by a task queue. Task queues export statistics on the use of system time by task queue threads. Task queues also use DTrace SDT probes to determine when a task queue starts and finishes execution of a task.

**Task Queue Kernel Statistics Counters**

Every task queue has an associated set of kstat counters. Examine the output of the following `kstat(1M)` command:

```
$ kstat -c taskq
module: unix instance: 0
name: ata_nexus_enum_tq  class:   taskq
crt ime  53.877907833
executed  0
maxtasks  0
nactive  1
nalloc  0
priority  60
snaptime  258059.249256749
tasks  0
threads  1
totaltime  0
```

The `kstat` output shown above includes the following information:

- The name of the task queue and its instance number
- The number of scheduled (tasks) and executed (executed) tasks
- The number of kernel threads processing the task queue (threads) and their priority (priority)
- The total time (in nanoseconds) spent processing all the tasks (totaltime)

The following example shows how you can use the `kstat` command to observe how a counter (number of scheduled tasks) increases over time:

```
$ kstat -p unix:0:callout_taskq:tasks 1 5
unix:0:callout_taskq:tasks  13994642
unix:0:callout_taskq:tasks  13994711
```

```
Queueing Tasks
Chapter 5 • Managing Events and Queueing Tasks 91
```
Task Queue DTrace SDT Probes

Task queues provide several useful SDT probes. All the probes described in this section have the following two arguments:

- The task queue pointer returned by `ddi_taskq_create()`
- The pointer to the `taskq_ent_t` structure. Use this pointer in your D script to extract the function and the argument.

You can use these probes to collect precise timing information about individual task queues and individual tasks being executed through them. For example, the following script prints the functions that were scheduled through task queues for every 10 seconds:

```d
#!/usr/sbin/dtrace -qs
sdt:genunix::taskq.enqueue
{ this->tq = (taskq_t *) arg0; this->tqe = (taskq_ent_t *) arg1; @[this->tq->tq_name, this->tq->tq_instance, this->tqe->tqent_func] = count(); }
tick-10s
{ printa ("%s(%d): %a called %@d times", @); trunc(@); }
```

On a particular machine, the above D script produced the following output:

```
callout_taskq(1): genunix'callout_execute called 51 times
callout_taskq(0): genunix'callout_execute called 701 times
kmem_taskq(0): genunix'kmem_update_timeout called 1 times
kmem_taskq(0): genunix'kmem_hash_rescale called 4 times
USB_hid_81_pipelnq_1(14): usba'hcdi cb_thread called 256 times
callout_taskq(0): genunix'callout_execute called 702 times
kmem_taskq(0): genunix'kmem_update_timeout called 1 times
kmem_taskq(0): genunix'kmem_hash_rescale called 4 times
callout_taskq(1): genunix'callout_execute called 28 times
USB_hid_81_pipelnq_1(14): usba'hcdi cb_thread called 228 times
callout_taskq(0): genunix'callout_execute called 706 times
callout_taskq(1): genunix'callout_execute called 141 times
USB_hid_81_pipelnq_1(14): usba'hcdi cb_thread called 141 times
callout_taskq(0): genunix'callout_execute called 708 times
```
Autoconfiguration means the driver loads code and static data into memory. This information is then registered with the system. Autoconfiguration also involves attaching individual device instances that are controlled by the driver.

This chapter provides information on the following subjects:
- “Driver Loading and Unloading” on page 93
- “Data Structures Required for Drivers” on page 94
- “Loadable Driver Interfaces” on page 97
- “Device Configuration Concepts” on page 100
- “Using Device IDs” on page 112

### Driver Loading and Unloading

The system loads driver binary modules from the drv subdirectory of the kernel module directory for autoconfiguration. See “Copying the Driver to a Module Directory” on page 493.

After a module is read into memory with all symbols resolved, the system calls the _init(9E) entry point for that module. The _init() function calls mod_install(9F), which actually loads the module.

**Note** – During the call to mod_install(), other threads are able to call attach(9E) as soon as mod_install() is called. From a programming standpoint, all _init() initialization must occur before mod_install() is called. If mod_install() fails (that is a nonzero value is returned), then the initialization must be backed out.

Upon successful completion of _init(), the driver is properly registered with the system. At this point, the driver is not actively managing any device. Device management happens as part of device configuration.
The system unloads driver binary modules either to conserve system memory or at the explicit request of a user. Before deleting the driver code and data from memory, the `_fini(9E)` entry point of the driver is invoked. The driver is unloaded, if and only if `_fini()` returns success.

The following figure provides a structural overview of a device driver. The shaded area highlights the driver data structures and entry points. The upper half of the shaded area contains data structures and entry points that support driver loading and unloading. The lower half is concerned with driver configuration.

![Module Loading and Autoconfiguration Entry Points](image)

**Data Structures Required for Drivers**

To support autoconfiguration, drivers are required to statically initialize the following data structures:

- `modlinkage(9S)`
- `modldrv(9S)`
- `dev_ops(9S)`
- `cb_ops(9S)`

The data structures in Figure 5-1 are relied on by the driver. These structures must be provided and be initialized correctly. Without these data structures, the driver might not load properly. As a result, the necessary routines might not be loaded. If an operation is not supported by the driver, the address of the `nodev(9F)` routine can be used as a placeholder. In some instances, the driver supports the entry point and only needs to return success or failure. In such cases, the address of the routine `nulldev(9F)` can be used.
Note - These structures should be initialized at compile-time. The driver should not access or change the structures at any other time.

**modlinkage Structure**

```c
static struct modlinkage xxmodlinkage = {
    MODREV_1, /* ml_rev */
    &xxmodldrv, /* ml_linkage[] */
    NULL /* NULL termination */
};
```

The first field is the version number of the module that loads the subsystem. This field should be MODREV_1. The second field points to driver's modldrv structure defined next. The last element of the structure should always be NULL.

**modldrv Structure**

```c
static struct modldrv xxmodldrv = {
    &mod_driverops, /* drv_modops */
    "generic driver v1.1", /* drv_linkinfo */
    &xx_dev_ops /* drv_dev_ops */
};
```

This structure describes the module in more detail. The first field provides information regarding installation of the module. This field should be set to &mod_driverops for driver modules. The second field is a string to be displayed by `modinfo(1M)`. The second field should contain sufficient information for identifying the version of source code that generated the driver binary. The last field points to the driver's dev_ops structure defined in the following section.

**dev_ops Structure**

```c
static struct dev_ops xx_dev_ops = {
    DEVO_REV, /* devo_rev */
    0, /* devo_refcnt */
    xxgetinfo, /* devo_getinfo: getinfo(9E) */
    nulldev, /* devo_identify: identify(9E) */
    xxprobes, /* devo_probe: probe(9E) */
    xxattach, /* devo_attach: attach(9E) */
    xxdetach, /* devo_detach: detach(9E) */
    nodedev, /* devo_reset: see devo_quiesce */
    &xx_cb_ops, /* devo_cb_ops */
    NULL, /* devo_bus_ops */
    &xxpower, /* devo_power: power(9E) */
    ddi_quiesce_not_needed, /* devo_quiesce: quiesce(9E) */
};
```
The `dev_ops(9S)` structure enables the kernel to find the autoconfiguration entry points of the device driver. The `devo_rev` field identifies the revision number of the structure. This field must be set to `DEVO_REV`. The `devo_refcnt` field must be initialized to zero. The function address fields should be filled in with the address of the appropriate driver entry point, except in the following cases:

- Set the `devo_identify` field to `nulldev(9F)`. The `identify()` entry point is obsolete.
- Set the `devo_probe` field to `nulldev(9F)` if a `probe(9E)` routine is not needed.
- Set the `devo_reset` field to `nodev(9F)`. The `nodev()` function returns `ENXIO`. See `devo_quiesce`.
- Set the `devo_power` field to `NULL` if a `power()` routine is not needed. Drivers for devices that provide Power Management functionality must have a `power(9E)` entry point. See Chapter 12, “Power Management.”
- Set the `devo_quiesce` field to `ddi_quiesce_not_needed()` if the driver does not need to implement `quiesce`. Drivers that manage devices must provide a `quiesce(9E)` entry point.

The `devo_cb_ops` member should include the address of the `cb_ops(9S)` structure. The `devo_bus_ops` field must be set to `NULL`.

**cb_ops Structure**

```c
static struct cb_ops xx_cb_ops = {
    xxopen,  /* open(9E) */
    xxclose, /* close(9E) */
    xxstrategy, /* strategy(9E) */
    xxprint, /* print(9E) */
    xxdump, /* dump(9E) */
    xxread, /* read(9E) */
    xxwrite, /* write(9E) */
    xxioctl, /* ioctl(9E) */
    xxdevmap, /* devmap(9E) */
    nodev, /* devmap(9E) */
    xxsegmap, /* segmap(9E) */
    xxchpoll, /* chpoll(9E) */
    xxprop_op, /* prop_op(9E) */
    NULL, /* streamtab(9S) */
    D_MP | D_64BIT, /* cb_flag */
    CB_REV, /* cb_rev */
    xxaread, /* aread(9E) */
    xxawrite /* awrite(9E) */
};
```

The `cb_ops(9S)` structure contains the entry points for the character operations and block operations of the device driver. Any entry points that the driver does not support should be initialized to `nodev(9F)`. For example, character device drivers should set all the block-only fields, such as `cb_strategy`, to `nodev(9F)`. Note that the `mmap(9E)` entry point is maintained for compatibility with previous releases. Drivers should use the `devmap(9E)` entry point for device memory mapping. If `devmap(9E)` is supported, set `mmap(9E)` to `nodev(9F)`.
The streamtab field indicates whether the driver is STREAMS-based. Only the network device drivers that are discussed in Chapter 19, “Drivers for Network Devices,” are STREAMS-based. All non-STREAMS-based drivers must set the streamtab field to NULL.

The cb_flag member contains the following flags:

- The D_MP flag indicates that the driver is safe for multithreading. The Oracle Solaris OS supports only thread-safe drivers so D_MP must be set.
- The D_64BIT flag causes the driver to use the uio_loffset field of the uio(9S) structure. The driver should set the D_64BIT flag in the cb_flag field to handle 64-bit offsets properly.
- The D_DEVMAP flag supports the devmap(9E) entry point. For information on devmap(9E), see Chapter 10, “Mapping Device and Kernel Memory.”

cb_rev is the cb_ops structure revision number. This field must be set to CB_REV.

## Loadable Driver Interfaces

Device drivers must be dynamically loadable. Drivers should also be unloadable to help conserve memory resources. Drivers that can be unloaded are also easier to test, debug, and patch.

Each device driver is required to implement _init(9E), _fini(9E), and _info(9E) entry points to support driver loading and unloading. The following example shows a typical implementation of loadable driver interfaces.

### EXAMPLE 6-1  Loadable Interface Section

```c
static void *statep; /* for soft state routines */
static struct cb_ops xx_cb_ops; /* forward reference */
static struct dev_ops xx_ops = {
    DEVO_REV,
    0,
    xxgetinfo,
    nulldev,
    xxprobe,
    xxattach,
    xxdetach,
    xxreset,
    nodev,
    &xx_cb_ops,
    NULL,
    xxpower,
    ddi_quiesce_not_needed,
};

static struct modldr modldr = {
    &mod_driverops,
    "xx driver v1.0",
    &xx_ops
};
```
static struct modlinkage modlinkage = {
    MODREV_1,
    &modldrv,
    NULL
};

int _init(void)
{
    int error;
    ddi_soft_state_init(&statep, sizeof (struct xxstate),
                        estimated_number_of_instances);
    /* further per-module initialization if necessary */
    error = mod_install(&modlinkage);
    if (error != 0) {
        /* undo any per-module initialization done earlier */
        ddi_soft_state_fini(&statep);
    }
    return (error);
}

int _fini(void)
{
    int error;
    if (error == 0) {
        /* release per-module resources if any were allocated */
        ddi_soft_state_fini(&statep);
    }
    return (error);
}

int _info(struct modinfo *modinfop)
{
    return (mod_info(&modlinkage, modinfop));
}

_init() Example

The following example shows a typical _init(9E) interface.

 EXAMPLE 6-2 _init() Function
static void **xxstatep;
int _init(void)
{
    int error;
    const int max_instance = 20; /* estimated max device instances */
ddi_soft_state_init(&xxstatep, sizeof (struct xxstate), max_instance);
error = mod_install(&xxmodlinkage);
if (error != 0) {
    /*
    * Cleanup after a failure
    */
    ddi_soft_state_fini(&xxstatep);
}  
return (error);

The driver should perform any one-time resource allocation or data initialization during driver loading in _init(). For example, the driver should initialize any mutexes global to the driver in this routine. The driver should not, however, use _init(9E) to allocate or initialize anything that has to do with a particular instance of the device. Per-instance initialization must be done in attach(9E). For example, if a driver for a printer can handle more than one printer at the same time, that driver should allocate resources specific to each printer instance in attach().

Note - Once _init(9E) has called mod_install(9F), the driver should not change any of the data structures attached to the modlinkage(9S) structure because the system might make copies or change the data structures.

_fini() Example

The following example demonstrates the _fini() routine.

```c
int  _fini(void) {  
    int error;
    error = mod_remove(&modlinkage);
    if (error != 0) {  
        return (error);
    }  
    /*
    * Cleanup resources allocated in _init()
    */
    ddi_soft_state_fini(&xxstatep);
    return (0);
}
```

Similarly, in _fini(), the driver should release any resources that were allocated in _init(). The driver must remove itself from the system module list.
Note — \texttt{fini()} might be called when the driver is attached to hardware instances. In this case, \texttt{mod\_remove(9F)} returns failure. Therefore, driver resources should not be released until \texttt{mod\_remove()} returns success.

\textbf{\_info()} Example

The following example demonstrates the \texttt{\_info(9E)} routine.

\begin{verbatim}
int \_info(struct modinfo \*modinfop)
{
    return (mod\_info(&xxmod\_linkage, modinfop));
}
\end{verbatim}

The driver is called to return module information. The entry point should be implemented as shown above.

\section*{Device Configuration Concepts}

For each node in the kernel device tree, the system selects a driver for the node based on the node name and the \texttt{compatible} property (see "Binding a Driver to a Device" on page 62). The same driver might bind to multiple device nodes. The driver can differentiate different nodes by instance numbers assigned by the system.

After a driver is selected for a device node, the driver’s \texttt{probe(9E)} entry point is called to determine the presence of the device on the system. If \texttt{probe()} is successful, the driver’s \texttt{attach(9E)} entry point is invoked to set up and manage the device. The device can be opened if and only if \texttt{attach()} returns success (see "attach() Entry Point" on page 104).

A device might be unconfigured to conserve system memory resources or to enable the device to be removed while the system is still running. To enable the device to be unconfigured, the system first checks whether the device instance is referenced. This check involves calling the driver’s \texttt{getinfo(9E)} entry point to obtain information known only to the driver (see "getinfo() Entry Point" on page 110). If the device instance is not referenced, the driver’s \texttt{detach(9E)} routine is invoked to unconfigure the device (see "detach() Entry Point" on page 109).

To recap, each driver must define the following entry points that are used by the kernel for device configuration:

- \texttt{probe(9E)}
- \texttt{attach(9E)}
- \texttt{detach(9E)}
Note that attach(), detach(), and getinfo() are required. probe() is only required for devices that cannot identify themselves. For self-identifying devices, an explicit probe() routine can be provided, or nulldev(9F) can be specified in the dev_ops structure for the probe() entry point.

Device Instances and Instance Numbers

The system assigns an instance number to each device. The driver might not reliably predict the value of the instance number assigned to a particular device. The driver should retrieve the particular instance number that has been assigned by calling ddi_get_instance(9F).

Instance numbers represent the system’s notion of devices. Each dev_info, that is, each node in the device tree, for a particular driver is assigned an instance number by the kernel. Furthermore, instance numbers provide a convenient mechanism for indexing data specific to a particular physical device. The most common use of instance numbers is ddi_get_soft_state(9F), which uses instance numbers to retrieve soft state data for specific physical devices.

Caution – For pseudo devices, that is, the children of pseudo nexuses, the instance numbers are defined in the driver.conf(4) file using the instance property. If the driver.conf file does not contain the instance property, the behavior is undefined. For hardware device nodes, the system assigns instance numbers when the device is first seen by the OS. The instance numbers persist across system reboots and OS upgrades.

Minor Nodes and Minor Numbers

Drivers are responsible for managing their minor number namespace. For example, the sd driver needs to export eight character minor nodes and eight block minor nodes to the file system for each disk. Each minor node represents either a block interface or a character interface to a portion of the disk. The getinfo(9E) entry point informs the system about the mapping from minor number to device instance (see “getinfo() Entry Point” on page 110).

probe() Entry Point

For non-self-identifying devices, the probe(9E) entry point should determine whether the hardware device is present on the system.

For probe() to determine whether the instance of the device is present, probe() needs to perform many tasks that are also commonly done by attach(9E). In particular, probe() might need to map the device registers.
Probing the device registers is device-specific. The driver often has to perform a series of tests of the hardware to assure that the hardware is really present. The test criteria must be rigorous enough to avoid misidentifying devices. For example, a device might appear to be present when in fact that device is not available, because a different device seems to behave like the expected device.

The test returns the following flags:

- **DDI_PROBE_SUCCESS** if the probe was successful
- **DDI_PROBE_FAILURE** if the probe failed
- **DDI_PROBE_DONTCARE** if the probe was unsuccessful yet `attach(9E)` still needs to be called
- **DDI_PROBE_PARTIAL** if the instance is not present now, but might be present in the future

For a given device instance, `attach(9E)` will not be called until `probe(9E)` has succeeded at least once on that device.

`probe(9E)` must free all the resources that `probe()` has allocated, because `probe()` might be called multiple times. However, `attach(9E)` is not necessarily called even if `probe(9E)` has succeeded.

`ddi_dev_is_sid(9F)` can be used in a driver’s `probe(9E)` routine to determine whether the device is self-identifying. `ddi_dev_is_sid()` is useful in drivers written for self-identifying and non-self-identifying versions of the same device.

The following example is a sample `probe()` routine.

```c
EXAMPLE 6-3 probe(9E) Routine

static int
xxprobe(dev_info_t *dip)
{
    ddi_acc_handle_t dev_hdl;
    ddi_device_acc_attr_t dev_attr;
    Pio_csr *csrp;
    uint8_t csrval;

    /*
     * if the device is self identifying, no need to probe
     */
    if (ddi_dev_is_sid(dip) == DDI_SUCCESS)
        return (DDI_PROBE_DONTCARE);

    /*
     * Initialize the device access attributes and map in
     * the devices CSR register (register 0)
     */
    dev_attr.devacc_attr_version = DDI_DEVICE_ATTR_V0;
    dev_attr.devacc_attr_endian_flags = DDI_STRUCTURE_LE_ACC;
    dev_attr.devacc_attr_dataorder = DDI_STRICTORDER_ACC;

    if (ddi_regs_map_setup(dip, 0, (caddr_t *)&csrp, 0, sizeof (Pio_csr),
```
When the driver's \texttt{probe(9E)} routine is called, the driver does not know whether the device being probed exists on the bus. Therefore, the driver might attempt to access device registers for a nonexistent device. A bus fault might be generated on some buses as a result.

The following example shows a \texttt{probe(9E)} routine that uses \texttt{ddi_poke8(9F)} to check for the existence of the device. \texttt{ddi_poke8()} cautiously attempts to write a value to a specified virtual address, using the parent nexus driver to assist in the process where necessary. If the address is not valid or the value cannot be written without an error occurring, an error code is returned. See also \texttt{ddi.peek(9F)}.

In this example, \texttt{ddi_regs_map_setup(9F)} is used to map the device registers.

\begin{verbatim}
EXAMPLE 6-4  probe(9E) Routine Using ddi_poke8(9F)

static int
xxprobe(dev_info_t *dip)
{
   ddi_acc_handle_t dev_hdl;
   ddi_device_acc_attr_t dev_attr;
   Pio_csr *csrp;
   uint8_t csrval;

   /*
   * if the device is self-identifying, no need to probe
   */
   if (ddi_dev_is_sid(dip) == DDI_SUCCESS)
      return (DDI_PROBE_DONTCARE);

   /*
   * Initialize the device access attributes and map in
   * the device's CSR register (register 0)
   */

   dev_attr = ddi_device_access_attr(dip);
   dev_hdl = ddi_open(dip, &dev_attr);

ddi_regs_map_setup(dev_hdl, dev_attr);

   return (DDI_PROBE_DONTCARE);
}
\end{verbatim}
EXAMPLE 6-4 probe(9E) Routine Using ddi_poke8(9F) (Continued)

    */
    dev_attr.devacc_attr_version = DDI_DEVICE_ATTR_V0;
    dev_attr.devacc_attr_endian_flags = DDI_STRUCTURE_LE_ACC;
    dev_attr.devacc_attr_dataorder = DDI_STRICTORDER_ACC;

    if (ddi_regs_map_setup(dip, 0, (caddr_t *)&csrp, 0, sizeof (Pio_csr),
                   &dev_attr, &dev_hdl) != DDI_SUCCESS)
        return (DDI_PROBE_FAILURE);

    /*
     * The bus can generate a fault when probing for devices that
     * do not exist. Use ddi_poke8(9f) to handle any faults that
     * might occur.
     *
     * Reset the device. Once the reset completes the CSR should read
     * back (PIO_DEV_READY | PIO_IDLE_INTR)
     */
    if (ddi_poke8(dip, csrp, PIO_RESET) != DDI_SUCCESS) {
        ddi_regs_map_free(&dev_hdl);
        return (DDI_FAILURE);
    }

    csrval = ddi_get8(dev_hdl, csrp);

    /*
     * tear down the mappings and return probe success/failure
     */
    ddi_regs_map_free(&dev_hdl);
    if ((csrval & 0xff) == (PIO_DEV_READY | PIO_IDLE_INTR))
        return (DDI_PROBE_SUCCESS);
    else
        return (DDI_PROBE_FAILURE);
}

attach() Entry Point

The kernel calls a driver’s attach(9E) entry point to attach an instance of a device or to resume operation for an instance of a device that has been suspended or has been shut down by the power management framework. This section discusses only the operation of attaching device instances. Power management is discussed in Chapter 12, “Power Management.”

A driver’s attach(9E) entry point is called to attach each instance of a device that is bound to the driver. The entry point is called with the instance of the device node to attach, with DDI_ATTACH specified as the cmd argument to attach(9E). The attach entry point typically includes the following types of processing:

- Allocating a soft-state structure for the device instance
- Initializing per-instance mutexes
- Initializing condition variables
- Registering the device’s interrupts
- Mapping the registers and memory of the device instance
- Creating minor device nodes for the device instance
Reporting that the device instance has attached

**Driver Soft-State Management**

To assist device driver writers in allocating state structures, the Oracle Solaris DDI/DKI provides a set of memory management routines called *software state management routines*, which are also known as the *soft-state routines*. These routines dynamically allocate, retrieve, and destroy memory items of a specified size, and hide the details of list management. An *instance number* identifies the desired memory item. This number is typically the instance number assigned by the system.

Drivers typically allocate a soft-state structure for each device instance that attaches to the driver by calling `ddi_soft_state_zalloc(9F)`, passing the instance number of the device. Because no two device nodes can have the same instance number, `ddi_soft_state_zalloc(9F)` fails if an allocation already exists for a given instance number.

A driver’s character or block entry point (`cb_ops(9S)`) references a particular soft state structure by first decoding the device’s instance number from the `dev_t` argument that is passed to the entry point function. The driver then calls `ddi_get_soft_state(9F)`, passing the per-driver soft-state list and the instance number that was derived. A NULL return value indicates that effectively the device does not exist and the appropriate code should be returned by the driver.

See “Creating Minor Device Nodes” on page 105 for additional information on how instance numbers and device numbers, or `dev_t`s, are related.

**Lock Variable and Conditional Variable Initialization**

Drivers should initialize any per-instance locks and condition variables during attach. The initialization of any locks that are acquired by the driver’s interrupt handler must be initialized prior to adding any interrupt handlers. See Chapter 3, “Multithreading,” for a description of lock initialization and usage. See Chapter 8, “Interrupt Handlers,” for a discussion of interrupt handler and lock issues.

**Creating Minor Device Nodes**

An important part of the attach process is the creation of *minor nodes* for the device instance. A minor node contains the information exported by the device and the DDI framework. The system uses this information to create a *special file* for the minor node under `/devices`.

Minor nodes are created when the driver calls `ddi_create_minor_node(9F)`. The driver supplies a *minor number*, a *minor name*, a *minor node type*, and whether the minor node represents a block or character device.

Drivers can create any number of minor nodes for a device. The Oracle Solaris DDI/DKI expects certain classes of devices to have minor nodes created in a particular format. For example, disk drivers are expected to create 16 minor nodes for each physical disk instance attached. Eight minor nodes are created, representing the `a-h` block device interfaces, with an additional eight minor nodes for the `raw - h`, `raw` character device interfaces.
The minor number passed to `ddi_create_minor_node(9F)` is defined wholly by the driver. The minor number is usually an encoding of the instance number of the device with a minor node identifier. In the preceding example, the driver creates minor numbers for each of the minor nodes by shifting the instance number of the device left by three bits and using the OR of that result with the minor node index. The values of the minor node index range from 0 to 7. Note that minor nodes `a` and `a`, `raw` share the same minor number. These minor nodes are distinguished by the `spec_type` argument passed to `ddi_create_minor_node()`.

The minor node type passed to `ddi_create_minor_node(9F)` classifies the type of device, such as disks, tapes, network interfaces, frame buffers, and so forth.

The following table lists the types of possible nodes that might be created.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDI_NT_SERIAL</td>
<td>Serial port</td>
</tr>
<tr>
<td>DDI_NT_SERIAL_DO</td>
<td>Dialout ports</td>
</tr>
<tr>
<td>DDI_NT_BLOCK</td>
<td>Hard disks</td>
</tr>
<tr>
<td>DDI_NT_BLOCK_CHAN</td>
<td>Hard disks with channel or target numbers</td>
</tr>
<tr>
<td>DDI_NT_CD</td>
<td>ROM drives (CD-ROM)</td>
</tr>
<tr>
<td>DDI_NT_CD_CHAN</td>
<td>ROM drives with channel or target numbers</td>
</tr>
<tr>
<td>DDI_NT_FD</td>
<td>Floppy disks</td>
</tr>
<tr>
<td>DDI_NT_TAPE</td>
<td>Tape drives</td>
</tr>
<tr>
<td>DDI_NT_NET</td>
<td>Network devices</td>
</tr>
<tr>
<td>DDI_NT_DISPLAY</td>
<td>Display devices</td>
</tr>
<tr>
<td>DDI_NT_MOUSE</td>
<td>Mouse</td>
</tr>
<tr>
<td>DDI_NT_KEYBOARD</td>
<td>Keyboard</td>
</tr>
<tr>
<td>DDI_NT_AUDIO</td>
<td>Audio Device</td>
</tr>
<tr>
<td>DDI_PSEUDO</td>
<td>General pseudo devices</td>
</tr>
</tbody>
</table>

The node types `DDI_NT_BLOCK`, `DDI_NT_BLOCK_CHAN`, `DDI_NT_CD`, and `DDI_NT_CD_CHAN` cause `devfsadm(1M)` to identify the device instance as a disk and to create names in the `/dev/dsk` or `/dev/rdsk` directory.

The node type `DDI_NT_TAPE` causes `devfsadm(1M)` to identify the device instance as a tape and to create names in the `/dev/rmt` directory.
The node types DDI_NT_SERIAL and DDI_NT_SERIAL_DO cause devfsadm(1M) to perform these actions:

- Identify the device instance as a serial port
- Create names in the /dev/term directory
- Add entries to the /etc/inittab file

Vendor-supplied strings should include an identifying value such as a name or stock symbol to make the strings unique. The string can be used in conjunction with devfsadm(1M) and the devlinks.tab file (see the devlinks(1M) man page) to create logical names in /dev.

**Deferred Attach**

open(9E) might be called on a minor device before attach(9E) has succeeded on the corresponding instance. open() must then return ENXIO, which causes the system to attempt to attach the device. If the attach() succeeds, the open() is retried automatically.

**EXAMPLE 6-5 Typical attach() Entry Point**

```c
/*
 * Attach an instance of the driver. We take all the knowledge we
 * have about our board and check it against what has been filled in
 * for us from our FCode or from our driver.conf(4) file.
 */
static int
xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    int instance;
    Pio *pio_p;
    ddi_device_acc_attr_t da_attr;
    static int pio_validate_device(dev_info_t *);
    switch (cmd) {
        case DDI_ATTACH:
            /*
             * first validate the device conforms to a configuration this driver
             * supports
             */
            if (pio_validate_device(dip) == 0)
                return (DDI_FAILURE);
            /*
             * Allocate a soft state structure for this device instance
             * Store a pointer to the device node in our soft state structure
             * and a reference to the soft state structure in the device
             * node.
             */
            instance = ddi_get_instance(dip);
            if (ddi_soft_state_zalloc(pio_softstate, instance) != 0)
                return (DDI_FAILURE);
            pio_p = ddi_get_soft_state(pio_softstate, instance);
            ddi_set_driver_private(dip, (caddr_t)pio_p);
            pio_p->dip = dip;
            /*
             * Before adding the interrupt, get the interrupt block
             * cookie associated with the interrupt specification to
             */
```
EXAMPLE 6-5  Typical attach() Entry Point  (Continued)

  * initialize the mutex used by the interrupt handler.
  */
if (ddi_get_iblock_cookie(dip, 0, &pio_p->iblock_cookie) !=
    DDI_SUCCESS) {
    ddi_soft_state_free(pio_softstate, instance);
    return (DDI_FAILURE);
}
mutex_init(&pio_p->mutex, NULL, MUTEX_DRIVER, pio_p->iblock_cookie);
  /*
  * Now that the mutex is initialized, add the interrupt itself.
  */
if (ddi_add_intr(dip, 0, NULL, NULL, pio_intr, (caddr_t)instance) !=
    DDI_SUCCESS) {
    mutex_destroy(&pio_p->mutex);
    ddi_soft_state_free(pio_softstate, instance);
    return (DDI_FAILURE);
}
  /*
  * Initialize the device access attributes for the register mapping
  */
dev_acc_attr.devacc_attr_version = DDI_DEVICE_ATTR_V0;
dev_acc_attr.devacc_attr_endian_flags = DDI_STRUCTURE_LE_ACC;
dev_acc_attr.devacc_attr_dataorder = DDI_STRICTORDER_ACC;
  /*
  * Map in the csr register (register 0)
  */
if (ddi_regs_map_setup(dip, 0, (caddr_t *)&(pio_p->csr), 0,
    sizeof (Pio_csr), &dev_acc_attr, &pio_p->csr_handle) !=
    DDI_SUCCESS) {
    ddi_remove_intr(pio_p->dip, 0, pio_p->iblock_cookie);
    mutex_destroy(&pio_p->mutex);
    ddi_soft_state_free(pio_softstate, instance);
    return (DDI_FAILURE);
}
  /*
  * Map in the data register (register 1)
  */
if (ddi_regs_map_setup(dip, 1, (caddr_t *)&(pio_p->data), 0,
    sizeof (uchar_t), &dev_acc_attr, &pio_p->data_handle) !=
    DDI_SUCCESS) {
    ddi_remove_intr(pio_p->dip, 0, pio_p->iblock_cookie);
    mutex_destroy(&pio_p->mutex);
    ddi_soft_state_free(pio_softstate, instance);
    return (DDI_FAILURE);
}
  /*
  * Create an entry in /devices for user processes to open(2)
  * This driver will create a minor node entry in /devices
  * of the form: /devices/....pio@X,Y:pio
  */
if (ddi_create_minor_node(dip, ddi_get_name(dip), S_IFCHR,
    instance, DDI_PSEUDO, 0) == DDI_FAILURE) {
    ddi_remove_intr(pio_p->dip, 0, pio_p->iblock_cookie);
    ddi_regs_map_free(&pio_p->csr_handle);
EXAMPLE 6–5  Typical attach() Entry Point  (Continued)

```c
    ddi_regs_map_free(&pio_p->data_handle);
    mutex_destroy(&pio_p->mutex);
    ddi_soft_state_free(pio_softstate, instance);
    return (DDI_FAILURE);
}
/*
 * reset device (including disabling interrupts)
 */
    ddi_put8(pio_p->csr_handle, pio_p->csr, PIO_RESET);
/*
 * report the name of the device instance which has attached
 */
    ddi_report_dev(dip);
    return (DDI_SUCCESS);

    case DDI_RESUME:
    return (DDI_SUCCESS);

    default:
    return (DDI_FAILURE);
}
```

**Note** – The attach() routine must not make any assumptions about the order of invocations on different device instances. The system might invoke `attach()` concurrently on different device instances. The system might also invoke `attach()` and `detach()` concurrently on different device instances.

### detach() Entry Point

The kernel calls a driver’s `detach(9E)` entry point to detach an instance of a device or to suspend operation for an instance of a device by power management. This section discusses the operation of detaching device instances. Refer to Chapter 12, “Power Management,” for a discussion of power management issues.

A driver’s `detach()` entry point is called to detach an instance of a device that is bound to the driver. The entry point is called with the instance of the device node to be detached and with `DDI_DETACH`, which is specified as the `cmd` argument to the entry point.

A driver is required to cancel or wait for any time outs or callbacks to complete, then release any resources that are allocated to the device instance before returning. If for some reason a driver cannot cancel outstanding callbacks for free resources, the driver is required to return the device to its original state and return `DDI_FAILURE` from the entry point, leaving the device instance in the attached state.
There are two types of callback routines: those callbacks that can be canceled and those that cannot be canceled. `timeout(9F)` and `bufcall(9F)` callbacks can be atomically cancelled by the driver during `detach(9E)`. Other types of callbacks such as `scsi_init_pkt(9F)` and `ddi_dma_buf_bind_handle(9F)` cannot be canceled. The driver must either block in `detach()` until the callback completes or else fail the request to detach.

**EXAMPLE 6–6** Typical `detach()` Entry Point

```c
/*
 * detach(9e)
 * free the resources that were allocated in attach(9e)
 */
static int
xxdetach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
    Pio    *pio_p;
    int     instance;

    switch (cmd) {
    case DDI_DETACH:

        instance = ddi_get_instance(dip);
        pio_p = ddi_get_soft_state(pio_softstate, instance);

        /*
         * turn off the device
         * free any resources allocated in attach
         */
        ddi_put8(pio_p->csr_handle, pio_p->csr, PIO_RESET);
        ddi_remove_minor_node(dip, NULL);
        ddi_regs_map_free(&pio_p->csr_handle);
        ddi_regs_map_free(&pio_p->data_handle);
        ddi_remove_intr(pio_p->dip, 0, pio_p->iblock_cookie);
        mutex_destroy(&pio_p->mutex);
        ddi_soft_state_free(pio_softstate, instance);
        return (DDI_SUCCESS);

    case DDI_SUSPEND:
    default:
        return (DDI_FAILURE);
    }
}
```

**getinfo() Entry Point**

The system calls `getinfo(9E)` to obtain configuration information that only the driver knows. The mapping of minor numbers to device instances is entirely under the control of the driver. The system sometimes needs to ask the driver which device a particular `dev_t` represents.

The `getinfo()` function can take either `DDI_INFO_DEVT2INSTANCE` or `DDI_INFO_DEVT2DEVINFO` as its `infocmd` argument. The `DDI_INFO_DEVT2INSTANCE` command requests the instance number of a device. The `DDI_INFO_DEVT2DEVINFO` command requests a pointer to the `dev_info` structure of a device.
In the DDI_INFO_DEVT2INSTANCE case, arg is a dev_t, and getinfo() must translate the minor number in dev_t to an instance number. In the following example, the minor number is the instance number, so getinfo() simply passes back the minor number. In this case, the driver must not assume that a state structure is available, since getinfo() might be called before attach(). The mapping defined by the driver between the minor device number and the instance number does not necessarily follow the mapping shown in the example. In all cases, however, the mapping must be static.

In the DDI_INFO_DEVT2DEVINFO case, arg is again a dev_t, so getinfo() first decodes the instance number for the device. getinfo() then passes back the dev_info pointer saved in the driver’s soft state structure for the appropriate device, as shown in the following example.

EXAMPLE 6-7 Typical getinfo() Entry Point

```c
/*
 * getinfo(9e)
 * Return the instance number or device node given a dev_t
 */
static int
xxgetinfo(dev_info_t *dip, ddi_info_cmd_t infocmd, void *arg, void **result)
{
    int error;
    Pio *pio_p;
    int instance = getminor((dev_t)arg);

    switch (infocmd) {
        /*
        * return the device node if the driver has attached the
        * device instance identified by the dev_t value which was passed
        */
        case DDI_INFO_DEVT2DEVINFO:
            pio_p = ddi_get_soft_state(pio_softstate, instance);
            if (pio_p == NULL) {
                *result = NULL;
                error = DDI_FAILURE;
            } else {
                mutex_enter(&pio_p->mutex);
                *result = pio_p->dip;
                mutex_exit(&pio_p->mutex);
                error = DDI_SUCCESS;
            }
            break;
        /*
        * the driver can always return the instance number given a dev_t
        * value, even if the instance is not attached.
        */
        case DDI_INFO_DEVT2INSTANCE:
            *result = (void *)instance;
            error = DDI_SUCCESS;
            break;
        default:
            *result = NULL;
            error = DDI_FAILURE;
    }
    return (error);
```
Note – The getinfo() routine must be kept in sync with the minor nodes that the driver creates. If the minor nodes get out of sync, any hotplug operations might fail and cause a system panic.

Using Device IDs

The Oracle Solaris DDI interfaces enable drivers to provide the device ID, a persistent unique identifier for a device. The device ID can be used to identify or locate a device. The device ID is independent of the /devices name or device number (dev_t). Applications can use the functions defined in libdevid(3LIB) to read and manipulate the device IDs registered by the drivers.

Before a driver can export a device ID, the driver needs to verify the device is capable of either providing a unique ID or of storing a host-generated unique ID in a not normally accessible area. WWN (world-wide number) is an example of a unique ID that is provided by the device. Device NVRAM and reserved sectors are examples of non-accessible areas where host-generated unique IDs can be safely stored.

Registering Device IDs

Drivers typically initialize and register device IDs in the driver’s attach(9E) handler. As mentioned above, the driver is responsible for registering a device ID that is persistent. As such, the driver might be required to handle both devices that can provide a unique ID directly (WWN) and devices where fabricated IDs are written to and read from stable storage.

Registering a Device-Supplied ID

If the device can supply the driver with an identifier that is unique, the driver can simply initialize the device ID with this identifier and register the ID with the Oracle Solaris DDI.

/*
 * The device provides a guaranteed unique identifier, 
 * in this case a SCSI3-WWN. The WWN for the device has been 
 * stored in the device’s soft state.
 */

if (ddi_devid_init(dip, DEVID_SCSI3_WWN, un->un_wwn_len, un->un_wwn, 
 &un->un_devid) != DDI_SUCCESS) 
 return (DDI_FAILURE);

(void) ddi_devid_register(dip, un->un_devid);
Registering a Fabricated ID

A driver might also register device IDs for devices that do not directly supply a unique ID. Registering these IDs requires the device to be capable of storing and retrieving a small amount of data in a reserved area. The driver can then create a fabricated device ID and write it to the reserved area.

```c
/*
 * the device doesn’t supply a unique ID, attempt to read
 * a fabricated ID from the device’s reserved data.
 */
if (xxx_read_deviceid(un, &devid_buf) == XXX_OK) {
    if (ddi_devid_valid(devid_buf) == DDI_SUCCESS) {
        devid_sz = ddi_devi_sizeof(devid_buf);
        un->un_devid = kmem_alloc(devid_sz, KM_SLEEP);
        bcopy(devid_buf, un->un_devid, devid_sz);
        ddi_devid_register(dip, un->un_devid);
        return (XXX_OK);
    }
}
/*
 * we failed to read a valid device ID from the device
 * fabricate an ID, store it on the device, and register
 * it with the DDI
 */
if (ddi_devid_init(dip, DEVID_FAB, 0, NULL, &un->un_devid)
    == DDI_FAILURE) {
    return (XXX_FAILURE);
}
if (xxx_write_deviceid(un) != XXX_OK) {
    ddi_devid_free(un->un_devid);
    un->un_devid = NULL;
    return (XXX_FAILURE);
}
ddi_devid_register(dip, un->un_devid);
return (XXX_OK);
```

Unregistering Device IDs

Drivers typically unregister and free any device IDs that are allocated as part of the detach(9E) handling. The driver first calls ddi_devid_unregister(9F) to unregister the device ID for the device instance. The driver must then free the device ID handle itself by calling ddi_devid_free(9F), and then passing the handle that had been returned by ddi_devid_init(9F). The driver is responsible for managing any space allocated for WWN or Serial Number data.
The Oracle Solaris OS provides driver developers with a comprehensive set of interfaces for accessing device memory. These interfaces are designed to shield the driver from platform-specific dependencies by handling mismatches between processor and device endianness as well as enforcing any data order dependencies the device might have. By using these interfaces, you can develop a single-source driver that runs on both the SPARC and x86 processor architectures as well as the various platforms from each respective processor family.

This chapter provides information on the following subjects:

- "Managing Differences in Device and Host Endianness" on page 116
- "Managing Data Ordering Requirements" on page 116
- "ddi_device_acc_attr Structure" on page 116
- "Mapping Device Memory" on page 117
- "Mapping Setup Example" on page 117
- "Alternate Device Access Interfaces" on page 120

### Device Memory

Devices that support programmed I/O are assigned one or more regions of bus address space that map to addressable regions of the device. These mappings are described as pairs of values in the reg property associated with the device. Each value pair describes a segment of a bus address.

Drivers identify a particular bus address mapping by specifying the register number, or regspec, which is an index into the devices’ reg property. The reg property identifies the busaddr and size for the device. Drivers pass the register number when making calls to DDI functions such as ddi_regs_map_setup(9F). Drivers can determine how many mappable regions have been assigned to the device by calling ddi_dev_nregs(9F).
Managing Differences in Device and Host Endianness

The data format of the host can have different endian characteristics than the data format of the device. In such a case, data transferred between the host and device would need to be byte-swapped to conform to the data format requirements of the destination location. Devices with the same endian characteristics of the host require no byte-swapping of the data.

Drivers specify the endian characteristics of the device by setting the appropriate flag in the `ddi_device_acc_attr(9S)` structure that is passed to `ddi_regs_map_setup(9F)`. The DDI framework then performs any required byte-swapping when the driver calls a `ddi_getX` routine like `ddi_get8(9F)` or a `ddi_putX` routine like `ddi_put16(9F)` to read or write to device memory.

Managing Data Ordering Requirements

Platforms can reorder loads and stores of data to optimize performance of the platform. Because reordering might not be allowed by certain devices, the driver is required to specify the device’s ordering requirements when setting up mappings to the device.

**ddi_device_acc_attr Structure**

This structure describes the endian and data order requirements of the device. The driver is required to initialize and pass this structure as an argument to `ddi_regs_map_setup(9F)`.

```c
typedef struct ddi_device_acc_attr {
    ushort_t devacc_attr_version;
    uchar_t devacc_attr_endian_flags;
    uchar_t devacc_attr_dataorder;
} ddi_device_acc_attr_t;
```

- **devacc_attr_version** Specifies DDI_DEVICE_ATTR_V0
- **devacc_attr_endian_flags** Describes the endian characteristics of the device. Specified as a bit value whose possible values are:
  - `DDI_NEVERSWAP_ACC` – Never swap data
  - `DDI_STRUCTURE_BE_ACC` – The device data format is big-endian
  - `DDI_STRUCTURE_LE_ACC` – The device data format is little-endian
- **devacc_attr_dataorder** Describes the order in which the CPU must reference data as required by the device. Specified as an enumerated value, where data access restrictions are ordered from most strict to least strict.
- **DDI_STRICTORDER_ACC** – The host must issue the references in order, as specified by the programmer. This flag is the default behavior.
- **DDI_UNORDERED_OK_ACC** – The host is allowed to reorder loads and stores to device memory.
- **DDI_MERGING_OK_ACC** – The host is allowed to merge individual stores to consecutive locations. This setting also implies reordering.
- **DDI_LOADCACHING_OK_ACC** – The host is allowed to read data from the device until a store occurs.
- **DDI_STORECACHING_OK_ACC** – The host is allowed to cache data written to the device. The host can then defer writing the data to the device until a future time.

**Note** – The system can access data more strictly than the driver specifies in `devacc_attr_dataorder`. The restriction to the host diminishes while moving from strict data ordering to cache storing in terms of data accesses by the driver.

### Mapping Device Memory

Drivers typically map all regions of a device during `attach(9E)`. The driver maps a region of device memory by calling `ddi_regs_map_setup(9F)`, specifying the register number of the region to map, the device access attributes for the region, an offset, and size. The DDI framework sets up the mappings for the device region and returns an opaque handle to the driver. This data access handle is passed as an argument to the `ddi_get8(9F)` or `ddi_put8(9F)` family of routines when reading data from or writing data to that region of the device.

The driver verifies that the shape of the device mappings match what the driver is expecting by checking the number of mappings exported by the device. The driver calls `ddi_dev_nregs(9F)` and then verifies the size of each mapping by calling `ddi_dev_regsize(9F)`.

### Mapping Setup Example

The following simple example demonstrates the DDI data access interfaces. This driver is for a fictional little endian device that accepts one character at a time and generates an interrupt when ready for another character. This device implements two register sets: the first is an 8-bit CSR register, and the second is an 8-bit data register.
EXAMPLE 7–1 Mapping Setup

```c
#define CSR_REG 0
#define DATA_REG 1
/*
 * Initialize the device access attributes for the register
 * mapping
 */
dev_acc_attr.devacc_attr_version = DDI_DEVICE_ATTR_V0;
dev_acc_attr.devacc_attr_endian_flags = DDI_STRUCTURE_LE_ACC;
dev_acc_attr.devacc_attr_dataorder = DDI_STRICTORDER_ACC;
/*
 * Map in the csr register (register 0)
 */
if (ddi_regs_map_setup(dip, CSR_REG, (caddr_t *)&(pio_p->csr), 0,
    sizeof (Pio_csr), &dev_acc_attr, &pio_p->csr_handle) != DDI_SUCCESS) {
    mutex_destroy(&pio_p->mutex);
    ddi_soft_state_free(pio_softstate, instance);
    return (DDI_FAILURE);
}
/*
 * Map in the data register (register 1)
 */
if (ddi_regs_map_setup(dip, DATA_REG, (caddr_t *)&(pio_p->data), 0,
    sizeof (uchar_t), &dev_acc_attr, &pio_p->data_handle) != DDI_SUCCESS) {
    mutex_destroy(&pio_p->mutex);
    mutex_destroy(&pio_p->csr_handle);
    ddi_regs_map_free(&pio_p->csr_handle);
    return (DDI_FAILURE);
}
```

Device Access Functions

Drivers use the `ddi_get8(9F)` and `ddi_put8(9F)` family of routines in conjunction with the handle returned by `ddi_regs_map_setup(9F)` to transfer data to and from a device. The DDI framework automatically handles any byte-swapping that is required to meet the endian format for the host or device, and enforces any store-ordering constraints the device might have.

The DDI provides interfaces for transferring data in 8-bit, 16-bit, 32-bit, and 64-bit quantities, as well as interfaces for transferring multiple values repeatedly. See the man pages for the `ddi_get8(9F), ddi_put8(9F), ddi_rep_get8(9F)` and `ddi_rep_put8(9F)` families of routines for a complete listing and description of these interfaces.

The following example builds on Example 7–1 where the driver mapped the device’s CSR and data registers. Here, the driver’s `write(9E)` entry point, when called, writes a buffer of data to the device one byte at a time.
EXAMPLE 7-2  Mapping Setup: Buffer

```c
static int
pio_write(dev_t dev, struct uio *uiop, cred_t *credp)
{
    int retval;
    int error = OK;
    Pio *pio_p = ddi_get_soft_state(pio_softstate, getminor(dev));
    if (pio_p == NULL)
        return (ENXIO);
    mutex_enter(&pio_p->mutex);
    /*
       * enable interrupts from the device by setting the Interrupt
       * Enable bit in the devices CSR register
       */
    ddi_put8(pio_p->csr_handle, pio_p->csr,
        (ddi_get8(pio_p->csr_handle, pio_p->csr) | PIO_INTR_ENABLE));
    while (uiop->uio_resid > 0) {
        /*
           * This device issues an IDLE interrupt when it is ready
           * to accept a character; the interrupt can be cleared
           * by setting PIO_INTR_CLEAR. The interrupt is reasserted
           * after the next character is written or the next time
           * PIO_INTR_ENABLE is toggled on.
           *
           * wait for interrupt (see pio_intr)
           */
        cv_wait(&pio_p->cv, &pio_p->mutex);
        /*
           * get a character from the user’s write request
           * fail the write request if any errors are encountered
           */
        if ((retval = uwritec(uiop)) == -1) {
            error = retval;
            break;
        }
        /*
           * pass the character to the device by writing it to
           * the device’s data register
           */
        ddi_put8(pio_p->data_handle, pio_p->data, (uchar_t)retval);
    }
    /*
       * disable interrupts by clearing the Interrupt Enable bit
       * in the CSR
       */
    ddi_put8(pio_p->csr_handle, pio_p->csr,
        (ddi_get8(pio_p->csr_handle, pio_p->csr) & ~PIO_INTR_ENABLE));
    mutex_exit(&pio_p->mutex);
    return (error);
}
```
Alternate Device Access Interfaces

In addition to implementing all device accesses through the `ddi_get8(9F)` and `ddi_put8(9F)` families of interfaces, the Oracle Solaris OS provides interfaces that are specific to particular bus implementations. While these functions can be more efficient on some platforms, use of these routines can limit the ability of the driver to remain portable across different bus versions of the device.

Memory Space Access

With memory mapped access, device registers appear in memory address space. The `ddi_getX` family of routines and the `ddi_putX` family are available for use by drivers as an alternative to the standard device access interfaces.

I/O Space Access

With I/O space access, the device registers appear in I/O space, where each addressable element is called an I/O port. The `ddi io_get8(9F)` and `ddi io_put8(9F)` routines are available for use by drivers as an alternative to the standard device access interfaces.

PCI Configuration Space Access

To access PCI configuration space without using the normal device access interfaces, a driver is required to map PCI configuration space by calling `pci_config_setup(9F)` in place of `ddi_regs_map_setup(9F)`. The driver can then call the `pci_config_get8(9F)` and `pci_config_put8(9F)` families of interfaces to access PCI configuration space.
This chapter describes mechanisms for handling interrupts, such as allocating, registering, servicing, and removing interrupts. This chapter provides information on the following subjects:

- “Interrupt Handler Overview” on page 121
- “Device Interrupts” on page 122
- “Registering Interrupts” on page 128
- “Interrupt Resource Management” on page 134
- “Interrupt Handler Functionality” on page 145
- “Handling High-Level Interrupts” on page 147

Interrupt Handler Overview

An interrupt is a hardware signal from a device to a CPU. An interrupt tells the CPU that the device needs attention and that the CPU should stop any current activity and respond to the device. If the CPU is not performing a task that has higher priority than the priority of the interrupt, then the CPU suspends the current thread. The CPU then invokes the interrupt handler for the device that sent the interrupt signal. The job of the interrupt handler is to service the device and stop the device from interrupting. When the interrupt handler returns, the CPU resumes the work it was doing before the interrupt occurred.

The Oracle Solaris DDI/DKI provides interfaces for performing the following tasks:

- Determining interrupt type and registration requirements
- Registering interrupts
- Servicing interrupts
- Masking interrupts
- Getting interrupt pending information
- Getting and setting priority information
Device Interrupts

I/O buses implement interrupts in two common ways: vectored and polled. Both methods commonly supply a bus-interrupt priority level. Vectored devices also supply an interrupt vector. Polled devices do not supply interrupt vectors.

To stay current with changing bus technologies, the Oracle Solaris OS has been enhanced to accommodate both newer types of interrupts and more traditional interrupts that have been in use for many years. Specifically, the operating system now recognizes three types of interrupts:

- **Legacy interrupts** – Legacy or fixed interrupts refer to interrupts that use older bus technologies. With these technologies, interrupts are signaled by using one or more external pins that are wired “out-of-band,” that is, separately from the main lines of the bus. Newer bus technologies such as PCI Express maintain software compatibility by emulating legacy interrupts through in-band mechanisms. These emulated interrupts are treated as legacy interrupts by the host OS.

- **Message-signaled interrupts** – Instead of using pins, message-signaled interrupts (MSI) are in-band messages and can target addresses in the host bridge. (See “PCI Local Bus” on page 545 for more information on host bridges.) MSIs can send data along with the interrupt message. Each MSI is unshared so that an MSI that is assigned to a device is guaranteed to be unique within the system. A PCI function can request up to 32 MSI messages.

- **Extended message-signaled interrupts** – Extended message-signaled interrupts (MSI-X) are an enhanced version of MSIs. MSI-X interrupts have the following added advantages:
  - Support 2048 messages rather than 32 messages
  - Support independent message address and message data for each message
  - Support per-message masking
  - Enable more flexibility when software allocates fewer vectors than hardware requests. The software can reuse the same MSI-X address and data in multiple MSI-X slots.

Note – Some newer bus technologies such as PCI Express require MSIs but can accommodate legacy interrupts by using INTx emulation. INTx emulation is used for compatibility purposes, but is not considered to be good practice.

High-Level Interrupts

A bus prioritizes a device interrupt at a bus-interrupt level. The bus interrupt level is then mapped to a processor-interrupt level. A bus interrupt level that maps to a CPU interrupt priority above the scheduler priority level is called a high-level interrupt. High-level interrupt handlers are restricted to calling the following DDI interfaces:

- `mutex_enter(9F)` and `mutex_exit(9F)` on a mutex that is initialized with an interrupt priority associated with the high-level interrupt
The following DDI get and put routines: 

- `ddi_get8(9F)`
- `ddi_put8(9F)`
- `ddi_get16(9F)`
- `ddi_put16(9F)`
- `ddi_get32(9F)`
- `ddi_put32(9F)`
- `ddi_get64(9F)`
- `ddi_put64(9F)`

A bus-interrupt level by itself does not determine whether a device interrupts at a high level. A particular bus-interrupt level can map to a high-level interrupt on one platform, but map to an ordinary interrupt on another platform.

A driver is not required to support devices that have high-level interrupts. However, the driver is required to check the interrupt level. If the interrupt priority is greater than or equal to the highest system priority, the interrupt handler runs in high-level interrupt context. In this case, the driver can fail to attach, or the driver can use a two-level scheme to handle interrupts. For more information, see “Handling High-Level Interrupts” on page 147.

Legacy Interrupts

The only information that the system has about a device interrupt is the priority level of the bus interrupt and the interrupt request number. An example of the priority level for a bus interrupt is the IPL on an SBus in a SPARC machine. An example of an interrupt request number is the IRQ on an ISA bus in an x86 machine.

When an interrupt handler is registered, the system adds the handler to a list of potential interrupt handlers for each IPL or IRQ. When the interrupt occurs, the system must determine which device actually caused the interrupt, among all devices that are associated with a given IPL or IRQ. The system calls all the interrupt handlers for the designated IPL or IRQ until one handler claims the interrupt.

The following buses are capable of supporting polled interrupts:

- SBus
- ISA
- PCI

Standard and Extended Message-Signaled Interrupts

Both standard (MSI) and extended (MSI-X) message-signaled interrupts are implemented as in-band messages. A message-signaled interrupt is posted as a write with an address and value that are specified by the software.

MSI Interrupts

Conventional PCI specifications include optional support for Message Signaled Interrupts (MSI). An MSI is an in-band message that is implemented as a posted write. The address and the data for the MSI are specified by software and are specific to the host bridge. Because the
messages are in-band, the receipt of the message can be used to "push" data that is associated with the interrupt. By definition, MSI interrupts are unshared. Each MSI message that is assigned to a device is guaranteed to be a unique message in the system. PCI functions can request 1, 2, 4, 8, 16, or 32 MSI messages. Note that the system software can allocate fewer MSI messages to a function than the function requested. The host bridge can be limited in the number of unique MSI messages that are allocated for devices.

**MSI-X Interrupts**

MSI-X interrupts are enhanced versions of MSI interrupts that have the same features as MSI interrupts with the following key differences:

- A maximum of 2048 MSI-X interrupt vectors are supported per device.
- Address and data entries are unique per interrupt vector.
- MSI-X supports per function masking and per vector masking.

With MSI-X interrupts, an unallocated interrupt vector of a device can use a previously added or initialized MSI-X interrupt vector to share the same vector address, vector data, interrupt handler, and handler arguments. Use the `ddi_intr_dup_handler(9F)` function to alias the resources provided by the Oracle Solaris OS to the unallocated interrupt vectors on an associated device. For example, if 2 MSI-X interrupts are allocated to a driver and 32 interrupts are supported on the device, then the driver can use `ddi_intr_dup_handler()` to alias the 2 interrupts it received to the 30 additional interrupts on the device.

The `ddi_intr_dup_handler()` function can duplicate interrupts that were added with `ddi_intr_add_handler(9F)` or initialized with `ddi_intr_enable(9F)`.

A duplicated interrupt is disabled initially. Use `ddi_intr_enable()` to enable the duplicated interrupt. You cannot remove the original MSI-X interrupt handler until all duplicated interrupt handlers that are associated with this original interrupt handler are removed. To remove a duplicated interrupt handler, first call `ddi_intr_disable(9F)`, and then call `ddi_intr_free(9F)`. When all duplicated interrupt handlers that are associated with this original interrupt handler are removed, you can use `ddi_intr_remove_handler(9F)` to remove the original MSI-X interrupt handler. See the `ddi_intr_dup_handler(9F)` man page for examples.

**Software Interrupts**

The Oracle Solaris DDI/DKI supports software interrupts, also known as *soft interrupts*. Soft interrupts are initiated by software rather than by a hardware device. Handlers for these interrupts must also be added to and removed from the system. Soft interrupt handlers run in interrupt context and therefore can be used to do many of the tasks that belong to an interrupt handler.
Hardware interrupt handlers must perform their tasks quickly, because the handlers might have to suspend other system activity while doing these tasks. This requirement is particularly true for high-level interrupt handlers, which operate at priority levels greater than the priority level of the system scheduler. High-level interrupt handlers mask the operations of all lower-priority interrupts, including the interrupt operations of the system clock. Consequently, the interrupt handler must avoid involvement in activities that might cause it to sleep, such as acquiring a mutex.

If the handler sleeps, then the system might hang because the clock is masked and incapable of scheduling the sleeping thread. For this reason, high-level interrupt handlers normally perform a minimum amount of work at high-priority levels and delegate other tasks to software interrupts, which run below the priority level of the high-level interrupt handler. Because software interrupt handlers run below the priority level of the system scheduler, software interrupt handlers can do the work that the high-level interrupt handler was incapable of doing.

**DDI Interrupt Functions**

The Oracle Solaris OS provides a framework for registering and unregistering interrupts and provides support for Message Signaled Interrupts (MSIs). Interrupt management interfaces enable you to manipulate priorities, capabilities, and interrupt masking, and to obtain pending information.

**Interrupt Capability Functions**

Use the following functions to obtain interrupt information:

- `ddi_intr_get_navail(9F)`
  - Returns the number of interrupts available for a specified hardware device and interrupt type.

- `ddi_intr_get_nintrs(9F)`
  - Returns the number of interrupts that the device supports for the specified interrupt type.

- `ddi_intr_get_supported_types(9F)`
  - Returns the hardware interrupt types that are supported by both the device and the host.

- `ddi_intr_get_cap(9F)`
  - Returns interrupt capability flags for the specified interrupt.

**Interrupt Initialization and Destruction Functions**

Use the following functions to create and remove interrupts:

- `ddi_intr_alloc(9F)`
  - Allocates system resources and interrupt vectors for the specified type of interrupt.
**DDI Interrupt Functions**

- **ddi_intr_free(9F)**: Releases the system resources and interrupt vectors for a specified interrupt handle.
- **ddi_intr_set_cap(9F)**: Sets the capability of the specified interrupt through the use of the DDI_INTR_FLAG_LEVEL and DDI_INTR_FLAG_EDGE flags.
- **ddi_intr_add_handler(9F)**: Adds an interrupt handler.
- **ddi_intr_dup_handler(9F)**: Use with MSI-X only. Copies an address and data pair for an allocated interrupt vector to an unused interrupt vector on the same device.
- **ddi_intr_remove_handler(9F)**: Removes the specified interrupt handler.
- **ddi_intr_enable(9F)**: Enables the specified interrupt.
- **ddi_intr_disable(9F)**: Disables the specified interrupt.
- **ddi_intr_block_enable(9F)**: Use with MSI only. Enables the specified range of interrupts.
- **ddi_intr_block_disable(9F)**: Use with MSI only. Disables the specified range of interrupts.
- **ddi_intr_set_mask(9F)**: Sets an interrupt mask if the specified interrupt is enabled.
- **ddi_intr_clr_mask(9F)**: Clears an interrupt mask if the specified interrupt is enabled.
- **ddi_intr_get_pending(9F)**: Reads the interrupt pending bit if such a bit is supported by either the host bridge or the device.

**Priority Management Functions**

Use the following functions to obtain and set priority information:

- **ddi_intr_get_pri(9F)**: Returns the current software priority setting for the specified interrupt.
- **ddi_intr_set_pri(9F)**: Sets the interrupt priority level for the specified interrupt.
- **ddi_intr_get_hilevel_pri(9F)**: Returns the minimum priority level for a high-level interrupt.

**Soft Interrupt Functions**

Use the following functions to manipulate soft interrupts and soft interrupt handlers:
**DDIInterruptFunctions**

**ddi_intr_add_softint(9F)**
Adds a soft interrupt handler.

**ddi_intr_trigger_softint(9F)**
Triggers the specified soft interrupt.

**ddi_intr_remove_softint(9F)**
Removes the specified soft interrupt handler.

**ddi_intr_get_softint_pri(9F)**
Returns the soft interrupt priority for the specified interrupt.

**ddi_intr_set_softint_pri(9F)**
Changes the relative soft interrupt priority for the specified soft interrupt.

### Interrupt Function Examples

This section provides examples for performing the following tasks:

- Changing soft interrupt priority
- Checking for pending interrupts
- Setting interrupt masks
- Clearing interrupt masks

**EXAMPLE 8–1  Changing Soft Interrupt Priority**

Use the `ddi_intr_set_softint_pri(9F)` function to change the soft interrupt priority to 9.

```c
if (ddi_intr_set_softint_pri(mydev->mydev_softint_hdl, 9) != DDI_SUCCESS) 
    cmn_err (CE_WARN, "ddi_intr_set_softint_pri failed");
```

**EXAMPLE 8–2  Checking for Pending Interrupts**

Use the `ddi_intr_get_pending(9F)` function to check whether an interrupt is pending.

```c
if (ddi_intr_get_pending(mydev->htable[0], &pending) != DDI_SUCCESS) 
    cmn_err (CE_WARN, "ddi_intr_get_pending() failed");
else if (pending) 
    cmn_err (CE_NOTE, "ddi_intr_get_pending(): Interrupt pending");
```

**EXAMPLE 8–3  Setting Interrupt Masks**

Use the `ddi_intr_set_mask(9F)` function to set interrupt masking to prevent the device from receiving interrupts.

```c
if ((ddi_intr_set_mask(mydev->htable[0]) != DDI_SUCCESS)) 
    cmn_err (CE_WARN, "ddi_intr_set_mask() failed");
```
EXAMPLE 8–4  Clearing Interrupt Masks

Use the `ddi_intr_clr_mask(9F)` function to clear interrupt masking. The `ddi_intr_clr_mask(9F)` function fails if the specified interrupt is not enabled. If the `ddi_intr_clr_mask(9F)` function succeeds, the device starts generating interrupts.

```c
if (ddi_intr_clr_mask(mydevp->htable[0]) != DDI_SUCCESS)
    cmn_err(CE_WARN, "ddi_intr_clr_mask() failed");
```

Registering Interrupts

Before a device driver can receive and service interrupts, the driver must call `ddi_intr_add_handler(9F)` to register an interrupt handler with the system. Registering interrupt handlers provides the system with a way to associate an interrupt handler with an interrupt specification. The interrupt handler is called when the device might have been responsible for the interrupt. The handler has the responsibility of determining whether it should handle the interrupt and, if so, of claiming that interrupt.

**Tip** – Use the `::interrupts` command in the `mdb` or `kmdb` debugger to retrieve the registered interrupt information of a device on supported SPARC and x86 systems.

Registering Legacy Interrupts

To register a driver’s interrupt handler, the driver typically performs the following steps in its `attach(9E)` entry point:

1. Use `ddi_intr_get_supported_types(9F)` to determine which types of interrupts are supported.
2. Use `ddi_intr_get_nintrs(9F)` to determine the number of supported interrupt types.
3. Use `kmem_zalloc(9F)` to allocate memory for DDI interrupt handles.
4. For each interrupt type that you allocate, take the following steps:
   a. Use `ddi_intr_get_pri(9F)` to get the priority for the interrupt.
   b. If you need to set a new priority for the interrupt, use `ddi_intr_set_pri(9F)`.
   c. Use `mutex_init(9F)` to initialize the lock.
   d. Use `ddi_intr_add_handler(9F)` to register the handler for the interrupt.
   e. Use `ddi_intr_enable(9F)` to enable the interrupt.
5. Take the following steps to free each interrupt:
   a. Disable each interrupt using `ddi_intr_disable(9F)`.
   b. Remove the interrupt handler using `ddi_intr_remove_handler(9F)`. 
c. Remove the lock using `mutex_destroy(9F)`.

d. Free the interrupt using `ddi_intr_free(9F)` and `kmem_free(9F)` to free memory that was allocated for DDI interrupt handles.

**EXAMPLE 8-5  Registering a Legacy Interrupt**

The following example shows how to install an interrupt handler for a device called `mydev`. This example assumes that `mydev` supports one interrupt only.

```c
/* Determine which types of interrupts supported */
ret = ddi_intr_get_supported_types(mydevp->mydev_dip, &type);
if ((ret != DDI_SUCCESS) || (!(type & DDI_INTR_TYPE_FIXED))) {
    cmn_err(CE_WARN, "Fixed type interrupt is not supported");
    return (DDI_FAILURE);
}

/* Determine number of supported interrupts */
ret = ddi_intr_get_nintrs(mydevp->mydev_dip, DDI_INTR_TYPE_FIXED, &count);

/* Fixed interrupts can only have one interrupt. Check to make sure that number of supported interrupts and number of available interrupts are both equal to 1. */
if ((ret != DDI_SUCCESS) || (count != 1)) {
    cmn_err(CE_WARN, "No fixed interrupts");
    return (DDI_FAILURE);
}

/* Allocate memory for DDI interrupt handles */
mydevp->mydev_htable = kmem_zalloc(sizeof (ddi_intr_handle_t), KM_SLEEP);
ret = ddi_intr_alloc(mydevp->mydev_dip, mydevp->mydev_htable, DDI_INTR_TYPE_FIXED, 0, &count, &actual);
if ((ret != DDI_SUCCESS) || (actual != 1)) {
    cmn_err(CE_WARN, "ddi_intr_alloc() failed 0x%lx", ret);
    kmem_free(mydevp->mydev_htable, sizeof (ddi_intr_handle_t));
    return (DDI_FAILURE);
}

/* Sanity check that count and available are the same. */
ASSERT(count == actual);

/* Get the priority of the interrupt */
if (ddi_intr_get_pri(mydevp->mydev_htable[0], &mydevp->mydev_intr_pri)) {
    cmn_err(CE_WARN, "ddi_intr_alloc() failed 0x%x", ret);
    (void) ddi_intr_free(mydevp->mydev_htable[0]);
    kmem_free(mydevp->mydev_htable, sizeof (ddi_intr_handle_t));
    return (DDI_FAILURE);
}
```
EXAMPLE 8–5  Registering a Legacy Interrupt  (Continued)

```c
cmn_err(CE_NOTE, "Supported Interrupt pri = 0x%x", mydevp->mydev_intr_pri);

/* Test for high level mutex */
if (mydevp->mydev_intr_pri >= ddi_intr_get_hilevel_pri()) {
    cmn_err(CE_WARN, "Hi level interrupt not supported");
    (void) ddi_intr_free(mydevp->mydev_htable[0]);
    kmem_free(mydevp->mydev_htable, sizeof (ddi_intr_handle_t));
    return (DDI_FAILURE);
}

/* Initialize the mutex */
mutex_init(&mydevp->mydev_int_mutex, NULL, MUTEX_DRIVER,
    DDI_INTR_PRI(mydevp->mydev_intr_pri));

/* Register the interrupt handler */
if (ddi_intr_add_handler(mydevp->mydev_htable[0], mydev_intr, 
    (caddr_t)mydevp, NULL) != DDI_SUCCESS) {
    cmn_err(CE_WARN, "ddi_intr_add_handler() failed");
    mutex_destroy(&mydevp->mydev_int_mutex);
    (void) ddi_intr_free(mydevp->mydev_htable[0]);
    kmem_free(mydevp->mydev_htable, sizeof (ddi_intr_handle_t));
    return (DDI_FAILURE);
}

/* Enable the interrupt */
if (ddi_intr_enable(mydevp->mydev_htable[0]) != DDI_SUCCESS) {
    cmn_err(CE_WARN, "ddi_intr_enable() failed");
    (void) ddi_intr_remove_handler(mydevp->mydev_htable[0]);
    mutex_destroy(&mydevp->mydev_int_mutex);
    (void) ddi_intr_free(mydevp->mydev_htable[0]);
    kmem_free(mydevp->mydev_htable, sizeof (ddi_intr_handle_t));
    return (DDI_FAILURE);
}
return (DDI_SUCCESS);
}
```

EXAMPLE 8–6  Removing a Legacy Interrupt

The following example shows how legacy interrupts are removed.

```c
/* disable interrupt */
(void) ddi_intr_disable(mydevp->mydev_htable[0]);

/* Remove interrupt handler */
(void) ddi_intr_remove_handler(mydevp->mydev_htable[0]);

/* free interrupt handle */
```
EXAMPLE 8–6  Removing a Legacy Interrupt  (Continued)

```c
(void) ddi_intr_free(mydevp->mydev_htable[0]);
/* free memory */
kmem_free(mydevp->mydev_htable, sizeof (ddi_intr_handle_t));
```

**Registering MSI Interrupts**

To register a driver’s interrupt handler, the driver typically performs the following steps in its `attach(9E)` entry point:

1. Use `ddi_intr_get_supported_types(9F)` to determine which types of interrupts are supported.
2. Use `ddi_intr_get_nintrs(9F)` to determine the number of supported MSI interrupt types.
3. Use `ddi_intr_alloc(9F)` to allocate memory for the MSI interrupts.
4. For each interrupt type that you allocate, take the following steps:
   a. Use `ddi_intr_get_pri(9F)` to get the priority for the interrupt.
   b. If you need to set a new priority for the interrupt, use `ddi_intr_set_pri(9F)`.
   c. Use `mutex_init(9F)` to initialize the lock.
   d. Use `ddi_intr_add_handler(9F)` to register the handler for the interrupt.
5. Use one of the following functions to enable all the interrupts:
   - Use `ddi_intr_block_enable(9F)` to enable all the interrupts in a block.
   - Use `ddi_intr_enable(9F)` in a loop to enable each interrupt individually.

**EXAMPLE 8–7  Registering a Set of MSI Interrupts**

The following example illustrates how to register an MSI interrupt for a device called `mydev`.

```c
/* Get supported interrupt types */
if (ddi_intr_get_supported_types(devinfo, &intr_types) != DDI_SUCCESS) {
    cmn_err(CE_WARN, "ddi_intr_get_supported_types failed");
    goto attach_fail;
}

if (intr_types & DDI_INTR_TYPE_MSI)
    mydev_add_msi_intrs(mydevp);

/* Check count, available and actual interrupts */
static int
mydev_add_msi_intrs(mydev_t *mydevp)
{
    dev_info_t *devinfo = mydevp->devinfo;
    int count, avail, actual;
    int x, y, rc, inum = 0;

    /* Get number of interrupts */
```
EXAMPLE 8–7 Registering a Set of MSI Interrupts (Continued)

rc = ddi_intr_get_nintrs(devinfo, DDI_INTR_TYPE_MSI, &count);
if ((rc != DDI_SUCCESS) || (count == 0)) {
    cmn_err(CE_WARN, "ddi_intr_get_nintrs() failure, rc: %d, "
           "count: %d", rc, count);
    return (DDI_FAILURE);
}

/* Get number of available interrupts */
rc = ddi_intr_get_navail(devinfo, DDI_INTR_TYPE_MSI, &avail);
if ((rc != DDI_SUCCESS) || (avail == 0)) {
    cmn_err(CE_WARN, "ddi_intr_get_navail() failure, "
           "rc: %d, avail: %d\n", rc, avail);
    return (DDI_FAILURE);
}
if (avail < count) {
    cmn_err(CE_NOTE, "nintrs() returned %d, navail returned %d", count, avail);
}

/* Allocate memory for MSI interrupts */
mydevp->intr_size = count * sizeof (ddi_intr_handle_t);
mydevp->htable = kmem_alloc(mydevp->intr_size, KM_SLEEP);
rc = ddi_intr_alloc(devinfo, mydevp->htable, DDI_INTR_TYPE_MSI, inum,
                    count, &actual, DDI_INTR_ALLOC_NORMAL);
if ((rc != DDI_SUCCESS) || (actual == 0)) {
    cmn_err(CE_WARN, "ddi_intr_alloc() failed: %d", rc);
    kmem_free(mydevp->htable, mydevp->intr_size);
    return (DDI_FAILURE);
}
if (actual < count) {
    cmn_err(CE_NOTE, "Requested: %d, Received: %d", count, actual);
}

mydevp->intr_cnt = actual;
/*
 * Get priority for first msi, assume remaining are all the same
 */
if (ddi_intr_get_pri(mydevp->htable[0], &mydev->intr_pri) !=
    DDI_SUCCESS) {
    cmn_err(CE_WARN, "ddi_intr_get_pri() failed");
    /* Free already allocated intr */
    for (y = 0; y < actual; y++) {
        (void) ddi_intr_free(mydevp->htable[y]);
    }
    kmem_free(mydevp->htable, mydevp->intr_size);
    return (DDI_FAILURE);
}

/* Call ddi_intr_add_handler() */
for (x = 0; x < actual; x++) {

EXAMPLE 8–7  Registering a Set of MSI Interrupts  (Continued)

    if (ddi_intr_add_handler(mydevp->htable[x], mydev_intr, (caddr_t)mydevp, NULL) != DDI_SUCCESS) {
        cmn_err(CE_WARN, "ddi_intr_add_handler() failed");
        /* Free already allocated intr */
        for (y = 0; y < actual; y++) {
            (void) ddi_intr_free(mydevp->htable[y]);
        }
        kmem_free(mydevp->htable, mydevp->intr_size);
        return (DDI_FAILURE);
    }

    (void) ddi_intr_get_cap(mydevp->htable[0], &mydevp->intr_cap);
    if (mydev->m_intr_cap & DDI_INTR_FLAG_BLOCK) {
        /* Call ddi_intr_block_enable() for MSI */
        (void) ddi_intr_block_enable(mydev->m_htable, mydev->m_intr_cnt);
    } else {
        /* Call ddi_intr_enable() for MSI non block enable */
        for (x = 0; x < mydev->m_intr_cnt; x++) {
            (void) ddi_intr_enable(mydev->m_htable[x]);
        }
    }
    return (DDI_SUCCESS);
}

EXAMPLE 8–8  Removing MSI Interrupts

The following example shows how to remove MSI interrupts.

    static void
    mydev_rem_intrs(mydev_t *mydev)
    {
        int x;

        /* Disable all interrupts */
        if (mydev->m_intr_cap & DDI_INTR_FLAG_BLOCK) {
            /* Call ddi_intr_block_disable() */
            (void) ddi_intr_block_disable(mydev->m_htable, mydev->m_intr_cnt);
        } else {
            for (x = 0; x < mydev->m_intr_cnt; x++) {
                (void) ddi_intr_disable(mydev->m_htable[x]);
            }
        }

        /* Call ddi_intr_remove_handler() */
        for (x = 0; x < mydev->m_intr_cnt; x++) {
            (void) ddi_intr_remove_handler(mydev->m_htable[x]);
            (void) ddi_intr_free(mydev->m_htable[x]);
        }
        kmem_free(mydev->m_htable, mydev->m_intr_size);
    }
Interrupt Resource Management

This section discusses how a driver for a device that can generate many different interruptible conditions can utilize the Interrupt Resource Management feature to optimize its allocation of interrupt vectors.

The Interrupt Resource Management Feature

The Interrupt Resource Management feature can enable a device driver to use more interrupt resources by dynamically managing the driver's interrupt configuration. When the Interrupt Resource Management feature is not used, configuration of interrupt handling usually only occurs in a driver's attach(9E) routine. The Interrupt Resource Management feature monitors the system for changes, recalculates the number of interrupt vectors to grant to each device in response to those changes, and notifies each affected participating driver of the driver's new allocation of interrupt vectors. A participating driver is a driver that has registered a callback handler as described in "Callback Interfaces" on page 135. Changes that can cause interrupt vector reallocation include adding or removing devices, or an explicit request as described in "Modify Number of Interrupt Vectors Requested" on page 138.

The Interrupt Resource Management feature is not available on every Oracle Solaris platform. The feature is only available to PCI devices that utilize MSI-X interrupts.

Note – A driver that utilizes the Interrupt Resource Management feature must be able to adapt correctly when the feature is not available.

When the Interrupt Resource Management feature is available, it can enable a driver to gain access to more interrupt vectors than the driver might otherwise be allocated. A driver might process interrupt conditions more efficiently when utilizing a larger number of interrupt vectors.

The Interrupt Resource Management feature dynamically adjusts the number of interrupt vectors granted to each participating driver depending upon the following constraints:

- Total number available. A finite number of interrupt vectors exists in the system.
- Total number requested. A driver might be granted fewer, but never more than the number of interrupt vectors it requested.
- Fairness to other drivers. The total number of interrupt vectors available is shared by many drivers in a manner that is fair in relation to the total number requested by each driver.
The number of interrupt vectors made available to a device at any given time can vary:

- As other devices are dynamically added to or removed from the system
- As drivers dynamically change the number of interrupt vectors they request in response to load

A driver must provide the following support to use the Interrupt Resource Management feature:

- Callback support. Drivers must register a callback handler so they can be notified when their number of available interrupts has been changed by the system. Drivers must be able to increase or decrease their interrupt usage.
- Interrupt requests. Drivers must specify how many interrupts they want to use.
- Interrupt usage. Drivers must request the correct number of interrupts at any given time, based on:
  - What interruptible conditions their hardware can generate
  - How many processors can be used to process those conditions in parallel
- Interrupt flexibility. Drivers must be flexible enough to assign one or more interruptible conditions to each interrupt vector in a manner that best fits their current number of available interrupts. Drivers might need to reconfigure these assignments when their number of available interrupts increases or decreases at arbitrary times.

**Callback Interfaces**

A driver must use the following interfaces to register its callback support.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Data Structures</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_cb_register()</td>
<td>ddi_cb_flags_t, ddi_cb_handle_t</td>
<td>Register a callback handler function to receive specific types of actions.</td>
</tr>
<tr>
<td>ddi_cb_unregister()</td>
<td>ddi_cb_handle_t</td>
<td>Unregister a callback handler function.</td>
</tr>
<tr>
<td>(*ddi_cb_func_t)()</td>
<td>ddi_cb_action_t</td>
<td>Receive callback actions and specific arguments relevant to each action to be processed.</td>
</tr>
</tbody>
</table>

**Register a Callback Handler Function**

Use the `ddi_cb_register` function to register a callback handler function for a driver.

```c
int
ddi_cb_register (dev_info_t *dip, ddi_cb_flags_t cbflags,
```
The driver can register only one callback function. This one callback function is used to handle all individual callback actions. The cbflags parameter determines which types of actions should be received by the driver when they occur. The cbfunc() routine is called whenever a relevant action should be processed by the driver. The driver specifies two private arguments (arg1 and arg2) to send to itself during each execution of its cbfunc() routine.

The cbflags() parameter is an enumerated type that specifies which actions the driver supports.

typedef enum {
  DDI_CB_FLAG_INTR
} ddi_cb_flags_t;

To register support for Interrupt Resource Management actions, a driver must register a handler and include the DDI_CB_FLAG_INTR flag. When the callback handler is successfully registered, an opaque handle is returned through the ret_hdlp parameter. When the driver is finished with the callback handler, the driver can use the ret_hdlp parameter to unregister the callback handler.

Register the callback handler in the driver's attach(9F) entry point. Save the opaque handle in the driver's soft state. Unregister the callback handler in the driver's detach(9F) entry point.

Unregister a Callback Handler Function

Use the ddi_cb_unregister(9F) function to unregister a callback handler function for a driver.

int
 ddi_cb_unregister (ddi_cb_handle_t hdl);

Make this call in the driver's detach(9F) entry point. After this call, the driver no longer receives callback actions.

The driver also loses any additional support from the system that it gained from having a registered callback handling function. For example, some interrupt vectors previously made available to the driver are immediately taken back when the driver unregisters its callback handling function. Before returning successfully, the ddi_cb_unregister() function notifies the driver of any final actions that result from losing support from the system.

Callback Handler Function

Use the registered callback handling function to receive callback actions and receive arguments that are specific to each action to be processed.

typedef int (*ddi_cb_func_t)(dev_info_t *dip, ddi_cb_action_t cbaction,
  void *cbarg, void *arg1, void *arg2);

typedef enum {
  DDI_CB_FLAG_INTR
} ddi_cb_flags_t;

To register support for Interrupt Resource Management actions, a driver must register a handler and include the DDI_CB_FLAG_INTR flag. When the callback handler is successfully registered, an opaque handle is returned through the ret_hdlp parameter. When the driver is finished with the callback handler, the driver can use the ret_hdlp parameter to unregister the callback handler.

Register the callback handler in the driver's attach(9F) entry point. Save the opaque handle in the driver's soft state. Unregister the callback handler in the driver's detach(9F) entry point.

Unregister a Callback Handler Function

Use the ddi_cb_unregister(9F) function to unregister a callback handler function for a driver.

int
 ddi_cb_unregister (ddi_cb_handle_t hdl);

Make this call in the driver's detach(9F) entry point. After this call, the driver no longer receives callback actions.

The driver also loses any additional support from the system that it gained from having a registered callback handling function. For example, some interrupt vectors previously made available to the driver are immediately taken back when the driver unregisters its callback handling function. Before returning successfully, the ddi_cb_unregister() function notifies the driver of any final actions that result from losing support from the system.

Callback Handler Function

Use the registered callback handling function to receive callback actions and receive arguments that are specific to each action to be processed.

typedef int (*ddi_cb_func_t)(dev_info_t *dip, ddi_cb_action_t cbaction,
  void *cbarg, void *arg1, void *arg2);
The \texttt{cbaction} parameter specifies what action the driver is receiving a callback to process.

\begin{verbatim}
typedef enum {
    DDI_CB_INTR_ADD,
    DDI_CB_INTR_REMOVE
} ddi_cb_action_t;
\end{verbatim}

A \texttt{DDI_CB_INTR_ADD} action means that the driver now has more interrupts available to use. A \texttt{DDI_CB_INTR_REMOVE} action means that the driver now has fewer interrupts available to use. Cast the \texttt{cbarg} parameter to an \texttt{int} to determine the number of interrupts added or removed. The \texttt{cbarg} value represents the change in the number of interrupts that are available.

For example, get the change in the number of interrupts available:

\begin{verbatim}
    count = (int)(uintptr_t)cbarg;
\end{verbatim}

If the \texttt{cbaction} is \texttt{DDI_CB_INTR_ADD}, add \texttt{cbarg} number of interrupt vectors. If the \texttt{cbaction} is \texttt{DDI_CB_INTR_REMOVE}, free \texttt{cbarg} number of interrupt vectors.

See \texttt{ddi_cb_register(9F)} for an explanation of \texttt{arg1} and \texttt{arg2}.

The callback handling function must be able to perform correctly for the entire time that the function is registered. The callback function cannot depend upon any data structures that might be destroyed before the callback function is successfully unregistered.

The callback handling function must return one of the following values:

- \texttt{DDI_SUCCESS} if it correctly handled the action
- \texttt{DDI_FAILURE} if it encountered an internal error
- \texttt{DDI_ENOTSUP} if it received an unrecognized action

\section*{Interrupt Request Interfaces}

A driver must use the following interfaces to request interrupt vectors from the system.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|l|}
\hline
\textbf{Interface} & \textbf{Data Structures} & \textbf{Description} \\
\hline
\texttt{ddi_intr_alloc()} & \texttt{ddi_intr_handle_t} & Allocate interrupts. \\
\texttt{ddi_intr_set_nreq()} & & Change number of interrupt vectors requested. \\
\hline
\end{tabular}
\end{table}

\subsection*{Allocate an Interrupt}

Use the \texttt{ddi_intr_alloc(9F)} function to initially allocate interrupts.
The `ddi_intr_alloc()` function allows a driver to request a set of interrupt vectors for a device. Before calling this function, the driver must allocate an empty handle array large enough to contain the number of interrupts requested. The `ddi_intr_alloc()` function attempts to allocate the requested number of interrupt handles and initialize the array with the assigned interrupt vectors beginning at the offset specified by the `inum` parameter. The `actualp` parameter returns the actual number of interrupt vectors that were allocated.

A driver can use the `ddi_intr_alloc()` function in two ways:

- The driver can call the `ddi_intr_alloc()` function multiple times to allocate interrupt vectors to individual members of the interrupt handle array in separate steps.
- The driver can call the `ddi_intr_alloc()` function one time to allocate all of the interrupt vectors for the device at once.

If you are using the Interrupt Resource Management feature, call `ddi_intr_alloc()` one time to allocate all interrupt vectors at once. The `count` parameter is the total number of interrupt vectors requested by the driver. If the value in `actualp` is less than the value of `count`, then the system is not able to fulfill the request completely. The Interrupt Resource Management feature saves this request (`count` becomes `nreq` - see below) and might be able to allocate more interrupt vectors to this driver at a later time.

**Note** – When you use the Interrupt Resource Management feature, additional calls to `ddi_intr_alloc()` do not change the total number of interrupt vectors requested. Use the `ddi_intr_set_nreq(9F)` function to change the number of interrupt vectors requested.

### Modify Number of Interrupt Vectors Requested

Use the `ddi_intr_set_nreq(9F)` function to change the number of interrupt vectors requested.

```
int ddi_intr_set_nreq (dev_info_t *dip, int nreq);
```

When the Interrupt Resource Management feature is available, a driver can use the `ddi_intr_set_nreq()` function to dynamically adjust the total number of interrupt vectors requested. The driver might do this in response to the actual load that exists once the driver is attached.

A driver must first call `ddi_intr_alloc(9F)` to request an initial number of interrupt vectors. Any time after the `ddi_intr_alloc()` call, the driver can call `ddi_intr_set_nreq()` to change its request size. The specified `nreq` value is the driver's new total number of requested interrupt vectors. The Interrupt Resource Management feature might rebalance the number of interrupts allocated to each driver in the system in response to this new request. Whenever the Interrupt
Resource Management feature rebalances the number of interrupts allocated to drivers, each affected driver receives a callback notification that more or fewer interrupt vectors are available for the driver to use.

A driver might dynamically adjust its total number of requested interrupt vectors if, for example, it uses interrupts in conjunction with specific transactions that it is processing. A storage driver might associate a DMA engine with each ongoing transaction, thus requiring interrupt vectors for that reason. A driver might make calls to `ddi_intr_set_nreq()` in its `open(9F)` and `close(9F)` routines to scale its interrupt usage in response to actual use of the driver.

**Interrupt Usage and Flexibility**

A driver for a device that supports many different interruptible conditions must be able to map those conditions to an arbitrary number of interrupt vectors. The driver cannot assume that interrupt vectors that are allocated will remain available. Some currently available interrupts might later be taken back by the system to accommodate the needs of other drivers in the system.

A driver must be able to:

- Determine how many interrupts its hardware supports.
- Determine how many interrupts are appropriate to use. For example, the total number of processors in the system might affect this evaluation.
- Compare the number of interrupts needed with the number of interrupts available at any given time.

In summary, the driver must be able to select a mixture of interrupt handling functions and program its hardware to generate interrupts according to need and interrupt availability. In some cases multiple interrupts might be targeted to the same vector, and the interrupt handler for that interrupt vector must determine which interrupts occurred. The performance of the device can be affected by how well the driver maps interrupts to interrupt vectors.

**Example Implementation of Interrupt Resource Management**

One type of device driver that is an excellent candidate for interrupt resource management is a network device driver. The network device hardware supports multiple transmit and receive channels.

The network device generates a unique interrupt condition whenever the device receives a packet on one of its receive channels or transmits a packet on one of its transmit channels. The hardware can send a specific MSI-X interrupt for each event that can occur. A table in the hardware determines which MSI-X interrupt to generate for each event.
To optimize performance, the driver requests enough interrupts from the system to give each separate interrupt its own interrupt vector. The driver makes this request when it first calls `ddi_intr_alloc(9F)` in its `attach(9F)` routine.

The driver then evaluates the actual number of interrupts it received from `ddi_intr_alloc()` in `actualp`. It might receive all the interrupts it requested, or it might receive fewer interrupts.

A separate function inside the driver uses the total number of available interrupts to calculate which MSI-X interrupts to generate for each event. This function programs the table in the hardware accordingly.

- If the driver receives all of its requested interrupt vectors, each entry in the hardware table has its own unique MSI-X interrupt. A one-to-one mapping exists between interrupt conditions and interrupt vectors. The hardware generates a unique MSI-X interrupt for each type of event.
- If the driver has fewer interrupt vectors available, some MSI-X interrupt numbers must appear multiple times in the hardware table. The hardware generates the same MSI-X interrupt for more than one type of event.

The driver should have two different interrupt handler functions.

- One interrupt handler function performs a specific task in response to an interrupt. This simple function handles interrupts that are generated by only one of the possible hardware events.
- A second interrupt handler function is more complicated. This function handles the case where multiple interrupts are mapped to the same MSI-X interrupt vector.

In the example driver in this section, the function `xx_setup_interrupts()` uses the number of available interrupt vectors to program the hardware and calls the appropriate interrupt handler for each of those interrupt vectors. The `xx_setup_interrupts()` function is called in two places: after `ddi_intr_alloc()` is called in `xx_attach()`, and after interrupt vector allocations are adjusted in the `xx_cbfunc()` callback handler function.

```c
int
xx_setup_interrupts(xx_state_t *statep, int navail, xx_intrs_t *xx_intrs_p);
```

The `xx_setup_interrupts()` function is called with an array of `xx_intrs_t` data structures.

```c
typedef struct {
    ddi_intr_handler_t inthandler;
    void *arg1;
    void *arg2;
} xx_intrs_t;
```

This `xx_setup_interrupts()` functionality must exist in the driver independent of whether the Interrupt Resource Management feature is available. Drivers must be able to function with fewer interrupt vectors than the number requested during `attach`. If the Interrupt Resource Management feature is available, you can modify the driver to dynamically adjust to a new number of available interrupt vectors.
Other functionality that the driver must provide independent of whether the Interrupt Resource Management feature is available includes the ability to quiesce the hardware and resume the hardware. Quiesce and resume are needed for certain events related to power management and hotplugging. Quiesce and resume also are required to handle interrupt callback actions.

The quiesce function is called in `xx_detach()`.

```c
int xx_quiesce(xx_state_t *statep);
```

The resume function is called in `xx_attach()`.

```c
int xx_resume(xx_state_t *statep);
```

Make the following modifications to enhance this device driver to use the Interrupt Resource Management feature:

- Register a callback handler. The driver must register for the actions that indicate when fewer or more interrupts are available.
- Handle callbacks. The driver must quiesce its hardware, reprogram its interrupt handling, and resume its hardware in response to each such callback action.

```c
/*@
 * attach(9F) routine.
 *
 * Creates soft state, registers callback handler, initializes
 * hardware, and sets up interrupt handling for the driver.
 *
*/
xx_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    xx_state_t *statep = NULL;
    xx_intr_t *intrs = NULL;
    ddi_intr_handle_t *hdls;
    ddi_cb_handle_t cb_hdl;
    int instance;
    int type;
    int types;
    int nintrs;
    int nactual;
    int inum;

    /* Get device instance */
    instance = ddi_get_instance(dip);

    switch (cmd) {
    case DDI_ATTACH:
        /* Get soft state */
        if (ddi_soft_state_zalloc(state_list, instance) != 0)
            return (DDI_FAILURE);
        statep = ddi_get_soft_state(state_list, instance);
```
ddi_set_driver_private(dip, (caddr_t)statep);
statep->dip = dip;

/* Initialize hardware */
xx_initialize(statep);

/* Register callback handler */
if (ddi_cb_register(dip, DDI_CB_FLAG_INTR, xx_cbfunc,
        statep, NULL, &cb_hdl) != 0) {
    ddi_soft_state_free(state_list, instance);
    return (DDI_FAILURE);
}
statep->cb_hdl = cb_hdl;

/* Select interrupt type */
ddi_intr_get_supported_types(dip, &types);
if (types & DDI_INTR_TYPE_MSIX) {
    type = DDI_INTR_TYPE_MSIX;
} else if (types & DDI_INTR_TYPE_MSI) {
    type = DDI_INTR_TYPE_MSI;
} else {
    type = DDI_INTR_TYPE_FIXED;
}
statep->type = type;

/* Get number of supported interrupts */
ddi_intr_get_nintrs(dip, type, &nintrs);

/* Allocate interrupt handle array */
statep->hdls_size = nintrs * sizeof (ddi_intr_handle_t);
statep->hdls = kmalloc(statep->hdls_size, KM_SLEEP);

/* Allocate interrupt setup array */
statep->intrs_size = nintrs * sizeof (xx_intr_t);
statep->intrs = kmalloc(statep->intrs_size, KM_SLEEP);

/* Allocate interrupt vectors */
ddi_intr_alloc(dip, hdls, type, 0, nintrs, &nactual, 0);
statep->nactual = nactual;

/* Configure interrupt handling */
xx_setup_interrupts(statep, statep->nactual, statep->intrs);

/* Install and enable interrupt handlers */
for (inum = 0; inum < nactual; inum++) {
    ddi_intr_add_handler(&hdls[inum],
            intrs[inum].inhandler,
            intrs[inum].arg1, intrs[inum].arg2);
    ddi_intr_enable(hdls[inum]);
}
break;

break:

/* Get soft state */
statep = ddi_get_soft_state(state_list, instance);
if (statep == NULL)
    return (DDI_FAILURE);
/* Resume hardware */
xx_resume(statep);

break;
}

return (DDI_SUCCESS);

/* detach()F routine.
* Stops the hardware, disables interrupt handling, unregisters
* a callback handler, and destroys the soft state for the driver.
*/
xx_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
    xx_state_t *statep = NULL;
    int instance;
    int inum;

    /* Get device instance */
    instance = ddi_get_instance(dip);

    switch (cmd) {
    case DDI_DETACH:
        /* Get soft state */
        statep = ddi_get_soft_state(state_list, instance);
        if (statep == NULL)
            return (DDI_FAILURE);

        /* Stop device */
        xx_uninitialize(statep);

        /* Disable and free interrupts */
        for (inum = 0; inum < statep->nactual; inum++) {
            ddi_intr_disable(statep->hdls[inum]);
            ddi_intr_remove_handler(statep->hdls[inum]);
            ddi_intr_free(statep->hdls[inum]);
        }

        /* Unregister callback handler */
        ddi_cb_unregister(statep->cb_hdl);

        /* Free interrupt handle array */
        kmem_free(statep->hdls, statep->hdls_size);

        /* Free interrupt setup array */
        kmem_free(statep->intrs, statep->intrs_size);

        /* Free soft state */
        ddi_soft_state_free(state_list, instance);
        break;
    
    case DDI_SUSPEND:
/* Get soft state */
statep = ddi_get_soft_state(state_list, instance);
if (statep == NULL)
    return (DDI_FAILURE);

/* Suspend hardware */
xx_quiesce(statep);
break;
}

return (DDI_SUCCESS);

/* (**ddi_cbfunc()) routine. */
/* Adapt interrupt usage when availability changes. */
int
xx_cbfunc(dev_info_t *dip, ddi_cb_action_t cbaction, void *cbarg,
          void *arg1, void *arg2)
{
    xx_state_t *statep = (xx_state_t *)arg1;
    int count;
    int inum;
    int nactual;

    switch (cbaction) {
    case DDI_CB_INTR_ADD:
    case DDI_CB_INTR_REMOVE:
        /* Get change in availability */
        count = (int)(uintptr_t)cbarg;
        /* Suspend hardware */
        xx_quiesce(statep);
        /* Tear down previous interrupt handling */
        for (inum = 0; inum < statep->nactual; inum++) {
            ddi_intr_disable(statep->hdls[inum]);
            ddi_intr_remove_handler(statep->hdls[inum]);
        }
        /* Adjust interrupt vector allocations */
        if (cbaction == DDI_CB_INTR_ADD) {
            /* Allocate additional interrupt vectors */
            ddi_intr_alloc(dip, statep->hdls, statep->type,
                statep->nactual, count, &nactual, 0);
            /* Update actual count of available interrupts */
            statep->nactual += nactual;
        } else {
            /* Free removed interrupt vectors */
            for (inum = statep->nactual - count;
                inum < statep->nactual; inum++) {
                ddi_intr_disable(statep->hdls[inum]);
                ddi_intr_remove_handler(statep->hdls[inum]);
            }
        }
    default:
        break;
}
return (DDI_SUCCESS);
ddi_intr_free(statep->hdls[inum]);
}

/* Update actual count of available interrupts */
statep->nactual -= count;
}

/* Configure interrupt handling */
xx_setup_interrupts(statep, statep->nactual, statep->intrs);

/* Install and enable interrupt handlers */
for (inum = 0; inum < statep->nactual; inum++) {
  ddi_intr_add_handler(&statep->hdls[inum],
                      statep->intrs[inum].inthandler,
                      statep->intrs[inum].arg1,
                      statep->intrs[inum].arg2);
  ddi_intr_enable(statep->hdls[inum]);
}

/* Resume hardware */
xx_resume(statep);
break;

default:
  return (DDI_ENOTSUP);
}

return (DDI_SUCCESS);

Interrupt Handler Functionality

The driver framework and the device each place demands on the interrupt handler. All
interrupt handlers are required to do the following tasks:

- **Determine whether the device is interrupting and possibly reject the interrupt.**
  The interrupt handler first examines the device to determine whether this device issued the
  interrupt. If this device did not issue the interrupt, the handler must return
  DDI_INTR_UNCLAIMED. This step enables the implementation of device polling. Any device at
  the given interrupt priority level might have issued the interrupt. Device polling tells the
  system whether this device issued the interrupt.

- **Inform the device that the device is being serviced.**
  Informing a device about servicing is a device-specific operation that is required for the
  majority of devices. For example, SBus devices are required to interrupt until the driver tells
  the SBus devices to stop. This approach guarantees that all SBus devices that interrupt at the
  same priority level are serviced.

- **Perform any I/O request-related processing.**
Devices interrupt for different reasons, such as transfer done or transfer error. This step can involve using data access functions to read the device’s data buffer, examine the device’s error register, and set the status field in a data structure accordingly. Interrupt dispatching and processing are relatively time consuming.

- Do any additional processing that could prevent another interrupt.
  
  For example, read the next item of data from the device.

- Return DDI_INTR_CLAIMED.

- MSI interrupts must always be claimed.
  
  Claiming an interrupt is optional for MSI-X interrupts. In either case, the ownership of the interrupt need not be checked, because MSI and MSI-X interrupts are not shared with other devices.

- Drivers that support hotplugging and multiple MSI or MSI-X interrupts should retain a separate interrupt for hotplug events and register a separate ISR (interrupt service routine) for that interrupt.

The following example shows an interrupt routine for a device called mydev.

**EXAMPLE 8–9 Interrupt Example**

```c
static uint_t
mydev_intr(caddr_t arg1, caddr_t arg2)
{
    struct mydevstate *xsp = (struct mydevstate *)arg1;
    uint8_t t status;
    volatile uint8_t temp;

    /* Claim or reject the interrupt. This example assumes
     * that the device’s CSR includes this information.
     */
    mutex_enter(&xsp->high_mu);

    /* use data access routines to read status */
    status = ddi_get8(xsp->data_access_handle, &xsp->regp->csr);
    if (!(status & INTERRUPTING)) {
        mutex_exit(&xsp->high_mu);
        return (DDI_INTR_UNCLAIMED); /* dev not interrupting */
    }

    /* Inform the device that it is being serviced, and re-enable
     * interrupts. The example assumes that writing to the
     * CSR accomplishes this. The driver must ensure that this data
     * access operation makes it to the device before the interrupt
     * service routine returns. For example, using the data access
     * functions to read the CSR, if it does not result in unwanted
     * effects, can ensure this.
     */
    ddi_put8(xsp->data_access_handle, &xsp->regp->csr,
             CLEAR_INTERRUPT | ENABLE_INTERRUPTS);

    /* flush store buffers */
```
temp = ddi_get8(xsp->data_access_handle, &xsp->regp->csr);
mutex_exit(&xsp->mu);
return (DDI_INTR_CLAIMED);
}

Most of the steps performed by the interrupt routine depend on the specifics of the device itself. Consult the hardware manual for the device to determine the cause of the interrupt, detect error conditions, and access the device data registers.

Handling High-Level Interrupts

High-level interrupts are those interrupts that interrupt at the level of the scheduler and above. This level does not allow the scheduler to run. Therefore, high-level interrupt handlers cannot be preempted by the scheduler. High-level interrupts cannot block because of the scheduler. High-level interrupts can only use mutual exclusion locks for locking.

The driver must determine whether the device is using high-level interrupts. Do this test in the driver's attach(9E) entry point when you register interrupts. See “High-Level Interrupt Handling Example” on page 147.

- If the interrupt priority returned from ddi_intr_get_pri(9F) is greater than or equal to the priority returned from ddi_intr_get_hilevel_pri(9F), the driver can fail to attach, or the driver can implement a high-level interrupt handler. The high-level interrupt handler uses a lower-priority software interrupt to handle the device. To allow more concurrency, use a separate mutex to protect data from the high-level handler.
- If the interrupt priority returned from ddi_intr_get_pri(9F) is less than the priority returned from ddi_intr_get_hilevel_pri(9F), the attach(9E) entry point falls through to regular interrupt registration. In this case, a soft interrupt is not necessary.

High-Level Mutexes

A mutex initialized with an interrupt priority that represents a high-level interrupt is known as a high-level mutex. While holding a high-level mutex, the driver is subject to the same restrictions as a high-level interrupt handler.

High-Level Interrupt Handling Example

In the following example, the high-level mutex (xsp->high_mu) is used only to protect data shared between the high-level interrupt handler and the soft interrupt handler. The protected
data includes a queue used by both the high-level interrupt handler and the low-level handler, and a flag that indicates that the low-level handler is running. A separate low-level mutex (xsp->low_mu) protects the rest of the driver from the soft interrupt handler.

**EXAMPLE 8–10** Handling High-Level Interrupts With attach()

```c
static int
mydevattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    struct mydevstate *xsp;
    /* ... */

    ret = ddi_intr_get_supported_types(dip, &type);
    if ((ret != DDI_SUCCESS) || (!((type & DDI_INTR_TYPE_FIXED)))) {
        cmn_err(CE_WARN, "ddi_intr_get_supported_types() failed");
        return (DDI_FAILURE);
    }

    ret = ddi_intr_get_nintrs(dip, DDI_INTR_TYPE_FIXED, &count);
    /*
     * Fixed interrupts can only have one interrupt. Check to make
     * sure that number of supported interrupts and number of
     * available interrupts are both equal to 1.
     */
    if ((ret != DDI_SUCCESS) || (count != 1)) {
        cmn_err(CE_WARN, "No fixed interrupts found");
        return (DDI_FAILURE);
    }

    xsp->xs_htable = kmem_zalloc(count * sizeof (ddi_intr_handle_t),
                               KM_SLEEP);
    ret = ddi_intr_alloc(dip, xsp->xs_htable, DDI_INTR_TYPE_FIXED, 0,
                         count, &actual, 0);
    if ((ret != DDI_SUCCESS) || (actual != 1)) {
        cmn_err(CE_WARN, "ddi_intr_alloc failed 0x%x", ret);
        kmem_free(xsp->xs_htable, sizeof (ddi_intr_handle_t));
        return (DDI_FAILURE);
    }

    ret = ddi_intr_get_pri(xsp->xs_htable[0], &intr_pri);
    if (ret != DDI_SUCCESS) {
        cmn_err(CE_WARN, "ddi_intr_get_pri failed 0x%x", ret);
        (void) ddi_intr_free(xsp->xs_htable[0]);
        kmem_free(xsp->xs_htable, sizeof (ddi_intr_handle_t));
        return (DDI_FAILURE);
    }

    if (intr_pri >= ddi_intr_get_hilevel_pri()) {
        mutex_init(&xsp->high_mu, NULL, MUTEX_DRIVER,
                   DDI_INTR_PRI(intr_pri));

        ret = ddi_intr_add_handler(xsp->xs_htable[0],
                                    mydevhigh_intr, (caddr_t)xsp, NULL);
    }
```
EXAMPLE 8–10  Handling High-Level Interrupts With attach()  (Continued)

if (ret != DDI_SUCCESS) {
  cmn_err(CE_WARN, "ddi_intr_add_handler failed 0x%x", ret);
  mutex_destroy(&xsp->xs_int_mutex);
  (void) ddi_intr_free(xsp->xs_htable[0]);
  kmem_free(xsp->xs_htable, sizeof (ddi_intr_handle_t));
  return (DDI_FAILURE);
}

/* add soft interrupt */
if (ddi_intr_add_softint(xsp->xs_dip, &xsp->xs_softint_hdl, 
                        DDI_INTR_SOFTPRI_MAX, xs_soft_intr, (caddr_t)xsp) !=
    DDI_SUCCESS) {
  cmn_err(CE_WARN, "add soft interrupt failed");
  mutex_destroy(&xsp->high_mu);
  (void) ddi_intr_remove_handler(xsp->xs_htable[0]);
  (void) ddi_intr_free(xsp->xs_htable[0]);
  kmem_free(xsp->xs_htable, sizeof (ddi_intr_handle_t));
  return (DDI_FAILURE);
}

xsp->low_soft_pri = DDI_INTR_SOFTPRI_MAX;
mutex_init(&xsp->low_mu, NULL, MUTEX_DRIVER,
            DDI_INTR_PRI(xsp->low_soft_pri));
} else {
    /* regular interrupt registration continues from here */
    return (DDI_SUCCESS);
}

The high-level interrupt routine services the device and queues the data. The high-level routine triggers a software interrupt if the low-level routine is not running, as the following example demonstrates.

EXAMPLE 8–11  High-level Interrupt Routine

static uint_t
mydevhigh_intr(caddr_t arg1, caddr_t arg2)
{
  struct mydevstate  *xsp = (struct mydevstate *)arg1;
  uint8_t t status;
  volatile uint8_t t temp;
  int need_softint;

  mutex_enter(&xsp->high_mu);
  /* read status */
  status = ddi_get8(xsp->data_access_handle, &xsp->regp->csr);
  if (!((status & INTERRUPTING) { 
    mutex_exit(&xsp->high_mu);
  }

Chapter 8 • Interrupt Handlers  149
The low-level interrupt routine is started by the high-level interrupt routine, which triggers a software interrupt. The low-level interrupt routine runs until there is nothing left to process, as the following example shows.

```c
EXAMPLE 8-12   Low-Level Soft Interrupt Routine

static uint_t
mydev_soft_intr(caddr_t arg1, caddr_t arg2)
{
    struct mydevstate *mydevp = (struct mydevstate *)arg1;
    /* ... */
    mutex_enter(&mydevp->low_mu);
    mutex_enter(&mydevp->high_mu);
    if (mydevp->softint_count > 1) {
        mydevp->softint_count--;
        mutex_exit(&mydevp->high_mu);
        mutex_exit(&mydevp->low_mu);
        return (DDI_INTR_CLAIMED);
    }
    if (/* queue empty */ ) {
        mutex_exit(&mydevp->high_mu);
        mutex_exit(&mydevp->low_mu);
    }
    return (DDI_INTR_CLAIMED);
}
```
EXAMPLE 8–12  Low-Level Soft Interrupt Routine   (Continued)

    return (DDI_INTR_UNCLAIMED);
  }

mydevp->softint_running = 1;
while (EMBEDDED COMMENT:data on queue) {
  ASSERT(mutex owned(&mydevp->high_mu);
  /* Dequeue data from high-level queue. */
  mutex_exit(&mydevp->high_mu);
  /* normal interrupt processing */
  mutex_enter(&mydevp->high_mu);
}

mydevp->softint_running = 0;
mydevp->softint_count = 0;
mutex_exit(&mydevp->high_mu);
mutex_exit(&mydevp->low_mu);
return (DDI_INTR_CLAIMED);
Many devices can temporarily take control of the bus. These devices can perform data transfers that involve main memory and other devices. Because the device is doing the work without the help of the CPU, this type of data transfer is known as direct memory access (DMA). The following types of DMA transfers can be performed:

- Between two devices
- Between a device and memory
- Between memory and memory

This chapter explains transfers between a device and memory only. The chapter provides information on the following subjects:

- "DMA Model" on page 153
- "Types of Device DMA" on page 154
- "Types of Host Platform DMA" on page 155
- "DMA Software Components: Handles, Windows, and Cookies” on page 155
- "DMA Operations” on page 156
- “Managing DMA Resources” on page 161
- "DMA Windows” on page 173

**DMA Model**

The Oracle Solaris Device Driver Interface/Driver-Kernel Interface (DDI/DKI) provides a high-level, architecture-independent model for DMA. This model enables the framework, that is, the DMA routines, to hide architecture-specific details such as the following:

- Setting up DMA mappings
- Building scatter-gather lists
- Ensuring that I/O and CPU caches are consistent

Several abstractions are used in the DDI/DKI to describe aspects of a DMA transaction:

- **DMA object** – Memory that is the source or destination of a DMA transfer.
- **DMA handle** – An opaque object returned from a successful `ddi_dma_alloc_handle(9F)` call. The DMA handle can be used in subsequent DMA subroutine calls to refer to such DMA objects.

- **DMA cookie** – A `ddi_dma_cookie(9S)` structure (`ddi_dma_cookie_t`) describes a contiguous portion of a DMA object that is entirely addressable by the device. The cookie contains DMA addressing information that is required to program the DMA engine.

Rather than map an object directly into memory, device drivers allocate DMA resources for a memory object. The DMA routines then perform any platform-specific operations that are needed to set up the object for DMA access. The driver receives a DMA handle to identify the DMA resources that are allocated for the object. This handle is opaque to the device driver. The driver must save the handle and pass the handle in subsequent calls to DMA routines. The driver should not interpret the handle in any way.

Operations that provide the following services are defined on a DMA handle:

- Manipulating DMA resources
- Synchronizing DMA objects
- Retrieving attributes of the allocated resources

### Types of Device DMA

Devices perform one of the following three types of DMA:

- Bus-master DMA
- Third-party DMA
- First-party DMA

### Bus-Master DMA

The driver should program the device’s DMA registers directly in cases where the device acts like a true bus master. For example, a device acts like a bus master when the DMA engine resides on the device board. The transfer address and count are obtained from the DMA cookie to be passed on to the device.

### Third-Party DMA

Third-party DMA uses a system DMA engine resident on the main system board, which has several DMA channels that are available for use by devices. The device relies on the system’s DMA engine to perform the data transfers between the device and memory. The driver uses DMA engine routines (see the `ddi_dmae(9F)` function) to initialize and program the DMA engine. For each DMA data transfer, the driver programs the DMA engine and then gives the device a command to initiate the transfer in cooperation with that engine.
First-Party DMA

Under first-party DMA, the device uses a channel from the system’s DMA engine to drive that device’s DMA bus cycles. Use the `ddi_dmae_1stparty(9F)` function to configure this channel in a cascade mode so that the DMA engine does not interfere with the transfer.

Types of Host Platform DMA

The platform on which the device operates provides either direct memory access (DMA) or direct virtual memory access (DVMA).

On platforms that support DMA, the system provides the device with a physical address in order to perform transfers. In this case, the transfer of a DMA object can actually consist of a number of physically discontinuous transfers. An example is when an application transfers a buffer that spans several contiguous virtual pages that map to physically discontinuous pages. To deal with the discontinuous memory, devices for these platforms usually have some kind of scatter-gather DMA capability. Typically, x86 systems provide physical addresses for direct memory transfers.

On platforms that support DVMA, the system provides the device with a virtual address to perform transfers. In this case, memory management units (MMU) provided by the underlying platform translate device accesses to these virtual addresses into the proper physical addresses. The device transfers to and from a contiguous virtual image that can be mapped to discontinuous physical pages. Devices that operate in these platforms do not need scatter-gather DMA capability. Typically, SPARC platforms provide virtual addresses for direct memory transfers.

DMA Software Components: Handles, Windows, and Cookies

A DMA handle is an opaque pointer that represents an object, usually a memory buffer or address. A DMA handle enables a device to perform DMA transfers. Several different calls to DMA routines use the handle to identify the DMA resources that are allocated for the object.

An object represented by a DMA handle is completely covered by one or more DMA cookies. A DMA cookie represents a contiguous piece of memory that is used in data transfers by the DMA engine. The system divides objects into multiple cookies based on the following information:

- The `ddi_dma_attr(9S)` attribute structure provided by the driver
- Memory location of the target object
- Alignment of the target object

If an object does not fit within the limitations of the DMA engine, that object must be broken into multiple DMA windows. You can only activate and allocate resources for one window at a
time. Use the `ddi_dma_getwin(9F)` function to position between windows within an object. Each DMA window consists of one or more DMA cookies. For more information, see “DMA Windows” on page 173.

Some DMA engines can accept more than one cookie. Such engines perform scatter-gather I/O without the help of the system. If multiple cookies are returned from a bind, the driver should call `ddi_dma_nextcookie(9F)` repeatedly to retrieve each cookie. These cookies must then be programmed into the engine. The device can then be programmed to transfer the total number of bytes covered by the aggregate of these DMA cookies.

**DMA Operations**

The steps in a DMA transfer are similar among the types of DMA. The following sections present methods for performing DMA transfers.

---

**Note** – You do not need to ensure that the DMA object is locked in memory in block drivers for buffers that come from the file system. The file system has already locked the data in memory.

---

**Performing Bus-Master DMA Transfers**

The driver should perform the following steps for bus-master DMA.

1. Describe the DMA attributes. This step enables the routines to ensure that the device is able to access the buffer.
2. Allocate a DMA handle.
3. Ensure that the DMA object is locked in memory. See the `physio(9F)` or `ddi_umem_lock(9F)` man page.
4. Allocate DMA resources for the object.
5. Program the DMA engine on the device.
6. Start the engine.
7. When the transfer is complete, continue the bus master operation.
8. Perform any required object synchronizations.
9. Release the DMA resources.
10. Free the DMA handle.
Performing First-Party DMA Transfers

The driver should perform the following steps for first-party DMA.

1. Allocate a DMA channel.
2. Use `ddi_dmae_1stparty(9F)` to configure the channel.
3. Ensure that the DMA object is locked in memory. See the `physio(9F)` or `ddi_umem_lock(9F)` man page.
4. Allocate DMA resources for the object.
5. Program the DMA engine on the device.
6. Start the engine.
7. When the transfer is complete, continue the bus-master operation.
8. Perform any required object synchronizations.
9. Release the DMA resources.
10. Deallocate the DMA channel.

Performing Third-Party DMA Transfers

The driver should perform these steps for third-party DMA.

1. Allocate a DMA channel.
2. Retrieve the system's DMA engine attributes with `ddi_dmae_getattr(9F)`.
3. Lock the DMA object in memory. See the `physio(9F)` or `ddi_umem_lock(9F)` man page.
4. Allocate DMA resources for the object.
5. Use `ddi_dmae_prog(9F)` to program the system DMA engine to perform the transfer.
6. Perform any required object synchronizations.
7. Use `ddi_dmae_stop(9F)` to stop the DMA engine.
8. Release the DMA resources.
9. Deallocate the DMA channel.

Certain hardware platforms restrict DMA capabilities in a bus-specific way. Drivers should use `ddi_slaveonly(9F)` to determine whether the device is in a slot in which DMA is possible.

DMA Attributes

DMA attributes describe the attributes and limits of a DMA engine, which include:

- Limits on addresses that the device can access
- Maximum transfer count
- Address alignment restrictions
A device driver must inform the system about any DMA engine limitations through the `ddi_dma_attr(9S)` structure. This action ensures that DMA resources that are allocated by the system can be accessed by the device's DMA engine. The system can impose additional restrictions on the device attributes, but the system never removes any of the driver-supplied restrictions.

### `ddi_dma_attr` Structure
The DMA attribute structure has the following members:

```c
typedef struct ddi_dma_attr {
    uint_t dma_attr_version; /* version number */
    uint64_t dma_attr_addr_lo; /* low DMA address range */
    uint64_t dma_attr_addr_hi; /* high DMA address range */
    uint64_t dma_attr_count_max; /* DMA counter register */
    uint64_t dma_attr_align; /* DMA address alignment */
    uint_t dma_attr_burstsizes; /* DMA burst sizes */
    uint32_t dma_attr_minxfer; /* min effective DMA size */
    uint64_t dma_attr_maxxfer; /* max DMA xfer size */
    uint64_t dma_attr_seg; /* segment boundary */
    int dma_attr_sgllen; /* s/g length */
    uint32_t dma_attr_granular; /* granularity of device */
    uint_t dma_attr_flags; /* Bus specific DMA flags */
} ddi_dma_attr_t;
```

where:

- **dma_attr_version**: Version number of the attribute structure. `dma_attr_version` should be set to DMA_ATTR_V0.
- **dma_attr_addr_lo**: Lowest bus address that the DMA engine can access.
- **dma_attr_addr_hi**: Highest bus address that the DMA engine can access.
- **dma_attr_count_max**: Specifies the maximum transfer count that the DMA engine can handle in one cookie. The limit is expressed as the maximum count minus one. This count is used as a bit mask, so the count must also be one less than a power of two.
- **dma_attr_align**: Specifies alignment requirements when allocating memory from `ddi_dma_mem_alloc(9F)`. An example of an alignment requirement is alignment on a page boundary. The `dma_attr_align` field is used only when allocating memory. This field is ignored during bind operations. For bind operations, the driver must ensure that the buffer is aligned appropriately.
- **dma_attr_burstsizes**: Specifies the burst sizes that the device supports. A burst size is the amount of data the device can transfer before relinquishing the bus. This member is a binary encoding of burst sizes, which are assumed to be powers of two. For example, if the device is capable of doing...
1-byte, 2-byte, 4-byte, and 16-byte bursts, this field should be set to 0x17. The system also uses this field to determine alignment restrictions.

**dma_attr_minxfer**
Minimum effective transfer size that the device can perform. This size also influences restrictions on alignment and on padding.

**dma_attr_maxxfer**
Describes the maximum number of bytes that the DMA engine can accommodate in one I/O command. This limitation is only significant if dma_attr_maxxfer is less than (dma_attr_count_max + 1) * dma_attr_sgllen.

**dma_attr_seg**
Upper bound of the DMA engine's address register. dma_attr_seg is often used where the upper 8 bits of an address register are a latch that contains a segment number. The lower 24 bits are used to address a segment. In this case, dma_attr_seg would be set to 0xFFFFFFFF, which prevents the system from crossing a 24-bit segment boundary when allocating resources for the object.

**dma_attr_sgllen**
Specifies the maximum number of entries in the scatter-gather list. dma_attr_sgllen is the number of cookies that the DMA engine can consume in one I/O request to the device. If the DMA engine has no scatter-gather list, this field should be set to 1.

**dma_attr_granular**
This field gives the granularity in bytes of the DMA transfer ability of the device. An example of how this value is used is to specify the sector size of a mass storage device. When a bind operation requires a partial mapping, this field is used to ensure that the sum of the sizes of the cookies in a DMA window is a whole multiple of granularity. However, if the device does not have a scatter-gather capability, it is impossible for the DDI to ensure the granularity. For this case, the value of the dma_attr_granular field should be 1.

**dma_attr_flags**
This field can be set to DDI_DMA_FORCE_PHYSICAL, which indicates that the system should return physical rather than virtual I/O addresses if the system supports both. If the system does not support physical DMA, the return value from ddi_dma_alloc_handle(9F) is DDI_DMA_BADATTR. In this case, the driver has to clear DDI_DMA_FORCE_PHYSICAL and retry the operation.

**SBus Example**
A DMA engine on an SBus in a SPARC machine has the following attributes:

- Access to addresses ranging from 0xFF000000 to 0xFFFFFFFF only
- 32-bit DMA counter register
- Ability to handle byte-aligned transfers
- Support for 1-byte, 2-byte, and 4-byte burst sizes
- Minimum effective transfer size of 1 byte
- 32-bit address register
- No scatter-gather list
- Operation on sectors only, for example, a disk

A DMA engine on an SBus in a SPARC machine has the following attribute structure:

```c
static ddi_dma_attr_t attributes = {
    DMA_ATTR_V0, /* Version number */
    0xFFFFFFFF, /* low address */
    0xFFFFFFFF, /* high address */
    0xFFFFFFFF, /* counter register max */
    1, /* byte alignment */
    0x7, /* burst sizes: 0x1 | 0x2 | 0x4 */
    0x1, /* minimum transfer size */
    0xFFFFFFFF, /* max transfer size */
    0xFFFFFFFF, /* address register max */
    1, /* no scatter-gather */
    512, /* device operates on sectors */
    0, /* attr flag: set to 0 */
};
```

**ISA Bus Example**

A DMA engine on an ISA bus in an x86 machine has the following attributes:

- Access to the first 16 megabytes of memory only
- Inability to cross a 1-megabyte boundary in a single DMA transfer
- 16-bit counter register
- Ability to handle byte-aligned transfers
- Support for 1-byte, 2-byte, and 4-byte burst sizes
- Minimum effective transfer size of 1 byte
- Ability to hold up to 17 scatter-gather transfers
- Operation on sectors only, for example, a disk

A DMA engine on an ISA bus in an x86 machine has the following attribute structure:

```c
static ddi_dma_attr_t attributes = {
    DMA_ATTR_V0, /* Version number */
    0x00000000, /* low address */
    0x00FFFFFF, /* high address */
    0xFFFF, /* counter register max */
    1, /* byte alignment */
    0x7, /* burst sizes */
    0x1, /* minimum transfer size */
    0xFFFFFFFF, /* max transfer size */
    0xFFFFFFFF, /* address register max */
    17, /* scatter-gather */
    512, /* device operates on sectors */
};
```
Managing DMA Resources

This section describes how to manage DMA resources.

Object Locking

Before allocating the DMA resources for a memory object, the object must be prevented from moving. Otherwise, the system can remove the object from memory while the device is trying to write to that object. A missing object would cause the data transfer to fail and possibly corrupt the system. The process of preventing memory objects from moving during a DMA transfer is known as locking down the object.

The following object types do not require explicit locking:

- Buffers coming from the file system through strategy(9E). These buffers are already locked by the file system.
- Kernel memory allocated within the device driver, such as that allocated by ddi_dma_mem_alloc(9F).

For other objects such as buffers from user space, physio(9F) or ddi_unem_lock(9F) must be used to lock down the objects. Locking down objects with these functions is usually performed in the read(9E) or write(9E) routines of a character device driver. See “Data Transfer Methods” on page 269 for an example.

Allocating a DMA Handle

A DMA handle is an opaque object that is used as a reference to subsequently allocated DMA resources. The DMA handle is usually allocated in the driver's attach() entry point that uses ddi_dma_alloc_handle(9F). The ddi_dma_alloc_handle() function takes the device information that is referred to by dip and the device’s DMA attributes described by a ddi_dma_attr(9S) structure as parameters. The ddi_dma_alloc_handle() function has the following syntax:

```c
int ddi_dma_alloc_handle(dev_info_t *dip, 
    ddi_dma_attr_t *attr, int (*callback)(caddr_t),
    caddr_t arg, ddi_dma_handle_t *handlep);
```

where:

- dip Pointer to the device’s dev_info structure.
Allocate DMA Resources

Two interfaces allocate DMA resources:

- `ddi_dma_buf_bind_handle(9F)` – Used with `buf(9S)` structures
- `ddi_dma_addr_bind_handle(9F)` – Used with virtual addresses

DMA resources are usually allocated in the driver's `xxstart()` routine, if an `xxstart()` routine exists. See "Asynchronous Data Transfers (Block Drivers)" on page 299 for a discussion of `xxstart()`.

These two interfaces have the following syntax:

```c
int ddi_dma_addr_bind_handle(ddi_dma_handle_t handle, 
   struct as *as, caddr_t addr, 
   size_t len, uint_t flags, int (*callback)(caddr_t), 
   caddr_t arg, ddi_dma_cookie_t *cookiep, uint_t *ccountp);

int ddi_dma_buf_bind_handle(ddi_dma_handle_t handle, 
   struct buf *bp, uint_t flags, 
   int (*callback)(caddr_t), caddr_t arg, 
   ddi_dma_cookie_t *cookiep, uint_t *ccountp);
```

The following arguments are common to both `ddi_dma_addr_bind_handle(9F)` and `ddi_dma_buf_bind_handle(9F):

- `handle` – DMA handle and the object for allocating resources.
- `flags` – Set of flags that indicate the transfer direction and other attributes. `DDI_DMA_READ` indicates a data transfer from device to memory. `DDI_DMA_WRITE` indicates a data transfer from memory to device. See the `ddi_dma_addr_bind_handle(9F)` or `ddi_dma_buf_bind_handle(9F)` man page for a complete discussion of the available flags.
- `callback` – Address of callback function for handling resource allocation failures. See the `ddi_dma_alloc_handle(9F)` man page.
- `arg` – Argument to pass to the callback function.
- `cookiep` – Pointer to the first DMA cookie for this object.
- `ccountp` – Pointer to the number of DMA cookies for this object.
For `ddi_dma_addr_bind_handle(9F)`, the object is described by an address range with the following parameters:

- `as` Pointer to an address space structure. The value of `as` must be `NULL`.
- `addr` Base kernel address of the object.
- `len` Length of the object in bytes.

For `ddi_dma_buf_bind_handle(9F)`, the object is described by a `buf(9S)` structure pointed to by `bp`.

**Device Register Structure**

DMA-capable devices require more registers than were used in the previous examples.

The following fields are used in the device register structure to support DMA-capable device with no scatter-gather support:

```c
uint32_t dma_addr; /* starting address for DMA */
uint32_t dma_size; /* amount of data to transfer */
```

The following fields are used in the device register structure to support DMA-capable devices with scatter-gather support:

```c
struct sglentry {
    uint32_t dma_addr;
    uint32_t dma_size;
} sglist[SGLLEN];

caddr_t iopb_addr; /* When written, informs the device of the next */
/* command’s parameter block address. */
/* When read after an interrupt, contains */
/* the address of the completed command. */
```

**DMA Callback Example**

In Example 9–1, `xxstart()` is used as the callback function. The per-device state structure is used as the argument to `xxstart()`. The `xxstart()` function attempts to start the command. If the command cannot be started because resources are not available, `xxstart()` is scheduled to be called later when resources are available.

Because `xxstart()` is used as a DMA callback, `xxstart()` must adhere to the following rules, which are imposed on DMA callbacks:

- Resources cannot be assumed to be available. The callback must try to allocate resources again.
The callback must indicate to the system whether allocation succeeded. 

DDI_DMA_CALLBACK_RUNOUT should be returned if the callback fails to allocate resources, in which case xxstart() needs to be called again later. DDI_DMA_CALLBACK_DONE indicates success, so that no further callback is necessary.

EXAMPLE 9–1 DMA Callback Example

```c
static int xxstart(caddr_t arg)
{
    struct xxstate *xsp = (struct xxstate *)arg;
    struct device_reg *regp;
    int flags;
    mutex_enter(&xsp->mu);
    if (xsp->busy) {
        /* transfer in progress */
        mutex_exit(&xsp->mu);
        return (DDI_DMA_CALLBACK_RUNOUT);
    }
    xsp->busy = 1;
    regp = xsp->regp;
    if (/* transfer is a read */) {
        flags = DDI_DMA_READ;
    } else {
        flags = DDI_DMA_WRITE;
    }
    mutex_exit(&xsp->mu);
    if (ddi_dma_buf_bind_handle(xsp->handle,xsp->bp,flags, xxstart, 
       (caddr_t)xsp, &cookie, &ccount) != DDI_DMA_MAPPED) {
        /* really should check all return values in a switch */
        mutex_enter(&xsp->mu);
        xsp->busy=0;
        mutex_exit(&xsp->mu);
        return (DDI_DMA_CALLBACK_RUNOUT);
    }
    /* Program the DMA engine. */
    return (DDI_DMA_CALLBACK_DONE);
}
```

Determining Maximum Burst Sizes

Drivers specify the DMA burst sizes that their device supports in the dma_attr_burstsizes field of the ddi_dma_attr(9S) structure. This field is a bitmap of the supported burst sizes. However, when DMA resources are allocated, the system might impose further restrictions on the burst sizes that might be actually used by the device. The ddi_dma_burstsizes(9F) routine can be used to obtain the allowed burst sizes. This routine returns the appropriate burst size bitmap for the device. When DMA resources are allocated, a driver can ask the system for appropriate burst sizes to use for its DMA engine.
EXAMPLE 9-2 Determining Burst Size

```c
#define BEST_BURST_SIZE 0x20 /* 32 bytes */

if (ddi_dma_buf_bind_handle(xsp->handle,xsp->bp, flags, xxstart, 
    (caddr_t)xsp, &cookie, &ccount) != DDI_DMA_MAPPED) {
    /* error handling */
}
burst = ddi_dma_burstsizes(xsp->handle);
/* check which bit is set and choose one burstsize to */
/* program the DMA engine */
if (burst & BEST_BURST_SIZE) {
    /* program DMA engine to use this burst size */
} else {
    /* other cases */
}
```

Allocating Private DMA Buffers

Some device drivers might need to allocate memory for DMA transfers in addition to performing transfers requested by user threads and the kernel. Some examples of allocating private DMA buffers are setting up shared memory for communication with the device and allocating intermediate transfer buffers. Use `ddi_dma_mem_alloc(9F)` to allocate memory for DMA transfers.

```c
int ddi_dma_mem_alloc(ddi_dma_handle_t handle, size_t length, 
    ddi_device_acc_attr_t *accattrp, uint_t flags, 
    int (*waitfp)(caddr_t), caddr_t arg, caddr_t *kaddrp, 
    size_t *real_length, ddi_acc_handle_t *handlep);
```

where:

- **handle**  
  DMA handle

- **length**  
  Length in bytes of the desired allocation

- **accattrp**  
  Pointer to a device access attribute structure

- **flags**  
  Data transfer mode flags. Possible values are `DDI_DMA_CONSISTENT` and `DDI_DMA_STREAMING`.

- **waitfp**  
  Address of callback function for handling resource allocation failures. See the `ddi_dma_alloc_handle(9F)` man page.

- **arg**  
  Argument to pass to the callback function

- **kaddrp**  
  Pointer on a successful return that contains the address of the allocated storage

- **real_length**  
  Length in bytes that was allocated

- **handlep**  
  Pointer to a data access handle
The `flags` parameter should be set to `DDI_DMA_CONSISTENT` if the device accesses in a nonsequential fashion. Synchronization steps that use `ddi_dma_sync(9F)` should be as lightweight as possible due to frequent application to small objects. This type of access is commonly known as consistent access. Consistent access is particularly useful for I/O parameter blocks that are used for communication between a device and the driver.

On the x86 platform, allocation of DMA memory that is physically contiguous has these requirements:

- The length of the scatter-gather list `dma_attr_sgllen` in the `ddi_dma_attr(9S)` structure must be set to 1.
- Do not specify `DDI_DMA_PARTIAL`. `DDI_DMA_PARTIAL` allows partial resource allocation.

The following example shows how to allocate IOPB memory and the necessary DMA resources to access this memory. DMA resources must still be allocated, and the `DDI_DMA_CONSISTENT` flag must be passed to the allocation function.

```c
EXAMPLE9–3  Using ddi_dma_mem_alloc(9F)

if (ddi_dma_mem_alloc(xsp->iopb_handle, size, &accattr,
    DDI_DMA_CONSISTENT, DDI_DMA_SLEEP, NULL, &xsp->iopb_array,
    &real_length, &xsp->acchandle) != DDI_SUCCESS) {
    /* error handling */
    goto failure;
}
if (ddi_dma_addr_bind_handle(xsp->iopb_handle, NULL,
    xsp->iopb_array, real_length,
    DDI_DMA_READ | DDI_DMA_CONSISTENT, DDI_DMA_SLEEP,
    NULL, &cookie, &count) != DDI_DMA_MAPPED) {
    /* error handling */
    ddi_dma_mem_free(&xsp->acchandle);
    goto failure;
}
```

The `flags` parameter should be set to `DDI_DMA_STREAMING` for memory transfers that are sequential, unidirectional, block-sized, and block-aligned. This type of access is commonly known as streaming access.

In some cases, an I/O transfer can be sped up by using an I/O cache. I/O cache transfers one cache line at a minimum. The `ddi_dma_mem_alloc(9F)` routine rounds `size` to a multiple of the cache line to avoid data corruption.

The `ddi_dma_mem_alloc(9F)` function returns the actual size of the allocated memory object. Because of padding and alignment requirements, the actual size might be larger than the requested size. The `ddi_dma_addr_bind_handle(9F)` function requires the actual length.

Use the `ddi_dma_mem_free(9F)` function to free the memory allocated by `ddi_dma_mem_alloc(9F)`. 
**Handling Resource Allocation Failures**

The resource-allocation routines provide the driver with several options when handling allocation failures. The `waitfp` argument indicates whether the allocation routines block, return immediately, or schedule a callback, as shown in the following table.

<table>
<thead>
<tr>
<th><code>waitfp</code> value</th>
<th>Indicated Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDI_DMA_DONTWAIT</td>
<td>Driver does not want to wait for resources to become available</td>
</tr>
<tr>
<td>DDI_DMA_SLEEP</td>
<td>Driver is willing to wait indefinitely for resources to become available</td>
</tr>
<tr>
<td>Other values</td>
<td>The address of a function to be called when resources are likely to be available</td>
</tr>
</tbody>
</table>

**Programming the DMA Engine**

When the resources have been successfully allocated, the device must be programmed. Although programming a DMA engine is device specific, all DMA engines require a starting address and a transfer count. Device drivers retrieve these two values from the `DMA cookie` returned by a successful call from `ddi_dma_addr_bind_handle(9F)`, `ddi_dma_buf_bind_handle(9F)`, or `ddi_dma_getwin(9F)`. These functions all return the first DMA cookie and a cookie count indicating whether the DMA object consists of more than one cookie. If the cookie count `N` is greater than 1, `ddi_dma_nextcookie(9F)` must be called `N-1` times to retrieve all the remaining cookies.

A DMA cookie is of type `ddi_dma_cookie(9S)`. This type of cookie has the following fields:

```c
uint64_t _dmac_ll; /* 64-bit DMA address */
uint32_t _dmac_la[2]; /* 2 x 32-bit address */
size_t dmac_size; /* DMA cookie size */
uint_t dmac_type; /* bus specific type bits */
```

The `_dmac_ll` address specifies a 64-bit I/O address that is appropriate for programming the device's DMA engine. If a device has a 64-bit DMA address register, a driver should use this field to program the DMA engine. The `_dmac_address` field specifies a 32-bit I/O address that
should be used for devices that have a 32-bit DMA address register. The dmac_size field contains the transfer count. Depending on the bus architecture, the dmac_type field in the cookie might be required by the driver. The driver should not perform any manipulations, such as logical or arithmetic, on the cookie.

**EXAMPLE 9-4  ddi_dma_cookie(9S) Example**

```c
ddi_dma_cookie_t cookie;
if (ddi_dma_buf_bind_handle(xsp->handle, xsp->bp, flags, xxstart, 
    (caddr_t)xsp, &cookie, &xsp->ccount) != DDI_DMA_MAPPED) {
    /* error handling */
}
sglp = regp->sglist;
for (cnt = 1; cnt <= SGLLEN; cnt++, sglp++) {
    /* store the cookie parms into the S/G list */
    ddi_put32(xsp->access_hdl, &sglp->dma_size, 
        (uint32_t)cookie.dmac_size);
    ddi_put32(xsp->access_hdl, &sglp->dma_addr, 
        cookie.dmac_address);
    /* Check for end of cookie list */
    if (cnt == xsp->ccount)
        break;
    /* Get next DMA cookie */
    (void) ddi_dma_nextcookie(xsp->handle, &cookie);
} /* start DMA transfer */
ddi_put8(xsp->access_hdl, &regp->csr, 
    ENABLE_INTERRUPTS | START_TRANSFER);
```

**Freeing the DMA Resources**

After a DMA transfer is completed, usually in the interrupt routine, the driver can release DMA resources by calling `ddi_dma_unbind_handle(9F)`.

As described in “Synchronizing Memory Objects” on page 171, `ddi_dma_unbind_handle(9F)` calls `ddi_dma_sync(9F)`, eliminating the need for any explicit synchronization. After calling `ddi_dma_unbind_handle(9F)`, the DMA resources become invalid, and further references to the resources have undefined results. The following example shows how to use `ddi_dma_unbind_handle(9F)`.

**EXAMPLE 9-5  Freeing DMA Resources**

```c
static uint_t
xxintr(caddr_t arg)
{
    struct xxstate *xsp = (struct xxstate *)arg;
    uint8_t status;
    volatile uint8_t temp;
    mutex enter(&xsp->mu);
    /* read status */
    status = ddi_get8(xsp->access_hdl, &xsp->regp->csr);
```
EXAMPLE 9–5 Freeing DMA Resources (Continued)

```c
if (!(status & INTERRUPTING)) {
    mutex_exit(&xsp->mu);
    return (DDI_INTR_UNCLAIMED);
}

ddi_put8(xsp->access_hdl, &xsp->regp->csr, CLEAR_INTERRUPT);
/* for store buffers */
temp = ddi_get8(xsp->access_hdl, &xsp->regp->csr);
// Check for errors. */
xsp->busy = 0;
mutex_exit(&xsp->mu);
if (/* pending transfers */ ) {
    (void) xxstart((caddr_t)xsp);
}
return (DDI_INTR_CLAIMED);
```

The DMA resources should be released. The DMA resources should be reallocated if a different object is to be used in the next transfer. However, if the same object is always used, the resources can be allocated once. The resources can then be reused as long as intervening calls to `ddi_dma_sync(9F)` remain.

**Freeing the DMA Handle**

When the driver is detached, the DMA handle must be freed. The `ddi_dma_free_handle(9F)` function destroys the DMA handle and destroys any residual resources that the system is caching on the handle. Any further references of the DMA handle will have undefined results.

**Canceling DMA Callbacks**

DMA callbacks cannot be canceled. Canceling a DMA callback requires some additional code in the driver's `detach(9E)` entry point. The `detach()` routine must not return `DDI_SUCCESS` if any outstanding callbacks exist. See Example 9–6. When DMA callbacks occur, the `detach()` routine must wait for the callback to run. When the callback has finished, `detach()` must prevent the callback from rescheduling itself. Callbacks can be prevented from rescheduling through additional fields in the state structure, as shown in the following example.

EXAMPLE 9–6 Canceling DMA Callbacks

```c
static int
xxdetach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
    /* ... */
    mutex_enter(&xsp->callback_mutex);
    xsp->cancel_callbacks = 1;
    while (xsp->callback_count > 0) {
```
EXAMPLE 9–6  Canceling DMA Callbacks  (Continued)

    cv_wait(&xsp->callback_cv, &xsp->callback_mutex);
}
mutex_exit(&xsp->callback_mutex);
/* ... */

static int
xxstrategy(struct buf *bp)
{
    /* ... */
    mutex_enter(&xsp->callback_mutex);
    xsp->bp = bp;
    error = ddi_dma_buf_bind_handle(xsp->handle, xsp->bp, flags, xxdmacallback, (caddr_t)xsp, &cookie, &ccount);
    if (error == DDI_DMA_NORESOURCES)
        xsp->callback_count++;
    mutex_exit(&xsp->callback_mutex);
    /* ... */
}

static int
xxdmacallback(caddr_t callbackarg)
{
    struct xxstate *xsp = (struct xxstate *)callbackarg;
    /* ... */
    mutex_enter(&xsp->callback_mutex);
    if (xsp->cancel_callbacks) {
        /* do not reschedule, in process of detaching */
        xsp->callback_count--;
        if (xsp->callback_count == 0)
            cv_signal(&xsp->callback_cv);
        mutex_exit(&xsp->callback_mutex);
        return (DDI_DMA_CALLBACK_DONE); /* don't reschedule it */
    }
    /*
    * Presumably at this point the device is still active
    * and will not be detached until the DMA has completed.
    * A return of 0 means try again later
    */
    error = ddi_dma_buf_bind_handle(xsp->handle, xsp->bp, flags, DDI_DMA_DONTWAIT, NULL, &cookie, &ccount);
    if (error == DDI_DMA_MAPPED) {
        /* Program the DMA engine. */
        xsp->callback_count--;
        mutex_exit(&xsp->callback_mutex);
        return (DDI_DMA_CALLBACK_DONE);
    }
    if (error != DDI_DMA_NORESOURCES) {
        xsp->callback_count--;
        mutex_exit(&xsp->callback_mutex);
        return (DDI_DMA_CALLBACK_DONE);
    }
    mutex_exit(&xsp->callback_mutex);
    return (DDI_DMA_CALLBACK_RUNOUT);
}
Synchronizing Memory Objects

In the process of accessing the memory object, the driver might need to synchronize the memory object with respect to various caches. This section provides guidelines on when and how to synchronize memory objects.

Cache

CPU cache is a very high-speed memory that sits between the CPU and the system’s main memory. I/O cache sits between the device and the system’s main memory, as shown in the following figure.

![CPU and System I/O Caches](image)

When an attempt is made to read data from main memory, the associated cache checks for the requested data. If the data is available, the cache supplies the data quickly. If the cache does not have the data, the cache retrieves the data from main memory. The cache then passes the data on to the requester and saves the data in case of a subsequent request.

Similarly, on a write cycle, the data is stored in the cache quickly. The CPU or device is allowed to continue executing, that is, transferring data. Storing data in a cache takes much less time than waiting for the data to be written to memory.

With this model, after a device transfer is complete, the data can still be in the I/O cache with no data in main memory. If the CPU accesses the memory, the CPU might read the wrong data from the CPU cache. The driver must call a synchronization routine to flush the data from the I/O cache and update the CPU cache with the new data. This action ensures a consistent view of the memory for the CPU. Similarly, a synchronization step is required if data modified by the CPU is to be accessed by a device.
You can create additional caches and buffers between the device and memory, such as bus extenders and bridges. Use `ddi_dma_sync()` to synchronize all applicable caches.

### ddi_dma_sync() Function

A memory object might have multiple mappings, such as for the CPU and for a device, by means of a DMA handle. A driver with multiple mappings needs to call `ddi_dma_sync()` if any mappings are used to modify the memory object. Calling `ddi_dma_sync()` ensures that the modification of the memory object is complete before the object is accessed through a different mapping. The `ddi_dma_sync()` function can also inform other mappings of the object if any cached references to the object are now stale. Additionally, `ddi_dma_sync()` flushes or invalidates stale cache references as necessary.

Generally, the driver must call `ddi_dma_sync()` when a DMA transfer completes. The exception to this rule is if deallocating the DMA resources with `ddi_dma_unbind_handle()` does an implicit `ddi_dma_sync()` on behalf of the driver. The syntax for `ddi_dma_sync()` is as follows:

```c
int ddi_dma_sync(ddi_dma_handle_t handle, off_t off, size_t length, uint_t type);
```

If the object is going to be read by the DMA engine of the device, the device’s view of the object must be synchronized by setting `type` to `DDI_DMA_SYNC_FORDEV`. If the DMA engine of the device has written to the memory object and the object is going to be read by the CPU, the CPU’s view of the object must be synchronized by setting `type` to `DDI_DMA_SYNC_FORCPU`.

The following example demonstrates synchronizing a DMA object for the CPU:

```c
if (ddi_dma_sync(xsp->handle, 0, length, DDI_DMA_SYNC_FORCPU) == DDI_SUCCESS) {
    /* the CPU can now access the transferred data */
    /* ... */
} else {
    /* error handling */
}
```

Use the flag `DDI_DMA_SYNC_FORKERNEL` if the only mapping is for the kernel, as in the case of memory that is allocated by `ddi_dma_mem_alloc()`. The system tries to synchronize the kernel’s view more quickly than the CPU’s view. If the system cannot synchronize the kernel view faster, the system acts as if the `DDI_DMA_SYNC_FORCPU` flag were set.
DMA Windows

If an object does not fit within the limitations of the DMA engine, the transfer must be broken into a series of smaller transfers. The driver can break up the transfer itself. Alternatively, the driver can allow the system to allocate resources for only part of the object, thereby creating a series of DMA windows. Allowing the system to allocate resources is the preferred solution, because the system can manage the resources more effectively than the driver can manage the resources.

A DMA window has two attributes. The offset attribute is measured from the beginning of the object. The length attribute is the number of bytes of memory to be allocated. After a partial allocation, only a range of length bytes that starts at offset has allocated resources.

A DMA window is requested by specifying the DDI_DMA_PARTIAL flag as a parameter to ddi_dma_buf_bind_handle(9F) or ddi_dma_addr_bind_handle(9F). Both functions return DDI_DMA_PARTIAL_MAP if a window can be established. However, the system might allocate resources for the entire object, in which case DDI_DMA_MAPPED is returned. The driver should check the return value to determine whether DMA windows are in use. See the following example.

EXAMPLE 9–7 Setting Up DMA Windows

```c
static int
xxstart (caddr_t arg)
{
    struct xxstate *xsp = (struct xxstate *)arg;
    struct device_reg *regp = xsp->reg;
    ddi_dma_cookie_t cookie;
    int status;
    mutex_enter(&xsp->mu);
    if (xsp->busy) {
        /* transfer in progress */
        mutex_exit(&xsp->mu);
        return (DDI_DMA_CALLBACK_RUNOUT);
    }
    xsp->busy = 1;
    mutex_exit(&xsp->mu);
    if ( /* transfer is a read */) {
        flags = DDI_DMA_READ;
    } else {
        flags = DDI_DMA_WRITE;
    }
    flags |= DDI_DMA_PARTIAL;
    status = ddi_dma_buf_bind_handle(xsp->handle, xsp->bp,
        flags, xxstart, (caddr_t)xsp, &cookie, &ccount);
    if (status != DDI_DMA_MAPPED &&
        status != DDI_DMA_PARTIAL_MAP)
        return (DDI_DMA_CALLBACK_RUNOUT);
    if (status == DDI_DMA_PARTIAL_MAP) {
        ddi_dma_numwin(xsp->handle, &xsp->nwin);
        xsp->partial = 1;
        xsp->windex = 0;
    } else {
```
EXAMPLE 9–7  Setting Up DMA Windows   (Continued)

    xsp->partial = 0;
}
/* Program the DMA engine. */
return (DDI_DMA_CALLBACK_DONE);
}

TwofunctionsoperatewithDMAwindows. The first, ddi_dma_numwin(9F), returns the
number ofDMAwindowsforaparticularDMAobject. The other function,
ddi_dma_getwin(9F), allows repositioning within the object, that is, reallocation of system
resources. The ddi_dma_getwin() function shifts the current window to a new window within
the object. Because ddi_dma_getwin() reallocates system resources to the new window, the
previous window becomes invalid.

Caution – Do not move the DMA windows with a call to ddi_dma_getwin() before transfers into
the current window are complete. Wait until the transfer to the current window is complete,
which is when the interrupt arrives. Then call ddi_dma_getwin() to avoid data corruption.

The ddi_dma_getwin() function is normally called from an interrupt routine, as shown in
Example 9–8. The first DMA transfer is initiated as a result of a call to the driver. Subsequent
transfers are started from the interrupt routine.

The interrupt routine examines the status of the device to determine whether the device
completes the transfer successfully. If not, normal error recovery occurs. If the transfer is
successful, the routine must determine whether the logical transfer is complete. A complete
transfer includes the entire object as specified by the buf(9S) structure. In a partial transfer, only
one DMA window is moved. In a partial transfer, the interrupt routine moves the window with
ddi_dma_getwin(9F), retrieves a new cookie, and starts another DMA transfer.

If the logical request has been completed, the interrupt routine checks for pending requests. If
necessary, the interrupt routine starts a transfer. Otherwise, the routine returns without
invoking another DMA transfer. The following example illustrates the usual flow control.

EXAMPLE 9–8  Interrupt Handler Using DMA Windows
static uint_t
xxintr(caddr_t arg)
{
    struct xxstate *xsp = (struct xxstate *)arg;
    uint8_t status;
    volatile uint8_t temp;
    mutex_enter(&xsp->mu);
    /* read status */
    status = ddi_get8(xsp->access_hdl, &xsp->regp->csr);
    if (!((status & INTERRUPTING)) {
        mutex_exit(&xsp->mu);
EXAMPLE 9–8  Interrupt Handler Using DMA Windows  (Continued)

```c
return (DDI_INTR_UNCLAIMED);
}
ddi_put8(xsp->access_hdl, &xsp->regp->csr, CLEAR_INTERRUPT);
/* for store buffers */
temp = ddi_get8(xsp->access_hdl, &xsp->regp->csr);
if ( /* an error occurred during transfer */ ) {
    bioerror(xsp->bp, EIO);
    xsp->partial = 0;
} else {
    xsp->bp->b_resid -= /* amount transferred */;
}
if (xsp->partial && (++xsp->windex < xsp->nwin)) {
    /* device still marked busy to protect state */
    mutex_exit(&xsp->mu);
    (void) ddi_dma_getwin(xsp->handle, xsp->windex, &offset, &len, &cookie, &ccount);
    /* Program the DMA engine with the new cookie(s). */
    return (DDI_INTR_CLAIMED);
}
ddi_dma_unbind_handle(xsp->handle);
biodone(xsp->bp);
xsp->busy = 0;
xsp->partial = 0;
mutex_exit(&xsp->mu);
if ( /* pending transfers */ ) {
    (void) xxstart((caddr_t)xsp);
}
return (DDI_INTR_CLAIMED);
```
Some device drivers allow applications to access device or kernel memory through `mmap(2)`. Frame buffer drivers, for example, enable the frame buffer to be mapped into a user thread. Another example would be a pseudo driver that uses a shared kernel memory pool to communicate with an application. This chapter provides information on the following subjects:

- “Memory Mapping Overview” on page 177
- “Exporting the Mapping” on page 177
- “Associating Device Memory With User Mappings” on page 181
- “Associating Kernel Memory With User Mappings” on page 182

### Memory Mapping Overview

The steps that a driver must take to export device or kernel memory are as follows:

1. Set the `D_DEVMAP` flag in the `cb_flag` flag of the `cb_ops(9S)` structure.
2. Define a `devmap(9E)` driver entry point and optional `segmap(9E)` entry point to export the mapping.
3. Use `devmap_devmem_setup(9F)` to set up user mappings to a device. To set up user mappings to kernel memory, use `devmap_umem_setup(9F)`.

### Exporting the Mapping

This section describes how to use the `segmap(9E)` and `devmap(9E)` entry points.

### The segmap(9E) Entry Point

The `segmap(9E)` entry point is responsible for setting up a memory mapping requested by an `mmap(2)` system call. Drivers for many memory-mapped devices use `ddi_devmap_segmap(9F)` as the entry point rather than defining their own `segmap(9E)` routine. By providing a `segmap()`
entry point, a driver can take care of general tasks before or after creating the mapping. For example, the driver can check mapping permissions and allocate private mapping resources. The driver can also make adjustments to the mapping to accommodate non-page-aligned device buffers. The segmap() entry point must call the ddi_devmap_segmap(9F) function before returning. The ddi_devmap_segmap() function calls the driver’s devmap(9E) entry point to perform the actual mapping.

The segmap() function has the following syntax:

```c
int segmap(dev_t dev, off_t off, struct as *asp, caddr_t *addrp,
            off_t len, unsigned int prot, unsigned int maxprot,
            unsigned int flags, cred_t *credp);
```

where:

- **dev** Device whose memory is to be mapped.
- **off** Offset within device memory at which mapping begins.
- **asp** Pointer to the address space into which the device memory should be mapped.

Note that this argument can be either a `struct as *`, as shown in Example 10–1, or a `ddi_as_handle_t`, as shown in Example 10–2. This is because `ddidevmap.h` includes the following declaration:

```c
typedef struct as *ddi_as_handle_t
```

- **addrp** Pointer to the address in the address space to which the device memory should be mapped.
- **len** Length (in bytes) of the memory being mapped.
- **prot** A bit field that specifies the protections. Possible settings are PROT_READ, PROT_WRITE, PROT_EXEC, PROT_USER, and PROT_ALL. See the man page for details.
- **maxprot** Maximum protection flag possible for attempted mapping. The PROT_WRITE bit can be masked out if the user opened the special file read-only.
- **flags** Flags that indicate the type of mapping. Possible values include MAP_SHARED and MAP_PRIVATE.
- **credp** Pointer to the user credentials structure.

In the following example, the driver controls a frame buffer that allows write-only mappings. The driver returns EINVAL if the application tries to gain read access and then calls ddi_devmap_segmap(9F) to set up the user mapping.

**EXAMPLE 10–1** segmap(9E) Routine

```c
static int
xxsegmap(dev_t dev, off_t off, struct as *asp, caddr_t *addrp,
```
EXAMPLE 10–1  segmap(9E) Routine  (Continued)

    off_t len, unsigned int prot, unsigned int maxprot,
    unsigned int flags, cred_t *credp)
{    
    if (prot & PROT_READ)
        return (EINVAL);
    return (ddi_devmap_segmap(dev, off, as, addrp,
        len, prot, maxprot, flags, credp));
}

The following example shows how to handle a device that has a buffer that is not page-aligned in
its register space. This example maps a buffer that starts at offset 0x800, so that `mmap(2)` returns
an address that corresponds to the start of the buffer. The `devmap_devmem_setup(9F)` function
maps entire pages, requires the mapping to be page aligned, and returns an address to the start
of a page. If this address is passed through `segmap(9E)`, or if no `segmap()` entry point is defined,
`mmap()` returns the address that corresponds to the start of the page, not the address that
corresponds to the start of the buffer. In this example, the buffer offset is added to the
page-aligned address that was returned by `devmap_devmem_setup` so that the resulting address
returned is the desired start of the buffer.

EXAMPLE 10–2  Using the `segmap()` Function to Change the Address Returned by the `mmap()` Call

#define BUFFER_OFFSET 0x800

int
xx_segmap(dev_t dev, off_t off, ddi_as_handle_t as, caddr_t *addrp, off_t len,
    uint_t prot, uint_t maxprot, uint_t flags, cred_t *credp)
{
    int rval;
    unsigned long pagemask = ptob(1L) - 1L;

    if ((rval = ddi_devmap_segmap(dev, off, as, addrp, len, prot, maxprot,
      flags, credp)) == DDI_SUCCESS) {
        /*
        * The address returned by ddi_devmap_segmap is the start of the page
        * that contains the buffer. Add the offset of the buffer to get the
        * final address.
        */
        *addrp += BUFFER_OFFSET & pagemask);
    }
    return (rval);
}

The `devmap(9E)` Entry Point

The `devmap(9E)` entry point is called from the `ddi_devmap_segmap(9F)` function inside the
`segmap(9E)` entry point.
The `devmap(9E)` entry point is called as a result of the `mmap(2)` system call. The `devmap(9E)` function is called to export device memory or kernel memory to user applications. The `devmap()` function is used for the following operations:

- Validate the user mapping to the device or kernel memory
- Translate the logical offset within the application mapping to the corresponding offset within the device or kernel memory
- Pass the mapping information to the system for setting up the mapping

The `devmap()` function has the following syntax:

```c
int devmap(dev_t dev, devmap_cookie_t handle, offset_t off,
           size_t len, size_t *maplen, uint_t model);
```

where:

- **dev**: Device whose memory is to be mapped.
- **handle**: Device-mapping handle that the system creates and uses to describe a mapping to contiguous memory in the device or kernel.
- **off**: Logical offset within the application mapping that has to be translated by the driver to the corresponding offset within the device or kernel memory.
- **len**: Length (in bytes) of the memory being mapped.
- **maplen**: Enables driver to associate different kernel memory regions or multiple physically discontinuous memory regions with one contiguous user application mapping.
- **model**: Data model type of the current thread.

The system creates multiple mapping handles in one `mmap(2)` system call. For example, the mapping might contain multiple physically discontinuous memory regions.

Initially, `devmap(9E)` is called with the parameters `off` and `len`. These parameters are passed by the application to `mmap(2)`. `devmap(9E)` sets `maplen` to the length from `off` to the end of a contiguous memory region. The `maplen` value must be rounded up to a multiple of a page size. The `maplen` value can be set to less than the original mapping length `len`. If so, the system uses a new mapping handle with adjusted `off` and `len` parameters to call `devmap(9E)` repeatedly until the initial mapping length is satisfied.

If a driver supports multiple application data models, `model` must be passed to `ddi_model_convert_from(9F)`. The `ddi_model_convert_from()` function determines whether a data model mismatch exists between the current thread and the device driver. The device driver might have to adjust the shape of data structures before exporting the structures to a user thread that supports a different data model. See Appendix C, “Making a Device Driver 64-Bit Ready,” page for more details.
The devmap(9E) entrypoint must return -1 if the logical offset, off, is out of the range of memory exported by the driver.

**Associating Device Memory With User Mappings**

Call devmap_devmem_setup(9F) from the driver's devmap(9E) entry point to export device memory to user applications.

The devmap_devmem_setup(9F) function has the following syntax:

```c
int devmap_devmem_setup(devmap_cookie_t handle, dev_info_t *dip,
                       struct devmap_callback_ctl *callbacks, uint_t rnumber,
                       offset_t roff, size_t len, uint_t maxprot, uint_t flags,
                       ddi_device_acc_attr_t *accattrp);
```

where:

- **handle**: Opaque device-mapping handle that the system uses to identify the mapping.
- **dip**: Pointer to the device's dev_info structure.
- **callbacks**: Pointer to a devmap_callback_ctl(9S) structure that enables the driver to be notified of user events on the mapping.
- **rnumber**: Index number to the register address space set.
- **roff**: Offset into the device memory.
- **len**: Length in bytes that is exported.
- **maxprot**: Allows the driver to specify different protections for different regions within the exported device memory.
- **flags**: Must be set to DEVMAP_DEFAULTS.
- **accattrp**: Pointer to a ddi_device_acc_attr(9S) structure.

The roff and len arguments describe a range within the device memory specified by the register set rnumber. The register specifications that are referred to by rnumber are described by the reg property. For devices with only one register set, pass zero for rnumber. The range is defined by roff and len. The range is made accessible to the user's application mapping at the offset that is passed in by the devmap(9E) entry point. Usually the driver passes the devmap(9E) offset directly to devmap_devmem_setup(9F). The return address of mmap(2) then maps to the beginning address of the register set.

The maxprot argument enables the driver to specify different protections for different regions within the exported device memory. For example, to disallow write access for a region, set only PROT_READ and PROT_USER for that region.
The following example shows how to export device memory to an application. The driver first determines whether the requested mapping falls within the device memory region. The size of the device memory is determined using \texttt{ddi_dev_regsize(9F)}. The length of the mapping is rounded up to a multiple of a page size using \texttt{ptob(9F)} and \texttt{btopr(9F)}. Then \texttt{devmap_devmem_setup(9F)} is called to export the device memory to the application.

**EXAMPLE 10–3** Using the \texttt{devmap_devmem_setup()} Routine

```c
static int
 xxdevmap(dev_t dev, devmap_cookie_t handle, offset_t off, size_t len,
  size_t *maplen, uint_t model)
{
  struct xxstate *xsp;
  int error, rnumber;
  off_t regsize;

  /* Set up data access attribute structure */
  struct ddi_device_acc_attr xx_acc_attr = {
    DDI_DEVICE_ATTR_V8,
    DDI_NEVERSWAP_ACC,
    DDI_STRICTORDER_ACC
  };
  xsp = ddi_get_soft_state(statep, getminor(dev));
  if (xsp == NULL)
    return (-1);
  /* use register set 0 */
  rnumber = 0;
  /* get size of register set */
  if (ddi_dev_regsize(xsp->dip, rnumber, &regsize) != DDI_SUCCESS)
    return (-1);
  /* round up len to a multiple of a page size */
  len = ptob(btopr(len));
  if (off + len > regsize)
    return (-1);
  /* Set up the device mapping */
  error = devmap_devmem_setup(handle, xsp->dip, NULL, rnumber,
    off, len, PROT_ALL, DEVMAP_DEFAULTS, &xx_acc_attr);
  /* acknowledge the entire range */
  *maplen = len;
  return (error);
}
```

**Associating Kernel Memory With User Mappings**

Some device drivers might need to allocate kernel memory that is made accessible to user programs through \texttt{mmap(2)}. One example is setting up shared memory for communication between two applications. Another example is sharing memory between a driver and an application.

When exporting kernel memory to user applications, follow these steps:

1. Use \texttt{ddi_umem_alloc(9F)} to allocate kernel memory.
2. Use `devmap_umem_setup(9F)` to export the memory.
3. Use `ddi_umem_free(9F)` to free the memory when the memory is no longer needed.

## Allocating Kernel Memory for User Access

Use `ddi_umem_alloc(9F)` to allocate kernel memory that is exported to applications.

`ddi_umem_alloc()` uses the following syntax:

```c
void *ddi_umem_alloc(size_t size, int flag, ddi_umem_cookie_t *cookiep);
```

**where:**

- `size`: Number of bytes to allocate.
- `flag`: Used to determine the sleep conditions and the memory type.
- `cookiep`: Pointer to a kernel memory cookie.

`ddi_umem_alloc(9F)` allocates page-aligned kernel memory. `ddi_umem_alloc()` returns a pointer to the allocated memory. Initially, the memory is filled with zeroes. The number of bytes that are allocated is a multiple of the system page size, which is rounded up from the `size` parameter. The allocated memory can be used in the kernel. This memory can be exported to applications as well. `cookiep` is a pointer to the kernel memory cookie that describes the kernel memory being allocated. `cookiep` is used in `devmap_umem_setup(9F)` when the driver exports the kernel memory to a user application.

The `flag` argument indicates whether `ddi_umem_alloc(9F)` blocks or returns immediately, and whether the allocated kernel memory is pageable. The values for the `flag` argument as follows:

- **DDI_UMEM_NOSLEEP**: Driver does not need to wait for memory to become available. Return NULL if memory is not available.
- **DDI_UMEM_SLEEP**: Driver can wait indefinitely for memory to become available.
- **DDI_UMEM_PAGEABLE**: Driver allows memory to be paged out. If not set, the memory is locked down.

The `ddi_umem_lock()` function can perform device-locked-memory checks. The function checks against the limit value that is specified in `project.max.locked-memory`. If the current project locked-memory usage is below the limit, the project’s locked-memory byte count is increased. After the limit check, the memory is locked. The `ddi_umem_unlock()` function unlocks the memory, and the project’s locked-memory byte count is decremented.

The accounting method that is used is an imprecise full price model. For example, two callers of `umem_lockmemory()` within the same project with overlapping memory regions are charged twice.
For information about the `max-locked-memory` resource controls on Oracle Solaris systems with zones installed, see Resource Management and Oracle Solaris Zones Developer's Guide and see `resource_controls(5)`.

The following example shows how to allocate kernel memory for application access. The driver exports one page of kernel memory, which is used by multiple applications as a shared memory area. The memory is allocated in `segmap(9E)` when an application maps the shared page the first time. An additional page is allocated if the driver has to support multiple application data models. For example, a 64-bit driver might export memory both to 64-bit applications and to 32-bit applications. 64-bit applications share the first page, and 32-bit applications share the second page.

**EXAMPLE 10-4 Using the `ddi_umem_alloc()` Routine**

```c
static int
xxsegmap(dev_t dev, off_t off, struct *asp, caddr_t *addrp, off_t len,  
          unsigned int prot, unsigned int maxprot, unsigned int flags,  
          cred_t *credp)
{
    int error;
    minor_t instance = getminor(dev);
    struct xxstate *xsp = ddi_get_soft_state(statep, instance);

    size_t mem_size;
    /* 64-bit driver supports 64-bit and 32-bit applications */
    switch (ddi_mmap_get_model()) {
        case DDI_MODEL_LP64:
            mem_size = ptob(2);
            break;
        case DDI_MODEL_ILP32:
            mem_size = ptob(1);
            break;
    }

    mutex_enter(&xsp->mu);
    if (xsp->umem == NULL) {
        /* allocate the shared area as kernel pageable memory */
        xsp->umem = ddi_umem_alloc(mem_size,  
                                   DDI_UMEM_SLEEP | DDI_UMEM_PAGEABLE, &xsp->ucookie);
    }
    mutex_exit(&xsp->mu);

    /* Set up the user mapping */
    /* error = devmap_setup(dev, (offset_t)off, asp, addrp, len,  
                             prot, maxprot, flags, credp);  
    return (error);  
```
Exporting Kernel Memory to Applications

Use `devmap_umem_setup(9F)` to export kernel memory to user applications. `devmap_umem_setup()` must be called from the driver’s `devmap(9E)` entry point. The syntax for `devmap_umem_setup()` is as follows:

```c
int devmap_umem_setup(devmap_cookie_t handle, dev_info_t *dip,  
                      struct devmap_callback_ctl *callbacks, ddi_umem_cookie_t cookie,  
                      offset_t koff, size_t len, uint_t maxprot, uint_t flags,  
                      ddi_device_acc_attr_t *accattrp);
```

where:

- `handle` Opaque structure used to describe the mapping.
- `dip` Pointer to the device’s `dev_info` structure.
- `callbacks` Pointer to a `devmap_callback_ctl(9S)` structure.
- `cookie` Kernel memory cookie returned by `ddi_umem_alloc(9F)`.
- `koff` Offset into the kernel memory specified by cookie.
- `len` Length in bytes that is exported.
- `maxprot` Specifies the maximum protection possible for the exported mapping.
- `flags` Must be set to `DEVMAP_DEFAULTS`.
- `accattrp` Pointer to a `ddi_device_acc_attr(9S)` structure.

`handle` is a device-mapping handle that the system uses to identify the mapping. `handle` is passed in by the `devmap(9E)` entry point. `dip` is a pointer to the device’s `dev_info` structure. `callbacks` enables the driver to be notified of user events on the mapping. Most drivers set `callbacks` to NULL when kernel memory is exported.

`koff` and `len` specify a range within the kernel memory allocated by `ddi_umem_alloc(9F)`. This range is made accessible to the user’s application mapping at the offset that is passed in by the `devmap(9E)` entry point. Usually, the driver passes the `devmap(9E)` offset directly to `devmap_umem_setup(9F)`. The return address of `mmap(2)` then maps to the kernel address returned by `ddi_umem_alloc(9F)`. `koff` and `len` must be page-aligned.

`maxprot` enables the driver to specify different protections for different regions within the exported kernel memory. For example, one region might not allow write access by only setting `PROT_READ` and `PROT_USER`.

The following example shows how to export kernel memory to an application. The driver first checks whether the requested mapping falls within the allocated kernel memory region. If a 64-bit driver receives a mapping request from a 32-bit application, the request is redirected to
the second page of the kernel memory area. This redirection ensures that only applications compiled to the same data model share the same page.

EXAMPLE 10–5  devmap_umem_setup(9F) Routine

```c
static int
xxdevmap(dev_t dev, devmap_cookie_t handle, offset_t off, size_t len,
        size_t *maplen, uint_t model)
{
    struct xxstate *xsp;
    int    error;

    /* round up len to a multiple of a page size */
    len = ptob(btopr(len));
    /* check if the requested range is ok */
    if (off + len > ptob(1))
        return (ENXIO);
    xsp = ddi_get_soft_state(statep, getminor(dev));
    if (xsp == NULL)
        return (ENXIO);
    if (ddi_model_convert_from(model) == DDI_MODEL_ILP32)
        /* request from 32-bit application. Skip first page */
        off += ptob(1);
    /* export the memory to the application */
    error = devmap_umem_setup(handle, xsp->dip, NULL, xsp->ucookie,
        off, len, PROT_ALL, DEVMAP_DEFAULTS, NULL);
    *maplen = len;
    return (error);
}
```

**Freeing Kernel Memory Exported for User Access**

When the driver is unloaded, the memory that was allocated by `ddi_umem_alloc(9F)` must be freed by calling `ddi_umem_free(9F)`.

```c
void ddi_umem_free(ddi_umem_cookie_t cookie);
```

`cookie` is the kernel memory cookie returned by `ddi_umem_alloc(9F)`.
Device Context Management

Some device drivers, such as drivers for graphics hardware, provide user processes with direct access to the device. These devices often require that only one process at a time accesses the device.

This chapter describes the set of interfaces that enable device drivers to manage access to such devices. The chapter provides information on the following subjects:

- “Introduction to Device Context” on page 187
- “Context Management Model” on page 187
- “Context Management Operation” on page 189

Introduction to Device Context

This section introduces device context and the context management model.

What Is a Device Context?

The context of a device is the current state of the device hardware. The device driver manages the device context for a process on behalf of the process. The driver must maintain a separate device context for each process that accesses the device. The device driver has the responsibility to restore the correct device context when a process accesses the device.

Context Management Model

Frame buffers provide a good example of device context management. An accelerated frame buffer enables user processes to directly manipulate the control registers of the device through memory-mapped access. Because these processes do not use traditional system calls, a process that accesses the device need not call the device driver. However, the device driver must be
notified when a process is about to access a device. The driver needs to restore the correct device context and needs to provide any necessary synchronization.

To resolve this problem, the device context management interfaces enable a device driver to be notified when a user process accesses memory-mapped regions of the device, and to control accesses to the device’s hardware. Synchronization and management of the various device contexts are the responsibility of the device driver. When a user process accesses a mapping, the device driver must restore the correct device context for that process.

A device driver is notified whenever a user process performs any of the following actions:

- Accesses a mapping
- Duplicates a mapping
- Frees a mapping
- Creates a mapping

The following figure shows multiple user processes that have memory-mapped a device. The driver has granted process B access to the device, and process B no longer notifies the driver of accesses. However, the driver is still notified if either process A or process C accesses the device.

**FIGURE 11–1  Device Context Management**

At some point in the future, process A accesses the device. The device driver is notified and blocks future access to the device by process B. The driver then saves the device context for process B. The driver restores the device context of process A. The driver then grants access to process A, as illustrated in the following figure. At this point, the device driver is notified if either process B or process C accesses the device.
On a multiprocessor machine, multiple processes could attempt to access the device at the same time. This situation can cause thrashing. Some devices require a longer time to restore a device context. To prevent more CPU time from being used to restore a device context than to actually use that device context, the minimum time that a process needs to have access to the device can be set using \texttt{devmap\_set\_ctx\_timeout}(9F).

The kernel guarantees that once a device driver has granted access to a process, no other process is allowed to request access to the same device for the time interval specified by \texttt{devmap\_set\_ctx\_timeout}(9F).

**Context Management Operation**

The general steps for performing device context management are as follows:

1. Define a \texttt{devmap\_callback\_ctl(9S)} structure.
2. Allocate space to save device context if necessary.
3. Set up user mappings to the device and driver notifications with \texttt{devmap\_devmem\_setup(9F)}.
4. Manage user access to the device with \texttt{devmap\_load(9F)} and \texttt{devmap\_unload(9F)}.
5. Free the device context structure, if needed.

**devmap\_callback\_ctl Structure**

The device driver must allocate and initialize a \texttt{devmap\_callback\_ctl(9S)} structure to inform the system about the entry point routines for device context management.

This structure uses the following syntax:

```c
struct devmap_callback_ctl {
    int devmap_rev;
    int (*devmap_map)(devmap_cookie_t dhp, dev_t dev,
                       uint_t flags, offset_t off, size_t len, void **pvtp);
```

![FIGURE 11-2 Device Context Switched to User Process A](image-url)
The version number of the devmap_callbackCtl structure. The version number must be set to DEVMAP_OPS_REV.

devmap_map Must be set to the address of the driver's devmap_map(9E) entry point.

devmap_access Must be set to the address of the driver's devmap_access(9E) entry point.

devmap_dup Must be set to the address of the driver's devmap_dup(9E) entry point.

devmap_unmap Must be set to the address of the driver's devmap_unmap(9E) entry point.

Entry Points for Device Context Management

The following entry points are used to manage device context:

- devmap(9E)
- devmap_access(9E)
- devmap_contextmgt(9E)
- devmap_dup(9E)
- devmap_unmap(9E)

**devmap_map()** Entry Point

The syntax for devmap(9E) is as follows:

```c
int xxdevmap_map(devmap_cookie_t handle, dev_t dev, uint_t flags,
    offset_t offset, size_t len, void **new_devprivate);
```

The devmap_map() entry point is called after the driver returns from its devmap() entry point and the system has established the user mapping to the device memory. The devmap() entry point enables a driver to perform additional processing or to allocate mapping specific private data. For example, in order to support context switching, the driver has to allocate a context structure. The driver must then associate the context structure with the mapping.

The system expects the driver to return a pointer to the allocated private data in *new-devprivate. The driver must store offset and len, which define the range of the mapping, in its private data. Later, when the system calls devmap_unmap(9E), the driver uses this information to determine how much of the mapping is being unmapped.
flags indicates whether the driver should allocate a private context for the mapping. For example, a driver can allocate a memory region to store the device context if flags is set to MAP_PRIVATE. If MAP_SHARED is set, the driver returns a pointer to a shared region.

The following example shows a devmap() entry point. The driver allocates a new context structure. The driver then saves relevant parameters passed in by the entry point. Next, the mapping is assigned a new context either through allocation or by attaching the mapping to an already existing shared context. The minimum time interval that the mapping should have access to the device is set to one millisecond.

EXAMPLE 11–1 Using the devmap() Routine

```c
static int
int xxdevmap_map(devmap_cookie_t handle, dev_t dev, uint_t flags,
    offset_t offset, size_t len, void **new_devprivate)
{
    struct xxstate *xsp = ddi_get_soft_state(statep,
        getminor(dev));
    struct xxctx *newctx;
    /* create a new context structure */
    newctx = kmalloc(sizeof(struct xxctx), KM_SLEEP);
    newctx->xsp = xsp;
    newctx->handle = handle;
    newctx->offset = offset;
    newctx->flags = flags;
    newctx->len = len;
    mutex_enter(&xsp->ctx_lock);
    if (flags & MAP_PRIVATE) {
        /* allocate a private context and initialize it */
        newctx->context = kmalloc(XXCTX_SIZE, KM_SLEEP);
        xxctxinit(newctx);
    } else {
        /* set a pointer to the shared context */
        newctx->context = xsp->ctx_shared;
    }
    mutex_exit(&xsp->ctx_lock);
    /* give at least 1 ms access before context switching */
    devmap_set_ctx_timeout(handle, drv_usectohz(1000));
    /* return the context structure */
    *new_devprivate = newctx;
    return(0);
}
```

devmap_access() Entry Point

The devmap_access(9E) entry point is called when an access is made to a mapping whose translations are invalid. Mapping translations are invalidated when the mapping is created with devmap_devmem_setup(9F) in response to mmap(2), duplicated by fork(2), or explicitly invalidated by a call to devmap_unload(9F).

The syntax for devmap_access() is as follows:
int xxdevmap_access(devmap_cookie_t handle, void *devprivate, 
    offset_t offset, size_t len, uint_t type, uint_t rw);

where:

handle Mapping handle of the mapping that was accessed by a user process.
devprivate Pointer to the driver private data associated with the mapping.
offset Offset within the mapping that was accessed.
len Length in bytes of the memory being accessed.
type Type of access operation.
rw Specifies the direction of access.

The system expects devmap_access(9E) to call either devmap_do_ctxmgt(9F) or 
devmap_default_access(9F) to load the memory address translations before 
devmap_access() returns. For mappings that support context switching, the device driver 
should call devmap_do_ctxmgt(). This routine is passed all parameters from 
devmap_access(9E), as well as a pointer to the driver entry point devmap_contextmgt(9E), 
which handles the context switching. For mappings that do not support context switching, the 
driver should call devmap_default_access(9F). The purpose of devmap_default_access() is 
to call devmap_load(9F) to load the user translation.

The following example shows a devmap_access(9E) entry point. The mapping is divided into 
two regions. The region that starts at offset OFF_CTXMG with a length of CTXMG_SIZE bytes 
supports context management. The rest of the mapping supports default access.

EXAMPLE 11-2 Using the devmap_access() Routine

#define OFF_CTXMG 0
#define CTXMG_SIZE 0x20000
static int 
xxdevmap_access(devmap_cookie_t handle, void *devprivate, 
    offset_t off, size_t len, uint_t type, uint_t rw)
{
    offset_t diff;
    int   error;

    if ({diff = off - OFF_CTXMG} >= 0 && diff < CTXMG_SIZE) {
        error = devmap_do_ctxmgt(handle, devprivate, off, 
            len, type, rw, xxdevmap_contextmgt);
    } else {
        error = devmap_default_access(handle, devprivate, 
            off, len, type, rw);
    }
    return (error);
}
devmap_contextmgt() Entry Point

The syntax for devmap_contextmgt(9E) is as follows:

```c
int xxdevmap_contextmgt(devmap_cookie_t handle, void *devprivate,
                        offset_t offset, size_t len, uint_t type, uint_t rw);
```

devmap_contextmgt() should call devmap_unload(9F) with the handle of the mapping that currently has access to the device. This approach invalidates the translations for that mapping. The approach ensures that a call to devmap_access(9E) occurs for the current mapping the next time the mapping is accessed. The mapping translations for the mapping that caused the access event to occur need to be validated. Accordingly, the driver must restore the device context for the process requesting access. Furthermore, the driver must call devmap_load(9F) on the handle of the mapping that generated the call to this entry point.

Accesses to portions of mappings that have had their mapping translations validated by a call to devmap_load() do not generate a call to devmap_access(). A subsequent call to devmap_unload() invalidates the mapping translations. This call enables devmap_access() to be called again.

If either devmap_load() or devmap_unload() returns an error, devmap_contextmgt() should immediately return that error. If the device driver encounters a hardware failure while restoring a device context, a -1 should be returned. Otherwise, after successfully handling the access request, devmap_contextmgt() should return zero. A return of other than zero from devmap_contextmgt() causes a SIGBUS or SIGSEGV to be sent to the process.

The following example shows how to manage a one-page device context.

---

**Note** – xxctxsave() and xxctxrestore() are device-dependent context save and restore functions. xxctxsave() reads data from the registers and saves the data in the soft state structure. xxctxrestore() takes data that is saved in the soft state structure and writes the data to device registers. Note that the read, write, and save are all performed with the DDI/DKI data access routines.

**EXAMPLE 11-3**  Using the devmap_contextmgt() Routine

```c
static int
xxdevmap_contextmgt(devmap_cookie_t handle, void *devprivate,
         offset_t off, size_t len, uint_t type, uint_t rw)
{
    int error;
    struct xxctx *ctxp = devprivate;
    struct xxstate *xsp = ctpx->xsp;
    mutex_enter(&xsp->ctx.lock);
    /* unload mapping for current context */
    if (xsp->current_ctx != NULL) {
        if (error = devmap_unload(xsp->current_ctx->handle,
                                   off, len)) != 0) {
```
EXAMPLE 11–3 Using the `devmap_contextmgt()` Routine  (Continued)

```c
xsp->current_ctx = NULL;
mutex_exit(&xsp->ctx_lock);
return (error);
}
/* Switch device context - device dependent */
if (xxctxsave(xsp->current_ctx, off, len) < 0) {
    xsp->current_ctx = NULL;
    mutex_exit(&xsp->ctx_lock);
    return (-1);
}
if (xxctxrestore(ctxp, off, len) < 0){
    xsp->current_ctx = NULL;
    mutex_exit(&xsp->ctx_lock);
    return (-1);
}
```
```
xsp->current_ctx = ctxp;
/* establish mapping for new context and return */
error = devmap_load(handle, off, len, type, rw);
if (error)
    xsp->current_ctx = NULL;
mutex_exit(&xsp->ctx_lock);
return (error);
}
```

**devmap_dup()** Entry Point

The `devmap_dup(9E)` entry point is called when a device mapping is duplicated, for example, by a user process that calls `fork(2)`. The driver is expected to generate new driver private data for the new mapping.

The syntax for `devmap_dup()` is as follows:

```c
int xxdevmap_dup(devmap_cookie_t handle, void *devprivate,
                 devmap_cookie_t new-handle, void **new-devprivate);
```

where:

- `handle`  Mapping handle of the mapping being duplicated.
- `new-handle`  Mapping handle of the mapping that was duplicated.
- `devprivate`  Pointer to the driver private data associated with the mapping being duplicated.
- `*new-devprivate`  Should be set to point to the new driver private data for the new mapping.

Mappings that have been created with `devmap_dup()` by default have their mapping translations invalidated. Invalid mapping translations force a call to the `devmap_access(9E)` entry point the first time the mapping is accessed.
The following example shows a typical devmap_dup() routine.

**EXAMPLE 11-4 Using the devmap_dup() Routine**

```c
static int
xxdevmap_dup(devmap_cookie_t handle, void *devprivate,
    devmap_cookie_t new_handle, void **new_devprivate)
{
    struct xxctx *ctxp = devprivate;
    struct xxstate *xsp = ctxp->xsp;
    /* Create a new context for the duplicated mapping */
    newctx = kmem_alloc(sizeof (struct xxctx), KM_SLEEP);
    newctx->xsp = xsp;
    newctx->handle = new_handle;
    newctx->offset = ctxp->offset;
    newctx->flags = ctxp->flags;
    newctx->len = ctxp->len;
    mutex_enter(&xsp->ctx_lock);
    if (ctxp->flags & MAP_PRIVATE) {
        newctx->context = kmem_alloc(XXCTX_SIZE, KM_SLEEP);
        bcopy(ctxp->context, newctx->context, XXCTX_SIZE);
    } else {
        newctx->context = xsp->ctx_shared;
    }
    mutex_exit(&xsp->ctx_lock);
    *new_devprivate = newctx;
    return(0);
}
```

**devmap_unmap() Entry Point**

The devmap_unmap(9E) entry point is called when a mapping is unmapped. Unmapping can be caused by a user process exiting or by calling the munmap(2) system call.

The syntax for devmap_unmap() is as follows:

```c
void xxdevmap_unmap(devmap_cookie_t handle, void *devprivate,
    offset_t off, size_t len, devmap_cookie_t new-handle1,
    void **new-devprivate1, devmap_cookie_t new-handle2,
    void **new-devprivate2);
```

where:

- `handle` Mapping handle of the mapping being freed.
- `devprivate` Pointer to the driver private data associated with the mapping.
- `off` Offset within the logical device memory at which the unmapping begins.
- `len` Length in bytes of the memory being unmapped.
- `new-handle1` Handle that the system uses to describe the new region that ends at `off` - 1. The value of `new-handle1` can be NULL.
new-devprivate1  Pointer to be filled in by the driver with the private driver mapping data for
the new region that ends at off-1. new-devprivate1 is ignored if
new-handle1 is NULL.

new-handle2  Handle that the system uses to describe the new region that begins at off +
len. The value of new-handle2 can be NULL.

new-devprivate2  Pointer to be filled in by the driver with the driver private mapping data for
the new region that begins at off + len. new-devprivate2 is ignored if
new-handle2 is NULL.

The devmap_unmap() routine is expected to free any driver private resources that were allocated
when this mapping was created, either by devmap_map(9E) or by devmap_dup(9E). If the
mapping is only partially unmapped, the driver must allocate new private data for the
remaining mapping before freeing the old private data. Calling devmap_unload(9F) on the
handle of the freed mapping is not necessary, even if this handle points to the mapping with the
valid translations. However, to prevent future devmap_access(9E) problems, the device driver
should make sure the current mapping representation is set to “no current mapping”.

The following example shows a typical devmap_unmap() routine.

EXAMPLE 11–5  Using the devmap_unmap() Routine

static void
xxdevmap_unmap(devmap_cookie_t handle, void *devprivate,
    offset_t off, size_t len, devmap_cookie_t new_handle1,
    void **new_devprivate1, devmap_cookie_t new_handle2,
    void **new_devprivate2)
{
    struct xxctx *ctxp = devprivate;
    struct xxstate *xsp = ctxp->xsp;
    mutex_enter(&xsp->ctx_lock);
    /*
     * If new_handle1 is not NULL, we are unmapping
     * at the end of the mapping.
     */
    if (new_handle1 != NULL) {
        /* Create a new context structure for the mapping */
        newctx = kmem_alloc(sizeof (struct xxctx), KM_SLEEP);
        newctx->xsp = xsp;
        if (ctxp->flags & MAP_PRIVATE) {
            /* allocate memory for the private context and copy it */
            newctx->context = kmem_alloc(XXCTX_SIZE, KM_SLEEP);
            bcopy(ctxp->context, newctx->context, XXCTX_SIZE);
        } else {
            /* point to the shared context */
            newctx->context = xsp->ctx_shared;
        }
        newctx->handle = new_handle1;
        newctx->offset = ctxp->offset;
        newctx->len = off - ctxp->offset;
        *new_devprivate1 = newctx;
    }
}
/*
   * If new_handle2 is not NULL, we are unmapping
   * at the beginning of the mapping.
   */
if (new_handle2 != NULL) {
    /* Create a new context for the mapping */
    newctx = kmem_alloc(sizeof (struct xxctx), KM_SLEEP);
    newctx->xsp = xsp;
    if (ctxp->flags & MAP_PRIVATE) {
        newctx->context = kmem_alloc(XXCTX_SIZE, KM_SLEEP);
        bcopy(ctxp->context, newctx->context, XXCTX_SIZE);
    } else {
        newctx->context = xsp->ctx_shared;
    }
    newctx->handle = new_handle2;
    newctx->offset = off + len;
    newctx->flags = ctxp->flags;
    newctx->len = ctxp->len - (off + len - ctxp->off);
    *new_devprivate2 = newctx;
    }
    if (xsp->current_ctx == ctxp)
        xsp->current_ctx = NULL;
    mutex_exit(&xsp->ctx_lock);
    if (ctxp->flags & MAP_PRIVATE)
        kmem_free(ctxp->context, XXCTX_SIZE);
    kmem_free(ctxp, sizeof (struct xxctx));
}

**Associating User Mappings With Driver Notifications**

When a user process requests a mapping to a device with `mmap(2)`, the driver's `segmap(9E)` entry point is called. The driver must use `ddi_devmap_segmap(9F)` or `devmap_setup(9F)` when setting up the memory mapping if the driver needs to manage device contexts. Both functions call the driver's `devmap(9E)` entry point, which uses `devmap_devmem_setup(9F)` to associate the device memory with the user mapping. See Chapter 10, “Mapping Device and Kernel Memory,” for details on how to map device memory.

The driver must inform the system of the `devmap_callback_ctl(9S)` entry points to get notifications of accesses to the user mapping. The driver informs the system by providing a pointer to a `devmap_callback_ctl(9S)` structure to `devmap_devmem_setup(9F)`. A `devmap_callback_ctl(9S)` structure describes a set of entry points for context management. These entry points are called by the system to notify a device driver to manage events on the device mappings.

The system associates each mapping with a mapping handle. This handle is passed to each of the entry points for context management. The mapping handle can be used to invalidate and validate the mapping translations. If the driver invalidates the mapping translations, the driver
will be notified of any future access to the mapping. If the driver validates the mapping translations, the driver will no longer be notified of accesses to the mapping. Mappings are always created with the mapping translations invalidated so that the driver will be notified on first access to the mapping.

The following example shows how to set up a mapping using the device context management interfaces.

**EXAMPLE 11-6  devmap(9E) Entry Point With Context Management Support**

static struct devmap_callback_ctl xx_callback_ctl = {
    DEVMAP_OPS_REV, xxdevmap_map, xxdevmap_access,
    xxdevmap_dup, xxdevmap_unmap
};

static int
xxdevmap(dev_t dev, devmap_cookie_t handle, offset_t off,
    size_t len, size_t *maplen, uint_t model)
{
    struct xxstate *xsp;
    uint_t rnumber;
    int  error;

    /* Setup data access attribute structure */
    struct ddi_device_acc_attr xx_acc_attr = {
        DDI_DEVICE_ATTR_V0,
        DDI_NEVERSWAP_ACC,
        DDI_STRICTORDER_ACC
    };
    xsp = ddi_get_soft_state(statep, getminor(dev));
    if (xsp == NULL) return (ENXIO);
    len = ptob(btopr(len));
    rnumber = 0;
    /* Set up the device mapping */
    error = devmap_devmem_setup(handle, xsp->dip, &xx_callback_ctl,
        rnumber, off, len, PROT_ALL, 0, &xx_acc_attr);
    *maplen = len;
    return (error);
}

**Managing Mapping Accesses**

The device driver is notified when a user process accesses an address in the memory-mapped region that does not have valid mapping translations. When the access event occurs, the mapping translations of the process that currently has access to the device must be invalidated. The device context of the process that requested access to the device must be restored. Furthermore, the translations of the mapping of the process requesting access must be validated.
The functions `devmap_load(9F)` and `devmap_unload(9F)` are used to validate and invalidate mapping translations.

**devmap_load() Entry Point**

The syntax for `devmap_load(9F)` is as follows:

```c
int devmap_load(devmap_cookie_t handle, offset_t offset,
                 size_t len, uint_t type, uint_t rw);
```

`devmap_load()` validates the mapping translations for the pages of the mapping specified by `handle`, `offset`, and `len`. By validating the mapping translations for these pages, the driver is telling the system not to intercept accesses to these pages of the mapping. Furthermore, the system must not allow accesses to proceed without notifying the device driver.

`devmap_load()` must be called with the `offset` and the handle of the mapping that generated the access event for the access to complete. If `devmap_load(9F)` is not called on this handle, the mapping translations are not validated, and the process receives a `SIGBUS`.

**devmap_unload() Entry Point**

The syntax for `devmap_unload(9F)` is as follows:

```c
int devmap_unload(devmap_cookie_t handle, offset_t offset, size_t len);
```

`devmap_unload()` invalidates the mapping translations for the pages of the mapping specified by `handle`, `offset`, and `len`. By invalidating the mapping translations for these pages, the device driver is telling the system to intercept accesses to these pages of the mapping. Furthermore, the system must notify the device driver the next time that these mapping pages are accessed by calling the `devmap_access(9E)` entry point.

For both functions, requests affect the entire page that contains the `offset` and all pages up to and including the entire page that contains the last byte, as indicated by `offset + len`. The device driver must ensure that for each page of device memory being mapped, only one process has valid translations at any one time.

Both functions return zero if successful. If, however, an error occurred in validating or invalidating the mapping translations, that error is returned to the device driver. The device driver must return this error to the system.
Power Management

Power management provides the ability to control and manage the electrical power usage of a computer system or device. Power management enables systems to conserve energy by using less power when idle and by shutting down completely when not in use. For example, desktop computer systems can use a significant amount of power and often are left idle, particularly at night. Power management software can detect that the system is not being used. Accordingly, power management can power down the system or some of its components.

This chapter provides information on the following subjects:

- “Power Management Framework” on page 201
- “Device Power Management Model” on page 203
- “System Power Management Model” on page 210
- ”Power Management Device Access Example” on page 215
- “Power Management Flow of Control” on page 216

Power Management Framework

The Oracle Solaris Power Management framework depends on device drivers to implement device-specific power management functions. The framework is implemented in two parts:

- Device power management – Automatically turns off unused devices to reduce power consumption
- System power management – Automatically turns off the computer when the entire system is idle
**Device Power Management**

The framework enables devices to reduce their energy consumption after a specified idle time interval. As part of power management, system software checks for idle devices. The Power Management framework exports interfaces that enable communication between the system software and the device driver.

The Oracle Solaris Power Management framework provides the following features for device power management:

- A device-independent model for power-manageable devices.
- `dtpower(1M)`, a tool for configuring workstation power management.
- A set of DDI interfaces for notifying the framework about power management compatibility and idleness state.

**System Power Management**

System power management involves saving the state of the system prior to powering the system down. Thus, the system can be returned to the same state immediately when the system is turned back on.

To shut down an entire system with return to the state prior to the shutdown, take the following steps:

- Stop kernel threads and user processes. Restart these threads and processes later.
- Save the hardware state of all devices on the system to disk. Restore the state later.

---

*SPARC only –* System power management is currently implemented only on some SPARC systems supported by the Oracle Solaris OS.

The System Power Management framework in the Oracle Solaris OS provides the following features for system power management:

- A platform-independent model of system idleness.
- A set of interfaces for the device driver to override the method for determining which drivers have hardware state.
- A set of interfaces to enable the framework to call into the driver to save and restore the device state.
- A mechanism for notifying processes that a resume operation has occurred.
Device Power Management Model

The following sections describe the details of the device power management model. This model includes the following elements:

- Components
- Idleness
- Power levels
- Dependency
- Policy
- Device power management interfaces
- Power management entry points

Power Management Components

A device is power manageable if the power consumption of the device can be reduced when the device is idle. Conceptually, a power-manageable device consists of a number of power-manageable hardware units that are called components.

The device driver notifies the system about device components and their associated power levels. Accordingly, the driver creates a `pm-components(9P)` property in the driver’s `attach(9E)` entry point as part of driver initialization.

Most devices that are power manageable implement only a single component. An example of a single-component, power-manageable device is a disk whose spindle motor can be stopped to save power when the disk is idle.

If a device has multiple power-manageable units that are separately controllable, the device should implement multiple components.

An example of a two-component, power-manageable device is a frame buffer card with a monitor. Frame buffer electronics is the first component [component 0]. The frame buffer’s power consumption can be reduced when not in use. The monitor is the second component [component 1]. The monitor can also enter a lower power mode when the monitor is not in use. The frame buffer electronics and monitor are considered by the system as one device with two components.

Multiple Power Management Components

To the power management framework, all components are considered equal and completely independent of each other. If the component states are not completely compatible, the device driver must ensure that undesirable state combinations do not occur. For example, a frame buffer/monitor card has the following possible states: D0, D1, D2, and D3. The monitor attached to the card has the following potential states: On, Standby, Suspend, and Off. These states are not necessarily compatible with each other. For example, if the monitor is On, then the frame
buffer must be at D0, that is, full on. If the frame buffer driver gets a request to power up the
monitor to On while the frame buffer is at D3, the driver must call `pm_raise_power(9F)` to bring
the frame buffer up before setting the monitor On. System requests to lower the power of the
frame buffer while the monitor is On must be refused by the driver.

**Power Management States**

Each component of a device can be in one of two states: *busy* or *idle*. The device driver notifies
the framework of changes in the device state by calling `pm_busy_component(9F)` and
`pm_idle_component(9F)`. When components are initially created, the components are
considered idle.

**Power Levels**

From the `pm-components` property exported by the device, the Device Power Management
framework knows what power levels the device supports. Power-level values must be positive
integers. The interpretation of power levels is determined by the device driver writer. Power
levels must be listed in monotonically increasing order in the `pm-components` property. A
power level of 0 is interpreted by the framework to mean off. When the framework must power
up a device due to a dependency, the framework sets each component at its highest power level.

The following example shows a `pm-components` entry from the `.conf` file of a driver that
implements a single power-managed component consisting of a disk spindle motor. The disk
spindle motor is component 0. The spindle motor supports two power levels. These levels
represent “stopped” and “spinning at full speed.”

**EXAMPLE 12–1  Sample pm-component Entry**

```
pm-components="NAME=Spindle Motor", "0=Stopped", "1=Full Speed"
```

The following example shows how **Example 12–1** could be implemented in the `attach()`
routine of the driver.

**EXAMPLE 12–2  attach(9E) Routine With pm-components Property**

```
static char *pmcomps[] = {
    "NAME=Spindle Motor",
    "0=Stopped",
    "1=Full Speed"
};
/* ... */
xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    /* ... */
    if (ddi_prop_update_string_array(DDI_DEV_T_NONE, dip,
        "pm-components", &pmcomp[0],
```
The following example shows a frame buffer that implements two components. Component 0 is the frame buffer electronics that support four different power levels. Component 1 represents the state of power management of the attached monitor.

When a device driver is first attached, the framework does not know the power level of the device. A power transition can occur when:

- The driver calls `pm_raise_power(9F)` or `pm_lower_power(9F)`.  
- The framework has lowered the power level of a component because a time threshold has been exceeded.  
- Another device has changed power and a dependency exists between the two devices. See “Power Management Dependencies” on page 205.

After a power transition, the framework begins tracking the power level of each component of the device. Tracking also occurs if the driver has informed the framework of the power level. The driver informs the framework of a power level change by calling `pm_power_has_changed(9F)`.

The system calculates a default threshold for each potential power transition. These thresholds are based on the system idleness threshold. Another default threshold based on the system idleness threshold is used when the component power level is unknown.

**Power Management Dependencies**

Some devices should be powered down only when other devices are also powered down. For example, if a CD-ROM drive is allowed to power down, necessary functions, such as the ability to eject a CD, might be lost.

To prevent a device from powering down independently, you can make that device dependent on another device that is likely to remain powered on. Typically, a device is made dependent upon a frame buffer, because a monitor is generally on whenever a user is utilizing a system.
Where dependent-phys-path is the device that is kept powered up, such as the CD-ROM drive. phys-path represents the device whose power state is to be depended on, such as the frame buffer.

The following syntax enables you to indicate dependency in a general fashion:

device-dependency-property property phys-path

Such an entry mandates that any device that exports the property property must be dependent upon the device named by phys-path. Because this dependency applies especially to removable-media devices, /etc/power.conf includes the following line by default:

device_dependent-property removable-media /dev/fb

With this syntax, no device that exports the removable-media property can be powered down unless the console frame buffer is also powered down.

For more information, see the removable-media(9P) man page.

**Automatic Power Management for Devices**

If automatic power management is enabled, then all devices with a pm-components(9P) property automatically will use power management. After a component has been idle for a default period, the component is automatically lowered to the next lowest power level. The default period is calculated by the power management framework to set the entire device to its lowest power state within the system idleness threshold.

**Note** – By default, automatic power management is enabled on all SPARC desktop systems first shipped after July 1, 1999. This feature is disabled by default for all other systems.

**Device Power Management Interfaces**

A device driver that supports a device with power-manageable components must create a pm-components(9P) property. This property indicates to the system that the device has power-manageable components. pm-components also tells the system which power levels are available. The driver typically informs the system by calling 
ddi_prop_update_string_array(9F) from the driver's attach(9E) entry point. An alternative means of informing the system is from a driver.conf(4) file. See the pm-components(9P) man page for details.

**Busy-Idle State Transitions**

The driver must keep the framework informed of device state transitions from idle to busy or busy to idle. Where these transitions happen is entirely device-specific. The transitions between the busy and idle states depend on the nature of the device and the abstraction represented by
the specific component. For example, SCSI disk target drivers typically export a single component, which represents whether the SCSI target disk drive is spun up or not. The component is marked busy whenever an outstanding request to the drive exists. The component is marked idle when the last queued request finishes. Some components are created and never marked busy. For example, components created by `pm-components(9P)` are created in an idle state.

The `pm_busy_component(9F)` and `pm_idle_component(9F)` interfaces notify the power management framework of busy-idle state transitions. The `pm_busy_component(9F)` call has the following syntax:

```c
int pm_busy_component(dev_info_t *dip, int component);
```

`pm_busy_component(9F)` marks `component` as busy. While the component is busy, that component should not be powered off. If the component is already powered off, then marking that component busy does not change the power level. The driver needs to call `pm_raise_power(9F)` for this purpose. Calls to `pm_busy_component(9F)` are cumulative and require a corresponding number of calls to `pm_idle_component` to idle the component.

The `pm_idle_component(9F)` routine has the following syntax:

```c
int pm_idle_component(dev_info_t *dip, int component);
```

`pm_idle_component(9F)` marks `component` as idle. An idle component is subject to being powered off. `pm_idle_component(9F)` must be called once for each call to `pm_busy_component(9F)` in order to idle the component.

**Device Power State Transitions**

A device driver can call `pm_raise_power(9F)` to request that a component be set to at least a given power level. Setting the power level in this manner is necessary before using a component that has been powered off. For example, the `read(9E)` routine of a SCSI disk target driver might need to spin up the disk, if the disk has been powered off. The `pm_raise_power(9F)` function requests the power management framework to initiate a device power state transition to a higher power level. Normally, reductions in component power levels are initiated by the framework. However, a device driver should call `pm_lower_power(9F)` when detaching, in order to reduce the power consumption of unused devices as much as possible.

Powering down can pose risks for some devices. For example, some tape drives damage tapes when power is removed. Similarly, some disk drives have a limited tolerance for power cycles, because each cycle results in a head landing. Use the `no-involuntary-power-cycles(9P)` property to notify the system that the device driver should control all power cycles for the device. This approach prevents power from being removed from a device while the device driver is detached unless the device was powered off by a driver's call to `pm_lower_power(9F)` from its `detach(9E)` entry point.
The `pm_raise_power(9F)` function is called when the driver discovers that a component needed for some operation is at an insufficient power level. This interface causes the driver to raise the current power level of the component to the needed level. All the devices that depend on this device are also brought back to full power by this call.

Call the `pm_lower_power(9F)` function when the device is detaching once access to the device is no longer needed. Call `pm_lower_power(9F)` to set each component at the lowest power so that the device uses as little power as possible while not in use. The `pm_lower_power()` function must be called from the `detach()` entry point. The `pm_lower_power()` function has no effect if it is called from any other part of the driver.

The `pm_power_has_changed(9F)` function is called to notify the framework about a power transition. The transition might be due to the device changing its own power level. The transition might also be due to an operation such as suspend-resume. The syntax for `pm_power_has_changed(9F)` is the same as the syntax for `pm_raise_power(9F)`.

### power() Entry Point

The power management framework uses the `power(9E)` entry point.

`power()` uses the following syntax:

```c
int power(dev_info_t *dip, int component, int level);
```

When a component’s power level needs to be changed, the system calls the `power(9E)` entry point. The action taken by this entry point is device driver-specific. In the example of the SCSI target disk driver mentioned previously, setting the power level to 0 results in sending a SCSI command to spindown the disk, while setting the power level to the full power level results in sending a SCSI command to spin up the disk.

If a power transition can cause the device to lose state, the driver must save any necessary state in memory for later restoration. If a power transition requires the saved state to be restored before the device can be used again, then the driver must restore that state. The framework makes no assumptions about what power transactions cause the loss of state or require the restoration of state for automatically power-managed devices. The following example shows a sample `power()` routine.

**EXAMPLE 12-4** Using the `power()` Routine for a Single-Component Device

```c
int xxpower(dev_info_t *dip, int component, int level)
{
    struct xxstate *xsp;
    int instance;

    instance = ddi_get_instance(dip);
    xsp = ddi_get_soft_state(statep, instance);
```
EXAMPLE 12-4  Using the power() Routine for a Single-Component Device  (Continued)

/*
 * Make sure the request is valid
 */
if (!xx_valid_power_level(component, level))
    return (DDI_FAILURE);
mutex_enter(&xsp->mu);
/*
 * If the device is busy, don’t lower its power level
 */
if ((xsp->xx_busy[component] &&
    xsp->xx_power_level[component] > level) { mutex_exit(&xsp->mu);
    return (DDI_FAILURE);
}
if (xsp->xx_power_level[component] != level) {
    /*
     * device- and component-specific setting of power level
     * goes here
     */
    xsp->xx_power_level[component] = level;
    mutex_exit(&xsp->mu);
    return (DDI_SUCCESS);
}

The following example is a power() routine for a device with two components, where component 0 must be on when component 1 is on.

EXAMPLE 12-5  power(9E) Routine for Multiple-Component Device

int xxpower(dev_info_t *dip, int component, int level)
{
    struct xxstate *xsp;
    int instance;

    instance = ddi_get_instance(dip);
    xsp = ddi_get_soft_state(statep, instance);
    /*
     * Make sure the request is valid
     */
    if (!xx_valid_power_level(component, level))
        return (DDI_FAILURE);
    mutex_enter(&xsp->mu);
    /*
     * If the device is busy, don’t lower its power level
     */
    if (xsp->xx_busy[component] &&
        xsp->xx_power_level[component] > level) { mutex_exit(&xsp->mu);
        return (DDI_FAILURE);
    }
    /*
This code implements inter-component dependencies:
* If we are bringing up component 1 and component 0 is off, we must bring component 0 up first, and if
* we are asked to shut down component 0 while component 1 is up we must refuse
*/
if (component == 1 && level > 0 && xsp->xx_power_level[0] == 0) {
xsp->xx_busy[0]++;  
if (pm_busy_component(dip, 0) != DDI_SUCCESS) {
    /*
     * This can only happen if the args to
     * pm_busy_component()
     * are wrong, or pm-components property was not
     * exported by the driver.
     */
    xsp->xx_busy[0]--;
    mutex_exit(&xsp->mu);
    cmn_err(CE_WARN, "xxpower pm_busy_component()
        failed");
    return (DDI_FAILURE);
}
mutex_exit(&xsp->mu);
if (pm_raise_power(dip, 0, XX_FULL_POWER_0) != DDI_SUCCESS)
    return (DDI_FAILURE);
mutex_enter(&xsp->mu);
}
if (component == 0 && level == 0 && xsp->xx_power_level[1] != 0) {
    mutex_exit(&xsp->mu);
    return (DDI_FAILURE);
}
if (xsp->xx_power_level[component] != level) {
    /*
     * device- and component-specific setting of power level
     * goes here
     */
    xsp->xx_power_level[component] = level;
}
mux_exit(&xsp->mu);
return (DDI_SUCCESS);
}

System Power Management Model

This section describes the details of the System Power Management model. The model includes the following components:

- Autosubdown threshold
- Busy state
- Hardware state
- Policy
- Power management entry points
Autoshutdown Threshold

The system can be shut down, that is, powered off, automatically after a configurable period of idleness. This period is known as the autoshutdown threshold. This behavior is enabled by default for SPARC desktop systems first shipped after October 1, 1995 and before July 1, 1999.

Busy State

The busy state of the system can be measured in several ways. The currently supported built-in metric items are keyboard characters, mouse activity, tty characters, load average, disk reads, and NFS requests. Any one of these items can make the system busy. In addition to the built-in metrics, an interface is defined for running a user-specified process that can indicate that the system is busy.

Hardware State

Devices that export a reg property are considered to have hardware state that must be saved prior to shutting down the system. A device without the reg property is considered to be stateless. However, this consideration can be overridden by the device driver.

A device with hardware state but no reg property, such as a SCSI driver, must be called to save and restore the state if the driver exports a pm-hardware-state property with the value needs-suspend-resume. Otherwise, the lack of a reg property is taken to mean that the device has no hardware state. For information on device properties, see Chapter 4, "Properties."

A device with a reg property and no hardware state can export a pm-hardware-state property with the value no-suspend-resume. Using no-suspend-resume with the pm-hardware-state property keeps the framework from calling the driver to save and restore that state. For more information on power management properties, see the pm-components(9P) man page.

Automatic Power Management for Systems

The system is shut down if the system has been idle for autoshutdown threshold minutes:

Entry Points Used by System Power Management

System power management passes the command DDI_SUSPEND to the detach(9E) driver entry point to request the driver to save the device hardware state. System power management passes the command DDI_RESUME to the attach(9E) driver entry point to request the driver to restore the device hardware state.
**detach () Entry Point**

The syntax for `detach(9E)` is as follows:

```c
int detach(dev_info_t *dip, ddi_detach_cmd_t cmd);
```

A device with a `reg` property or a `pm-hardware-state` property set to `needs-suspend-resume` must be able to save the hardware state of the device. The framework calls into the driver’s `detach(9E)` entry point to enable the driver to save the state for restoration after the system power returns. To process the `DDI_SUSPEND` command, `detach(9E)` must perform the following tasks:

- Block further operations from being initiated until the device is resumed, except for `dump(9E)` requests.
- Wait until outstanding operations have completed. If an outstanding operation can be restarted, you can abort that operation.
- Cancel any timeouts and callbacks that are pending.
- Save any volatile hardware state to memory. The state includes the contents of device registers, and can also include downloaded firmware.

If the driver is unable to suspend the device and save its state to memory, then the driver must return `DDI_FAILURE`. The framework then aborts the system power management operation.

In some cases, powering down a device involves certain risks. For example, if a tape drive is powered off with a tape inside, the tape can be damaged. In such a case, `attach(9E)` should do the following:

- Call `ddi_removing_power(9F)` to determine whether a `DDI_SUSPEND` command can cause power to be removed from the device.
- Determine whether power removal can cause problems.

If both cases are true, the `DDI_SUSPEND` request should be rejected. Example 12–6 shows an `attach(9E)` routine using `ddi_removing_power(9F)` to check whether the `DDI_SUSPEND` command causes problems.

Dump requests must be honored. The framework uses the `dump(9E)` entry point to write out the state file that contains the contents of memory. See the `dump(9E)` man page for the restrictions that are imposed on the device driver when using this entry point.

Calling the `detach(9E)` entry point of a power-manageable component with the `DDI_SUSPEND` command should save the state when the device is powered off. The driver should cancel pending timeouts. The driver should also suppress any calls to `pm_raise_power(9F)` except for `dump(9E)` requests. When the device is resumed by a call to `attach(9E)` with a command of `DDI_RESUME`, timeouts and calls to `pm_raise_power()` can be resumed. The driver must keep sufficient track of its state to be able to deal appropriately with this possibility. The following example shows a `detach(9E)` routine with the `DDI_SUSPEND` command implemented.
EXAMPLE 12-6  detach(9E) Routine Implementing DDI_SUSPEND

```c
int
xxdetach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
    struct xxstate *xsp;
    int instance;

    instance = ddi_get_instance(dip);
    xsp = ddi_get_soft_state(statep, instance);

    switch (cmd) {
        case DDI_DETACH:
            /* ... */
            break;
        case DDI_SUSPEND:
            /* We do not allow DDI_SUSPEND if power will be removed and
               we have a device that damages tape when power is removed
               We do support DDI_SUSPEND for Device Reconfiguration. */
            if (ddi_removing_power(dip) && xxdamages_tape(dip))
                return (DDI_FAILURE);
            mutex_enter(&xsp->mu);
            xsp->xx_suspended = 1; /* stop new operations */
            /* Sleep waiting for all the commands to be completed *
            * If a callback is outstanding which cannot be cancelled
            * then either wait for the callback to complete or fail the
            * suspend request *
            * This section is only needed if the driver maintains a
            * running timeout */
            if (xsp->xx_timeout_id) {
                timeout_id_t temp_timeout_id = xsp->xx_timeout_id;
                xsp->xx_timeout_id = 0;
                mutex_exit(&xsp->mu);
                untimeout(temp_timeout_id);
                mutex_enter(&xsp->mu);
            }
            if (!xsp->xx_state_saved) {
                /* Save device register contents into
                * xsp->xx_device_state
                */
                mutex_exit(&xsp->mu);
                return (DDI_SUCCESS);
            }
            break;
        default:
            return (DDI_FAILURE);
    }
}
```

attach() Entry Point

The syntax for `attach(9E)` is as follows:

```c
int attach(dev_info_t *dip, ddi_attach_cmd_t cmd);
```
When power is restored to the system, each device with a reg property or with a pm-hardware-state property of value needs-suspend-resume has its attach(9E) entry point called with a command value of DDI_RESUME. If the system shutdown is aborted, each suspended driver is called to resume even though the power has not been shut off. Consequently, the resume code in attach(9E) must make no assumptions about whether the system actually lost power.

The power management framework considers the power level of the components to be unknown at DDI_RESUME time. Depending on the nature of the device, the driver writer has two choices:

- If the driver can determine the actual power level of the components of the device without powering the components up, such as by reading a register, then the driver should notify the framework of the power level of each component by calling pm_power_has_changed(9F).
- If the driver cannot determine the power levels of the components, then the driver should mark each component internally as unknown and call pm_raise_power(9F) before the first access to each component.

The following example shows an attach(9E) routine with the DDI_RESUME command.

**EXAMPLE 12–7** attach(9E) Routine Implementing DDI_RESUME

```c
int
xxattach(devinfo_t *dip, ddi_attach_cmd_t cmd)
{
    struct xxstate *xsp;
    int instance;

    instance = ddi_get_instance(dip);
    xsp = ddi_get_soft_state(statep, instance);

    switch (cmd) {
        case DDI_ATTACH:
            /* ... */
        case DDI_RESUME:
            /* ... */
                mutex_enter(&xsp->mu);
                if (xsp->xx_pm_state_saved) {
                    /* Restore device register contents from
                    * xsp->xx_device_state
                    */
                }
                /*
                * This section is optional and only needed if the
                * driver maintains a running timeout
                */
                xsp->xx_timeout_id = timeout( /* ... */ );
                xsp->xx_suspended = 0; /* allow new operations */
                cv_broadcast(&xsp->xx_suspend_cv);
                /* If it is possible to determine in a device-specific
                * way what the power levels of components are without
                * powering the components up,
```
Note – The detach(9E) and attach(9E) interfaces can also be used to resume a system that has been quiesced.

Power Management Device Access Example

If power management is supported, and detach(9E) and attach(9E) are used as in Example 12–6 and Example 12–7, then access to the device can be made from user context, for example, from read(2), write(2), and ioctl(2).

The following example demonstrates this approach. The example assumes that the operation about to be performed requires a component component that is operating at power level level.

EXAMPLE 12–8  Device Access
mutex_enter(&xsp->mu);
/* * Block command while device is suspended by DDI_SUSPEND */
while (xsp->xx_suspended)
  cv_wait(&xsp->xx_suspend_cv, &xsp->mu);
/* * Mark component busy so xx_power() will reject attempt to lower power */
xsp->xx_busy[component]++;
if (pm_busy_component(dip, component) != DDI_SUCCESS) {
  xsp->xx_busy[component]--;
  /* * Log error and abort */
}
if (xsp->xx_power_level[component] < level) {
  mutex_exit(&xsp->mu);
  if (pm_raise_power(dip, component, level) != DDI_SUCCESS) {
The code fragment in the following example can be used when device operation completes, for example, in the device’s interrupt handler.

**EXAMPLE 12–9  Device Operation Completion**

```c
/*
 * For each command completion, decrement the busy count and unstack
 * the pm_busy_component() call by calling pm_idle_component(). This
 * will allow device power to be lowered when all commands complete
 * (all pm_busy_component() counts are unstacked)
 */
xsp->xx_busy[component]--;  
if (pm_idle_component(dip, component) != DDI_SUCCESS) {
    xsp->xx_busy[component]++;  
    /* Log error and abort
    */
}
/*
 * If no more outstanding commands, wake up anyone (like DDI_SUSPEND)
 * waiting for all commands to be completed
 */
```

**Power Management Flow of Control**

Figure 12–1 illustrates the flow of control in the power management framework.

When a component’s activity is complete, a driver can call pm_idle_component(9F) to mark the component as idle. When the component has been idle for its threshold time, the framework can lower the power of the component to its next lower level. The framework calls the power(9E) function to set the component’s power to the next lower supported power level, if a lower level exists. The driver’s power(9E) function should reject any attempt to lower the power level of a component when that component is busy. The power(9E) function should save any state that could be lost in a transition to a lower level prior to making that transition.

When the component is needed at a higher level, the driver calls pm_busy_component(9F). This call keeps the framework from lowering the power still further and then calls pm_raise_power(9F) on the component. The framework next calls power(9E) to raise the power of the component before the call to pm_raise_power(9F) returns. The driver’s power(9E) code must restore any state that was lost in the lower level but that is needed in the higher level.
When a driver is detaching, the driver should call `pm_lower_power(9F)` for each component to lower its power to its lowest level. The framework can then call the driver's `power(9E)` routine to lower the power of the component before the call to `pm_lower_power(9F)` returns.

**Note:**

9E routines are always called by the framework.
9F routines are always called by the driver.
Changes to Power Management Interfaces

Prior to the Solaris 8 release, power management of devices was not automatic. Developers had to add an entry to `/etc/power.conf` for each device that was to be power-managed. The framework assumed that all devices supported only two power levels: 0 and standard power.

Power assumed an implied dependency of all other components on component 0. When component 0 changed to level 0, a call was made into the driver’s `detach(9E)` with the `DDI_PM_SUSPEND` command to save the hardware state. When component 0 changed from level 0, a call was made to the `attach(9E)` routine with the command `DDI_PM_RESUME` to restore hardware state.

The following interfaces and commands are obsolete, although they are still supported for binary purposes:

- `ddi_dev_is_needed(9F)`
- `pm_create_components(9F)`
- `pm_destroy_components(9F)`
- `pm_get_normal_power(9F)`
- `pm_set_normal_power(9F)`
- `DDI_PM_SUSPEND`
- `DDI_PM_RESUME`

Since the Solaris 8 release, devices that export the `pm-components` property automatically use power management if `autopm` is enabled.

The framework now knows from the `pm-components` property which power levels are supported by each device.

The framework makes no assumptions about dependencies among the different components of a device. The device driver is responsible for saving and restoring hardware state as needed when changing power levels.

These changes enable the power management framework to deal with emerging device technology. Power management now results in greater power savings. The framework can detect automatically which devices can save power. The framework can use intermediate power states of the devices. A system can now meet energy consumption goals without powering down the entire system and without any functions.

<table>
<thead>
<tr>
<th>Removed Interfaces</th>
<th>Equivalent Interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pm_create_components(9F)</code></td>
<td><code>pm-components(9P)</code></td>
</tr>
<tr>
<td><code>pm_set_normal_power(9F)</code></td>
<td><code>pm-components(9P)</code></td>
</tr>
<tr>
<td><code>pm_destroy_components(9F)</code></td>
<td>None</td>
</tr>
</tbody>
</table>
## Changes to Power Management Interfaces

### TABLE 12-1  Power Management Interfaces  (Continued)

<table>
<thead>
<tr>
<th>Removed Interfaces</th>
<th>Equivalent Interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pm_get_normal_power(9F)</code></td>
<td><code>None</code></td>
</tr>
<tr>
<td><code>ddi_dev_is_needed(9F)</code></td>
<td><code>pm_raise_power(9F)</code></td>
</tr>
<tr>
<td><code>None</code></td>
<td><code>pm_lower_power(9F)</code></td>
</tr>
<tr>
<td><code>None</code></td>
<td><code>pm_power_has_changed(9F)</code></td>
</tr>
<tr>
<td><code>DDI_PM_SUSPEND</code></td>
<td><code>None</code></td>
</tr>
<tr>
<td><code>DDI_PM_RESUME</code></td>
<td><code>None</code></td>
</tr>
</tbody>
</table>
Fault Management Architecture (FMA) I/O Fault Services enable driver developers to integrate fault management capabilities into I/O device drivers. The Oracle Solaris I/O fault services framework defines a set of interfaces that enable all drivers to coordinate and perform basic error handling tasks and activities. The Oracle Solaris FMA as a whole provides for error handling and fault diagnosis, in addition to response and recovery. FMA is a component of Oracle's Predictive Self-Healing strategy.

A driver is considered hardened when it uses the defensive programming practices described in this document in addition to the I/O fault services framework for error handling and diagnosis. The driver hardening test harness tests that the I/O fault services and defensive programming requirements have been correctly fulfilled.

This document contains the following section:

- “Oracle Fault Management Architecture I/O Fault Services” on page 221 provides a reference for driver developers who want to integrate fault management capabilities into I/O device drivers.

**Oracle Fault Management Architecture I/O Fault Services**

This section explains how to integrate fault management error reporting, error handling, and diagnosis for I/O device drivers. This section provides an in-depth examination of the I/O fault services framework and how to utilize the I/O fault service APIs within a device driver.

This section discusses the following topics:

- “Oracle Solaris Fault Manager” on page 222 describes additional background with a focus on a high-level overview of the Oracle Solaris Fault Manager, fmd(1M).
What Is Predictive Self-Healing?

Traditionally, systems have exported hardware and software error information directly to human administrators and to management software in the form of syslog messages. Often, error detection, diagnosis, reporting, and handling was embedded in the code of each driver.

A system like the Solaris OS predictive self-healing system is first and foremost self-diagnosing. Self-diagnosing means the system provides technology to automatically diagnose problems from observed symptoms, and the results of the diagnosis can then be used to trigger automated response and recovery. A fault in hardware or a defect in software can be associated with a set of possible observed symptoms called errors. The data generated by the system as the result of observing an error is called an error report or ereport.

In a system capable of self-healing, ereports are captured by the system and are encoded as a set of name-value pairs described by an extensible event protocol to form an ereport event. Ereport events and other data are gathered to facilitate self-healing, and are dispatched to software components called diagnosis engines designed to diagnose the underlying problems corresponding to the error symptoms observed by the system. A diagnosis engine runs in the background and silently consumes error telemetry until it can produce a diagnosis or predict a fault.

After processing sufficient telemetry to reach a conclusion, a diagnosis engine produces another event called a fault event. The fault event is then broadcast to all agents that are interested in the specific fault event. An agent is a software component that initiates recovery and responds to specific fault events. A software component known as the Oracle Solaris Fault Manager, fmd(1M), manages the multiplexing of events between ereport generators, diagnosis engines, and agent software.

Oracle Solaris Fault Manager

The Oracle Solaris Fault Manager, fmd(1M), is responsible for dispatching in-bound error telemetry events to the appropriate diagnosis engines. The diagnosis engine is responsible for identifying the underlying hardware faults or software defects that are producing the error symptoms. The fmd(1M) daemon is the Oracle Solaris OS implementation of a fault manager. It starts at boot time and loads all of the diagnosis engines and agents available on the system. The Oracle Solaris Fault Manager also provides interfaces for system administrators and service personnel to observe fault management activity.
Diagnosis, Suspect Lists, and Fault Events

Once a diagnosis has been made, the diagnosis is output in the form of a list.suspect event. A list.suspect event is an event comprised of one or more possible fault or defect events. Sometimes the diagnosis cannot narrow the cause of errors to a single fault or defect. For example, the underlying problem might be a broken wire connecting controllers to the main system bus. The problem might be with a component on the bus or with the bus itself. In this specific case, the list.suspect event will contain multiple fault events: one for each controller attached to the bus, and one for the bus itself.

In addition to describing the fault that was diagnosed, a fault event also contains four payload members for which the diagnosis is applicable.

- The resource is the component that was diagnosed as faulty. The `fmdump(1M)` command shows this payload member as "Problem in."
- The Automated System Recovery Unit (ASRU) is the hardware or software component that must be disabled to prevent further error symptoms from occurring. The `fmdump(1M)` command shows this payload member as "Affects."
- The Field Replaceable Unit (FRU) is the component that must be replaced or repaired to fix the underlying problem.
- The Label payload is a string that gives the location of the FRU in the same form as it is printed on the chassis or motherboard, for example next to a DIMM slot or PCI card slot. The `fmdump` command shows this payload member as "Location."

For example, after receiving a certain number of ECC correctable errors in a given amount of time for a particular memory location, the CPU and memory diagnosis engine issues a diagnosis (list.suspect event) for a faulty DIMM.

```
# fmdump -v -u 38bd6f1b-a4de-4c21-db4e-ccd26fa8573c
TIME UUID SUNW-MSG-ID
Oct 31 13:40:18.1864 38bd6f1b-a4de-4c21-db4e-ccd26fa8573c AMD-8000-8L
100% fault.cpu.amd.icachetag
Problem in: hc:///motherboard=0/chip=0/cpu=0
Affects: cpu:///cpuid=0
FRU: hc:///motherboard=0/chip=0
Location: SLOT 2
```

In this example, `fmd(1M)` has identified a problem in a resource, specifically a CPU (hc:///motherboard=0/chip=0/cpu=0). To suppress further error symptoms and to prevent an uncorrectable error from occurring, an ASRU, (cpu:///cpuid=0), is identified for retirement. The component that needs to be replaced is the FRU (hc:///motherboard=0/chip=0).

Response Agents

An agent is a software component that takes action in response to a diagnosis or repair. For example, the CPU and memory retire agent is designed to act on list.suspects that contain a
fault.cpu.* event. The cpumem-retire agent will attempt to off-line a CPU or retire a physical memory page from service. If the agent is successful, an entry in the fault manager’s ASRU cache is added for the page or CPU that was successfully retired. The \texttt{fmadm(1M)} utility, as shown in the example below, shows an entry for a memory rank that has been diagnosed as having a fault. ASRUs that the system does not have the ability to off-line, retire, or disable, will also have an entry in the ASRU cache, but they will be seen as degraded. Degraded means the resource associated with the ASRU is faulty, but the ASRU is unable to be removed from service. Currently Oracle Solaris agent software cannot act upon I/O ASRUs (device instances). All faulty I/O resource entries in the cache are in the degraded state.

```
# fmadm faulty
STATE RESOURCE / UUID
-------- ----------------------------------------------------------------------
degraded mem:///motherboard=0/chip=1/memory-controller=0/dimm=3/rank=0
ccae89df-2217-4f5c-add4-d920f78b4faf
-------- ----------------------------------------------------------------------
```

The primary purpose of a retire agent is to isolate (safely remove from service) the piece of hardware or software that has been diagnosed as faulty.

Agents can also take other important actions such as the following actions:

- Send alerts via SNMP traps. This can translate a diagnosis into an alert for SNMP that plugs into existing software mechanisms.
- Post a syslog message. Message specific diagnoses (for example, syslog message agent) can take the result of a diagnosis and translate it into a syslog message that administrators can use to take a specific action.
- Other agent actions such as update the FRUID. Response agents can be platform-specific.

**Message IDs and Dictionary Files**

The syslog message agent takes the output of the diagnosis (the list.suspect event) and writes specific messages to the console or \texttt{/var/adm/messages}. Often console messages can be difficult to understand. FMA remedies this problem by providing a defined fault message structure that is generated every time a list.suspect event is delivered to a syslog message.

The syslog agent generates a message identifier (MSG ID). The event registry generates dictionary files (.\texttt{dict} files) that map a list.suspect event to a structured message identifier that should be used to identify and view the associated knowledge article. Message files (.\texttt{po} files) map the message ID to localized messages for every possible list of suspected faults that the diagnosis engine can generate. The following is an example of a fault message emitted on a test system.

```
SUNW-MSG-ID: AMD-8000-7U, TYPE: Fault, VER: 1, SEVERITY: Major
EVENT-TIME: Fri Jul 28 04:26:51 PDT 2006
PLATFORM: Sun Fire V40z, CSN: XG051535088, HOSTNAME: parity
SOURCE: eft, REV: 1.16
```
EVENT-ID: add96f65-5473-69e6-dbe1-8b3d00d5c47b
DESC: The number of errors associated with this CPU has exceeded acceptable levels. Refer to http://support.oracle.com/msg/SMF-8000-05 for more information.
AUTO-RESPONSE: An attempt will be made to remove this CPU from service.
IMPACT: Performance of this system may be affected.
REC-ACTION: Schedule a repair procedure to replace the affected CPU.
Use fmdump -v -u <EVENT_ID> to identify the module.

System Topology
To identify where a fault might have occurred, diagnosis engines need to have the topology for a given software or hardware system represented. The fmd(1M) daemon provides diagnosis engines with a handle to a topology snapshot that can be used during diagnosis. Topology information is used to represent the resource, ASRU, and FRU found in each fault event. The topology can also be used to store the platform label, FRUID, and serial number identification.

The resource payload member in the fault event is always represented by the physical path location from the platform chassis outward. For example, a PCI controller function that is bridged from the main system bus to a PCI local bus is represented by its hc scheme path name:

hc:///motherboard=0/hostbridge=1/pcibus=0/pcidev=13/pcifn=0

The ASRU payload member in the fault event is typically represented by the Oracle Solaris device tree instance name that is bound to a hardware controller, device, or function. FMA uses the dev scheme to represent the ASRU in its native format for actions that might be taken by a future implementation of a retire agent specifically designed for I/O devices:

dev://pci@1e,600000/ide@d

The FRU payload representation in the fault event varies depending on the closest replaceable component to the I/O resource that has been diagnosed as faulty. For example, a fault event for a broken embedded PCI controller might name the motherboard of the system as the FRU that needs to be replaced:

hc:///motherboard=0

The label payload is a string that gives the location of the FRU in the same form as it is printed on the chassis or motherboard, for example next to a DIMM slot or PCI card slot:

Label: SLOT 2

Error Handling
This section describes how to use I/O fault services APIs to handle errors within a driver. This section discusses how drivers should indicate and initialize their fault management capabilities, generate error reports, and register the driver's error handler routine.
Drivers that have been instrumented to provide FMA error report telemetry detect errors and determine the impact of those errors on the services provided by the driver. Following the detection of an error, the driver should determine when its services have been impacted and to what degree.

An I/O driver must respond immediately to detected errors. Appropriate responses include:

- Attempt recovery
- Retry an I/O transaction
- Attempt fail-over techniques
- Report the error to the calling application/stack
- If the error cannot be constrained any other way, then panic

Errors detected by the driver are communicated to the fault management daemon as an ereport. An ereport is a structured event defined by the FMA event protocol. The event protocol is a specification for a set of common data fields that must be used to describe all possible error and fault events, in addition to the list of suspected faults. Ereports are gathered into a flow of error telemetry and dispatched to the diagnosis engine.

**Declaring Fault Management Capabilities**

A hardened device driver must declare its fault management capabilities to the I/O Fault Management framework. Use the `ddi_fm_init(9F)` function to declare the fault management capabilities of your driver.

```c
void ddi_fm_init(dev_info_t *dip, int *fmcap, ddi_iblock_cookie_t *ibcp)
```

The `ddi_fm_init()` function can be called from kernel context in a driver `attach(9E)` or `detach(9E)` entry point. The `ddi_fm_init()` function usually is called from the `attach()` entry point. The `ddi_fm_init()` function allocates and initializes resources according to `fmcap`. The `fmcap` parameter must be set to the bitwise-inclusive-OR of the following fault management capabilities:

- `DDI_FM_EREPORT_CAPABLE` - Driver is responsible for and capable of generating FMA protocol error events (ereports) upon detection of an error condition.
- `DDI_FM_ACCCHK_CAPABLE` - Driver is responsible for and capable of checking for errors upon completion of one or more access I/O transactions.
- `DDI_FM_DMACHECK_CAPABLE` - Driver is responsible for and capable of checking for errors upon completion of one or more DMA I/O transactions.
- `DDI_FM_ERRCB_CAPABLE` - Driver has an error callback function.

A hardened leaf driver generally sets all these capabilities. However, if its parent nexus is not capable of supporting any one of the requested capabilities, the associated bit is cleared and returned as such to the driver. Before returning from `ddi_fm_init(9F)`, the I/O fault services framework creates a set of fault management capability properties: `fm-ereport-capable`, `fm-accchk-capable`, `fm-dmachck-capable` and `fm-errcb-capable`. The currently supported fault management capability level is observable by using the `prtconf(1M)` command.
To make your driver support administrative selection of fault management capabilities, export and set the fault management capability level properties to the values described above in the `driver.conf(4)` file. The `fm-capable` properties must be set and read prior to calling `ddi_fm_init()` with the desired capability list.

The following example from the bge driver shows the `bge_fm_init()` function, which calls the `ddi_fm_init(9F)` function. The `bge_fm_init()` function is called in the `bge_attach()` function.

```c
static void
bge_fm_init(bge_t *bgep) {
    ddi_iblock_cookie_t iblk;

    /* Only register with IO Fault Services if we have some capability */
    if (bgep->fm_capabilities) {
        bge_reg_accastr.devacc_attr_access = DDI_FLAGERR_ACC;
        dma_attr.dma_attr_flags = DDI_DMA_FLAGERR;
        /*
         * Register capabilities with IO Fault Services
         */
        ddi_fm_init(bgep->devinfo, &bgep->fm_capabilities, &iblk);
        /*
         * Initialize pci ereport capabilities if ereport capable
         */
        if (DDI_FM_EREPORT_CAP(bgep->fm_capabilities) ||
            DDI_FM_ERRCB_CAP(bgep->fm_capabilities))
            pci_ereport_setup(bgep->devinfo);
        /*
         * Register error callback if error callback capable
         */
        if (DDI_FM_ERRCB_CAP(bgep->fm_capabilities))
            ddi_fm_handler_register(bgep->devinfo,
                bge_fm_error_cb, (void*) bgep);
    } else {
        /*
         * These fields have to be cleared of FMA if there are no
         * FMA capabilities at runtime.
         */
        bge_reg_accastr.devacc_attr_access = DDI_DEFAULT_ACC;
        dma_attr.dma_attr_flags = 0;
    }
}
```

**Cleaning Up Fault Management Resources**

The `ddi_fm_fini(9F)` function cleans up resources allocated to support fault management for `dip`.

```c
void ddi_fm_fini(dev_info_t *dip)
```

The `ddi_fm_fini()` function can be called from kernel context in a driver `attach(9E)` or `detach(9E)` entry point.
The following example from the bge driver shows the bge_fm_fini() function, which calls the ddi_fm_fini(9F) function. The bge_fm_fini() function is called in the bge_unattach() function, which is called in both the bge_attach() and bge_detach() functions.

```c
static void
bge_fm_fini(bge_t *bgep)
{
    /* Only unregister FMA capabilities if we registered some */
    if (bgep->fm_capabilities) {
        /* Release any resources allocated by pci_ereport_setup() */
        if (DDI_FM_EREPORT_CAP(bgep->fm_capabilities) ||
            DDI_FM_ERRCB_CAP(bgep->fm_capabilities))
            pci_ereport_teardown(bgep->devinfo);
        /*
        * Un-register error callback if error callback capable
        */
        if (DDI_FM_ERRCB_CAP(bgep->fm_capabilities))
            ddi_fm_handler_unregister(bgep->devinfo);
        /*
        * Unregister from IO Fault Services
        */
        ddi_fm_fini(bgep->devinfo);
    }
}
```

**Getting the Fault Management Capability Bit Mask**

The ddi_fm_capable(9F) function returns the capability bit mask currently set for dip.

```c
void ddi_fm_capable(dev_info_t *dip)
```

**Reporting Errors**

This section provides information about the following topics:

- "Queueing an Error Event" on page 228 discusses how to queue error events.
- "Detecting and Reporting PCI-Related Errors" on page 229 describes how to report PCI-related errors.
- "Reporting Standard I/O Controller Errors" on page 230 describes how to report standard I/O controller errors.
- "Service Impact Function" on page 232 discusses how to report whether an error has impacted the services provided by a device.

**Queueing an Error Event**

The ddi_fm_ereport_post(9F) function causes an erereport event to be queued for delivery to the fault manager daemon, fmd(1M).
void ddi_fm_ereport_post(dev_info_t *dip,
            const char *error_class,
            uint64_t ena,
            int sflag, ...)

The sflag parameter indicates whether the caller is willing to wait for system memory and event channel resources to become available.

The ENA indicates the Error Numeric Association (ENA) for this error report. The ENA might have been initialized and obtained from another error detecting software module such as a bus nexus driver. If the ENA is set to 0, it will be initialized by ddi_fm_ereport_post().

The name-value pair (nvpair) variable argument list contains one or more name, type, value pointer nvpair tuples for non-array data_type_t types or one or more name, type, number of element, value pointer tuples for data_type_t array types. The nvpair tuples make up the ereport event payload required for diagnosis. The end of the argument list is specified by NULL.

The ereport class names and payloads described in “Reporting Standard I/O Controller Errors” on page 230 for I/O controllers are used as appropriate for error_class. Other ereport class names and payloads can be defined, but they must be registered in the Oracle event registry and accompanied by driver specific diagnosis engine software, or the Eversholt fault tree (eft) rules.

void bge_fm_ereport(bge_t *bgep, char *detail)
{
    uint64_t ena;
    char buf[FM_MAX_CLASS];
    (void) snprintf(buf, FM_MAX_CLASS,
                   "%s.%s",
                   bgep->devinfo, detail);
    ena = fm_ena_generate(0, FM_ENA_FMT1);
    if (DDI_FM_EREPORT_CAP(bgep->fm_capabilities)) {
        ddi_fm_ereport_post(bgep->devinfo, buf, ena, DDI_NOSLEEP,
                             FM_VERSION, DATA_TYPE_UINT8, FM_EREPORT_VERS0, NULL);
    }
}

Detecting and Reporting PCI-Related Errors

PCI-related errors, including PCI, PCI-X, and PCI-E, are automatically detected and reported when you use pci_ereport_post(9F).

void pci_ereport_post(dev_info_t *dip, ddi_fm_error_t *derr, uint16_t *xx_status)

Drivers do not need to generate driver-specific ereports for errors that occur in the PCI Local Bus configuration status registers. The pci_ereport_post() function can report data parity errors, master aborts, target aborts, signaled system errors, and much more.

If pci_ereport_post() is to be used by a driver, then pci_ereport_setup(9F) must have been previously called during the driver’s attach(9E) routine, and pci_ereport_teardown(9F) must subsequently be called during the driver’s detach(9E) routine.

The bge code samples below show the bge driver invoking the pci_ereport_post() function from the driver’s error handler.
The I/O fault service error handling callback function

/*ARGSUSED*/
static int
bge_fm_error_cb(dev_info_t *dip, ddi_fm_error_t *err, const void *impl_data)
{
    /*
    * as the driver can always deal with an error
    * in any dma or access handle, we can just return
    * the fme_status value.
    */
    pci_ereport_post(dip, err, NULL);
    return (err->fme_status);
}

Reporting Standard I/O Controller Errors

A standard set of device ereports is defined for commonly seen errors for I/O controllers. These ereports should be generated whenever one of the error symptoms described in this section is detected.

The ereports described in this section are dispatched for diagnosis to the eft diagnosis engine, which uses a common set of standard rules to diagnose them. Any other errors detected by device drivers must be defined as ereport events in the Sun event registry and must be accompanied by device specific diagnosis software or eft rules.

DDI_FM_DEVICE_INVAL_STATE

The driver has detected that the device is in an invalid state.

A driver should post an error when it detects that the data it transmits or receives appear to be invalid. For example, in the bge code, the bge_chip_reset() and bge_receive_ring() routines generate the ereport.io.device.inval_state error when these routines detect invalid data.

/*
 * The SEND INDEX registers should be reset to zero by the
 * global chip reset; if they're not, there'll be trouble
 * later on.
 */
sx0 = bge_reg_get32(bgep, NIC_DIAG_SEND_INDEX_REG(0));
if (sx0 != 0) {
    BGE_REPORT((bgep, "SEND INDEX - device didn't RESET"));
    bge_fm_ereport(bgep, DDI_FM_DEVICE_INVAL_STATE);
    return (DDI_FAILURE);
}
/* ... */
/*
 * Sync (all) the receive ring descriptors
 * before accepting the packets they describe
 */
DMA_SYNC(rrp->desc, DDI_DMA_SYNC_FORKERNEL);
if (*rrp->prod_index_p >= rrp->desc.nslots) {
    bgep->bge_chip_state = BGE_CHIP_ERROR;
bge_fm_ereport(bgep, DDI_FM_DEVICE_INVAL_STATE);
return (NULL);
}

DDI_FM_DEVICE_INTERN_CORR
The device has reported a self-corrected internal error. For example, a correctable ECC error
has been detected by the hardware in an internal buffer within the device. This error flag is
not used in the bge driver.

DDI_FM_DEVICE_INTERN_UNCORR
The device has reported an uncorrectable internal error. For example, an uncorrectable ECC
error has been detected by the hardware in an internal buffer within the device.

This error flag is not used in the bge driver.

DDI_FM_DEVICE_STALL
The driver has detected that data transfer has stalled unexpectedly.

The bge_factotum_stall_check() routine provides an example of stall detection.

dogval = bge_atomic_shl32(&bgep->watchdog, 1);
if (dogval < bge_watchdog_count)
    return (B_FALSE);
BGE_REPORT((bgep,"Tx stall detected, 
            watchdog code 0x%x", dogval));
bge_fm_ereport(bgep, DDI_FM_DEVICE_STALL);
return (B_TRUE);

DDI_FM_DEVICE_NO_RESPONSE
The device is not responding to a driver command.

bge_chip_poll_engine(bge_t *bgep, bge_regno_t regno,
        uint32_t mask, uint32_t val)
{
    uint32_t regval;
    uint32_t n;
    for (n = 200; n; --n) {
        regval = bge_reg_get32(bgep, regno);
        if ((regval & mask) == val)
            return (B_TRUE);
        drv_usecwait(100);
    }
bge_fm_ereport(bgep, DDI_FM_DEVICE_NO_RESPONSE);
return (B_FALSE);
}

DDI_FM_DEVICE_BADINT_LIMIT
The device has raised too many consecutive invalid interrupts.

The bge_intr() routine within the bge driver provides an example of stuck interrupt
detection. The bge_fm_ereport() function is a wrapper for the ddi_fm_ereport_post(9F)
function. See the bge_fm_ereport() example in "Queueing an Error Event" on page 228.
if (bgep->missed_dmas >= bge_dma_miss_limit) {
    /*
    * If this happens multiple times in a row,
    * it means DMA is just not working. Maybe
    * the chip has failed, or maybe there’s a
    * problem on the PCI bus or in the host-PCI
    * bridge (Tomatillo).
    *
    * At all events, we want to stop further
    * interrupts and let the recovery code take
    * over to see whether anything can be done
    * about it ...
    */
    bge_fm_ereport(bgep,
        DDI_FM_DEVICE_BADINT_LIMIT);
    goto chip_stop;
}

Service Impact Function

A fault management capable driver must indicate whether or not an error has impacted the services provided by a device. Following detection of an error and, if necessary, a shutdown of services, the driver should invoke the ddi_fm_service_impact(9F) routine to reflect the current service state of the device instance. The service state can be used by diagnosis and recovery software to help identify or react to the problem.

The ddi_fm_service_impact() routine should be called both when an error has been detected by the driver itself, and when the framework has detected an error and marked an access or DMA handle as faulty.

void ddi_fm_service_impact(dev_info_t *dip, int svc_impact)

The following service impact values (svc_impact) are accepted by ddi_fm_service_impact():

DDI_SERVICE_LOST The service provided by the device is unavailable due to a device fault or software defect.

DDI_SERVICE_DEGRADED The driver is unable to provide normal service, but the driver can provide a partial or degraded level of service. For example, the driver might have to make repeated attempts to perform an operation before it succeeds, or it might be running at less that its configured speed.

DDI_SERVICE_UNAFFECTED The driver has detected an error, but the services provided by the device instance are unaffected.

DDI_SERVICE_RESTORED All of the device's services have been restored.

The call to ddi_fm_service_impact() generates the following ereports on behalf of the driver, based on the service impact argument to the service impact routine:

- ereport.io.service.lost
In the following bge code, the driver determines that it is unable to successfully restart transmitting or receiving packets as the result of an error. The service state of the device transitions to DDI_SERVICE_LOST.

```c
/*
 * All OK, reinitialize hardware and kick off GLD scheduling
 */
mutex_enter(bgep->genlock);
if (bge_restart(bgep, B_TRUE) != DDI_SUCCESS) {
    (void) bge_check_acc_handle(bgep, bgep->cfg_handle);
    (void) bge_check_acc_handle(bgep, bgep->io_handle);
    ddi_fm_service_impact(bgep->devinfo, DDI_SERVICE_LOST);
    mutex_exit(bgep->genlock);
    return (DDI_FAILURE);
}
```

**Note**—The `ddi_fm_service_impact()` function should not be called from the registered callback routine.
Layered Driver Interface (LDI)

The LDI is a set of DDI/DKI that enables a kernel module to access other devices in the system. The LDI also enables you to determine which devices are currently being used by kernel modules.

This chapter covers the following topics:
- "Kernel Interfaces" on page 236
- "User Interfaces" on page 251

LDI Overview

The LDI includes two categories of interfaces:

- **Kernel interfaces.** User applications use system calls to open, read, and write to devices that are managed by a device driver within the kernel. Kernel modules can use the LDI kernel interfaces to open, read, and write to devices that are managed by another device driver within the kernel. For example, a user application might use `read(2)` and a kernel module might use `ldi_read(9F)` to read the same device. See “Kernel Interfaces” on page 236.

- **User interfaces.** The LDI user interfaces can provide information to user processes regarding which devices are currently being used by other devices in the kernel. See “User Interfaces” on page 251.

The following terms are commonly used in discussing the LDI:

- **Target Device.** A target device is a device within the kernel that is managed by a device driver and is being accessed by a device consumer.

- **Device Consumer.** A device consumer is a user process or kernel module that opens and accesses a target device. A device consumer normally performs operations such as `open`, `read`, `write`, or `ioctl` on a target device.
Kernel Interfaces

- **Kernel Device Consumer.** A kernel device consumer is a particular kind of device consumer. A kernel device consumer is a kernel module that accesses a target device. The kernel device consumer usually is not the device driver that manages the target device that is being accessed. Instead, the kernel device consumer accesses the target device indirectly through the device driver that manages the target device.

- **Layered Driver.** A layered driver is a particular kind of kernel device consumer. A layered driver is a kernel driver that does not directly manage any piece of hardware. Instead, a layered driver accesses one or more target devices indirectly through the device drivers that manage those target devices. Volume managers and STREAMS multiplexers are good examples of layered drivers.

Kernel Interfaces

Some LDI kernel interfaces enable the LDI to track and report kernel device usage information. See "Layered Identifiers – Kernel Device Consumers" on page 236.

Other LDI kernel interfaces enable kernel modules to perform access operations such as open, read, and write a target device. These LDI kernel interfaces also enable a kernel device consumer to query property and event information about target devices. See "Layered Driver Handles – Target Devices" on page 237.

"LDI Kernel Interfaces Example" on page 241 shows an example driver that uses many of these LDI interfaces.

Layered Identifiers – Kernel Device Consumers

Layered identifiers enable the LDI to track and report kernel device usage information. A layered identifier (ldi_ident_t) identifies a kernel device consumer. Kernel device consumers must obtain a layered identifier prior to opening a target device using the LDI.

Layered drivers are the only supported types of kernel device consumers. Therefore, a layered driver must obtain a layered identifier that is associated with the device number, the device information node, or the stream of the layered driver. The layered identifier is associated with the layered driver. The layered identifier is not associated with the target device.

You can retrieve the kernel device usage information that is collected by the LDI by using the libdevinfo(3LIB) interfaces, the fuser(1M) command, or the prtconf(1M) command. For example, the prtconf(1M) command can show which target devices a layered driver is accessing or which layered drivers are accessing a particular target device. See "User Interfaces" on page 251 to learn more about how to retrieve device usage information.

The following describes the LDI layered identifier interfaces:

- **ldi_ident_t**
  - Layered identifier. An opaque type.
Layered Driver Handles – Target Devices

Kernel device consumers must use a layered driver handle (ldi_handle_t) to access a target device through LDI interfaces. The ldi_handle_t type is valid only with LDI interfaces. The LDI allocates and returns this handle when the LDI successfully opens a device. A kernel device consumer can then use this handle to access the target device through the LDI interfaces. The LDI deallocates the handle when the LDI closes the device. See “LDI Kernel Interfaces Example” on page 241 for an example.

This section discusses how kernel device consumers can access target devices and retrieve different types of information. See “Opening and Closing Target Devices” on page 237 to learn how kernel device consumers can open and close target devices. See “Accessing Target Devices” on page 238 to learn how kernel device consumers can perform operations such as read, write, strategy, and ioctl on target devices. “Retrieving Target Device Information” on page 239 describes interfaces that retrieve target device information such as device open type and device minor name. “Retrieving Target Device Property Values” on page 239 describes interfaces that retrieve values and address of target device properties. See “Receiving Asynchronous Device Event Notification” on page 240 to learn how kernel device consumers can receive event notification from target devices.

Opening and Closing Target Devices

This section describes the LDI kernel interfaces for opening and closing target devices. The open interfaces take a pointer to a layered driver handle. The open interfaces attempt to open the target device specified by the device number, device ID, or path name. If the open operation is successful, the open interfaces allocate and return a layered driver handle that can be used to access the target device. The close interface closes the target device associated with the specified layered driver handle and then frees the layered driver handle.

ldi_ident_from_dev(9F)  
Allocate and retrieve a layered identifier that is associated with a dev_t device number.

ldi_ident_from_dip(9F)  
Allocate and retrieve a layered identifier that is associated with a dev_info_t device information node.

ldi_ident_from_stream(9F)  
Allocate and retrieve a layered identifier that is associated with a stream.

ldi_ident_release(9F)  
Release a layered identifier that was allocated with ldi_ident_from_dev(9F), ldi_ident_from_dip(9F), or ldi_ident_from_stream(9F).
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ldi_open_by_dev(9F)</code></td>
<td>Open the device specified by the <code>dev_t</code> device number parameter.</td>
</tr>
<tr>
<td><code>ldi_open_by_devid(9F)</code></td>
<td>Open the device specified by the <code>ddi_devid_t</code> device ID parameter. You also must specify the minor node name to open.</td>
</tr>
<tr>
<td><code>ldi_open_by_name(9F)</code></td>
<td>Open a device by path name. The path name is a null-terminated string in the kernel address space. The path name must be an absolute path, beginning with a forward slash character (/).</td>
</tr>
<tr>
<td><code>ldi_close(9F)</code></td>
<td>Close a device that was opened with <code>ldi_open_by_dev(9F)</code>, <code>ldi_open_by_devid(9F)</code>, or <code>ldi_open_by_name(9F)</code>. After <code>ldi_close(9F)</code> returns, the layered driver handle of the device that was closed is no longer valid.</td>
</tr>
</tbody>
</table>

### Accessing Target Devices

This section describes the LDI kernel interfaces for accessing target devices. These interfaces enable a kernel device consumer to perform operations on the target device specified by the layered driver handle. Kernel device consumers can perform operations such as `read`, `write`, strategy, and `ioctl` on the target device.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ldi_handle_t</code></td>
<td>Layered driver handle for target device access. An opaque data structure.</td>
</tr>
<tr>
<td><code>ldi_read(9F)</code></td>
<td>Pass a read request to the device entry point for the target device. This operation is supported for block, character, and STREAMS devices.</td>
</tr>
<tr>
<td><code>ldi_aread(9F)</code></td>
<td>Pass an asynchronous read request to the device entry point for the target device. This operation is supported for block and character devices.</td>
</tr>
<tr>
<td><code>ldi_write(9F)</code></td>
<td>Pass a write request to the device entry point for the target device. This operation is supported for block, character, and STREAMS devices.</td>
</tr>
<tr>
<td><code>ldi_awrite(9F)</code></td>
<td>Pass an asynchronous write request to the device entry point for the target device. This operation is supported for block and character devices.</td>
</tr>
<tr>
<td><code>ldi_strategy(9F)</code></td>
<td>Pass a strategy request to the device entry point for the target device. This operation is supported for block and character devices.</td>
</tr>
<tr>
<td><code>ldi_dump(9F)</code></td>
<td>Pass a dump request to the device entry point for the target device. This operation is supported for block and character devices.</td>
</tr>
<tr>
<td><code>ldi_poll(9F)</code></td>
<td>Pass a poll request to the device entry point for the target device. This operation is supported for block, character, and STREAMS devices.</td>
</tr>
<tr>
<td><code>ldi_ioctl(9F)</code></td>
<td>Pass an <code>ioctl</code> request to the device entry point for the target device. This operation is supported for block, character, and STREAMS devices. The LDI supports STREAMS linking and STREAMS <code>ioctl</code>.</td>
</tr>
</tbody>
</table>
commands. See the “STREAM IOCTLS” section of the ldi_ioctl(9F) man page. See also the ioctl commands in the streamio(7I) man page.

**ldi_devmap(9F)**  
Pass a devmap request to the device entry point for the target device. This operation is supported for block and character devices.

**ldi_getmsg(9F)**  
Get a message block from a stream.

**ldi_putmsg(9F)**  
Put a message block on a stream.

### Retrieving Target Device Information

This section describes LDI interfaces that kernel device consumers can use to retrieve device information about a specified target device. A target device is specified by a layered driver handle. A kernel device consumer can receive information such as device number, device open type, device ID, device minor name, and device size.

**ldi_get_dev(9F)**  
Get the dev_t device number for the target device specified by the layered driver handle.

**ldi_get_otyp(9F)**  
Get the open flag that was used to open the target device specified by the layered driver handle. This flag tells you whether the target device is a character device or a block device.

**ldi_get_devid(9F)**  
Get the ddi_devid_t device ID for the target device specified by the layered driver handle. Use ddi_devid_free(9F) to free the ddi_devid_t when you are finished using the device ID.

**ldi_get_minor_name(9F)**  
Retrieve a buffer that contains the name of the minor node that was opened for the target device. Use kmem_free(9F) to release the buffer when you are finished using the minor node name.

**ldi_get_size(9F)**  
Retrieve the partition size of the target device specified by the layered driver handle.

### Retrieving Target Device Property Values

This section describes LDI interfaces that kernel device consumers can use to retrieve property information about a specified target device. A target device is specified by a layered driver handle. A kernel device consumer can receive values and addresses of properties and determine whether a property exists.

**ldi_prop_exists(9F)**  
Return 1 if the property exists for the target device specified by the layered driver handle. Return 0 if the property does not exist for the specified target device.
Receiving Asynchronous Device Event Notification

The LDI enables kernel device consumers to register for event notification and to receive event notification from target devices. A kernel device consumer can register an event handler that will be called when the event occurs. The kernel device consumer must open a device and receive a layered driver handle before the kernel device consumer can register for event notification with the LDI event notification interfaces.

The LDI event notification interfaces enable a kernel device consumer to specify an event name and to retrieve an associated kernel event cookie. The kernel device consumer can then pass the layered driver handle (ldi_handle_t), the cookie (ddi_eventcookie_t), and the event handler to ldi_add_event_handler(9F) to register for event notification. When registration completes successfully, the kernel device consumer receives a unique LDI event handler identifier (ldi_callback_id_t). The LDI event handler identifier is an opaque type that can be used only with the LDI event notification interfaces.
The LDI provides a framework to register for events generated by other devices. The LDI itself does not define any event types or provide interfaces for generating events.

The following describes the LDI asynchronous event notification interfaces:

- `ldi_callback_id_t` Event handler identifier. An opaque type.
- `ldi_get_eventcookie(9F)` Retrieve an event service cookie for the target device specified by the layered driver handle.
- `ldi_add_event_handler(9F)` Add the callback handler specified by the `ldi_callback_id_t` registration identifier. The callback handler is invoked when the event specified by the `ddi_eventcookie_t` cookie occurs.
- `ldi_remove_event_handler(9F)` Remove the callback handler specified by the `ldi_callback_id_t` registration identifier.

**LDI Kernel Interfaces Example**

This section shows an example kernel device consumer that uses some of the LDI calls discussed in the preceding sections in this chapter. This section discusses the following aspects of this example module:

- “Device Configuration File” on page 241
- “Driver Source File” on page 242
- “Test the Layered Driver” on page 250

This example kernel device consumer is named `lyr`. The `lyr` module is a layered driver that uses LDI calls to send data to a target device. In its `open(9E)` entry point, the `lyr` driver opens the device that is specified by the `lyr_targ` property in the `lyr.conf` configuration file. In its `write(9E)` entry point, the `lyr` driver writes all of its incoming data to the device specified by the `lyr_targ` property.

**Device Configuration File**

In the configuration file shown below, the target device that the `lyr` driver is writing to is the console.

```
EXAMPLE 14–1  Configuration File
#
# Use is subject to license terms.
#
#pragma ident  "%%M\%  %I\%  %E\% SMI"

name="lyr" parent="pseudo" instance=1;
lyr_targ="/dev/console";
```
Driver Source File

In the driver source file shown below, the `lyr_state_t` structure holds the soft state for the `lyr` driver. The soft state includes the layered driver handle (`lh`) for the `lyr_targ` device and the layered identifier (`li`) for the `lyr` device. For more information on soft state, see "Retrieving Driver Soft State Information" on page 523.

In the `lyr_open()` entry point, `ddi_prop_lookup_string(9F)` retrieves from the `lyr_targ` property the name of the target device for the `lyr` device to open. The `ldi_ident_from_dev(9F)` function gets an LDI layered identifier for the `lyr` device. The `ldi_open_by_name(9F)` function opens the `lyr_targ` device and gets a layered driver handle for the `lyr_targ` device.

Note that if any failure occurs in `lyr_open()`, the `ldi_close(9F)`, `ldi_ident_release(9F)`, and `ddi_prop_free(9F)` calls undo everything that was done. The `ldi_close(9F)` function closes the `lyr_targ` device. The `ldi_ident_release(9F)` function releases the `lyr` layered identifier. The `ddi_prop_free(9F)` function frees resources allocated when the `lyr_targ` device name was retrieved. If no failure occurs, the `ldi_close(9F)` and `ldi_ident_release(9F)` functions are called in the `lyr_close()` entry point.

In the last line of the driver module, the `ldi_write(9F)` function is called. The `ldi_write(9F)` function takes the data written to the `lyr` device in the `lyr_write()` entry point and writes that data to the `lyr_targ` device. The `ldi_write(9F)` function uses the layered driver handle for the `lyr_targ` device to write the data to the `lyr_targ` device.

```c
#include <sys/types.h>
#include <sys/file.h>
#include <sys/errno.h>
#include <sys/open.h>
#include <sys/cmn_err.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/sunldi.h>

typedef struct lyr_state {
    ldi_handle_t lh;
    ldi_ident_t li;
    dev_info_t *dip;
    minor_t minor;
    int flags;
    kmutex_t lock;
} lyr_state_t;

#define LYR_OPENED 0x1 /* lh is valid */
#define LYR_IDENTED 0x2 /* li is valid */
```

EXAMPLE 14-2  Driver Source File
EXAMPLE 14–2  Driver Source File  (Continued)

static int lyr_info(dev_info_t *, ddi_info_cmd_t, void *, void **);
static int lyr_attach(dev_info_t *, ddi_attach_cmd_t);
static int lyr_detach(dev_info_t *, ddi_detach_cmd_t);
static int lyr_open(dev_t *, int, int, cred_t *);
static int lyr_close(dev_t, int, int, cred_t *);
static int lyr_write(dev_t, struct uio *, cred_t *);

static void *lyr_statep;

static struct cb_ops lyr_cb_ops = {
    lyr_open, /* open */
    lyr_close, /* close */
    nodev, /* strategy */
    nodev, /* print */
    nodev, /* dump */
    nodev, /* read */
    lyr_write, /* write */
    nodev, /* ioctl */
    nodev, /* devmap */
    nodev, /* segmap */
    nochpoll, /* poll */
    ddi_prop_op, /* prop_op */
    NULL, /* streamtab */
    D NEW | D MP, /* cb flag */
    CB_REV, /* cb_rev */
    nodev, /* aread */
    nodev /* awrite */
};

static struct dev_ops lyr_dev_ops = {
    DEVO_REV, /* devo_rev, */
    0, /* refcnt */
    lyr_info, /* getinfo */
    nulldev, /* identify */
    nulldev, /* probe */
    lyr_attach, /* attach */
    lyr_detach, /* detach */
    nodev, /* reset */
    &lyr_cb_ops, /* cb_ops */
    NULL, /* bus_ops */
    NULL, /* power */
    ddi_quiesce_not_needed, /* quiesce */
};

static struct modldriv modldriv = {
    &mod_driverops,
    "LDI example driver",
    &lyr_dev_ops
};

static struct modlinkage modlinkage = {
    MODREV_1,
    &modldriv,
EXAMPLE 14–2  Driver Source File  (Continued)

    NULL
    
    int
    _init(void)
    {
        int rv;

        if ((rv = ddi_soft_state_init(&lyr_statep, sizeof (lyr_state_t),
            0)) != 0) {
            cmn_err(CE_WARN, "lyr_init: soft state init failed\n";
            return (rv);
        }
        if ((rv = mod_install(&modlinkage)) != 0) {
            cmn_err(CE_WARN, "lyr_init: mod_install failed\n";
            goto FAIL;
        }
        return (rv);
        /*NOTEREACHED*/
    FAIL:
        ddi_soft_state_fini(&lyr_statep);
        return (rv);
    }

    int
    _info(struct modinfo *modinfop)
    {
        return (mod_info(&modlinkage, modinfop));
    }

    int
    _fini(void)
    {
        int rv;

        if ((rv = mod_remove(&modlinkage)) != 0) {
            return(rv);
        }
        ddi_soft_state_fini(&lyr_statep);
        return (rv);
    }

    /* 1:1 mapping between minor number and instance */
    static int
    lyr_info(dev_info_t *dip, ddi_info_cmd_t infocmd, void *arg, void **result)
    {
        int inst;
        minor_t minor;
        lyr_state_t *statep;
        char *myname = "lyr_info";

        minor = getminor((dev_t)arg);
        inst = minor;
        switch (infocmd) {
            }
case DDI_INFO_DEVT2DEVINFO:
    statep = ddi_get_soft_state(lyr_statep, inst);
    if (statep == NULL) {
        cmn_err(CE_WARN, "%s: get soft state 
             "failed on inst %d\n", myname, inst);
        return (DDI_FAILURE);
    }
    *result = (void *)statep->dip;
    break;

case DDI_INFO_DEVT2INSTANCE:
    *result = (void *)inst;
    break;

default:
    break;
}

return (DDI_SUCCESS);

static int
lyr_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    int inst;
    lyr_state_t *statep;
    char *myname = "lyr_attach";

    switch (cmd) {
    case DDI_ATTACH:
        inst = ddi_get_instance(dip);
        if (ddi_soft_state_zalloc(lyr_statep, inst) != DDI_SUCCESS) {
            cmn_err(CE_WARN, "%s: ddi_soft_state_zallac failed 
                     "on inst %d\n", myname, inst);
            goto FAIL;
        }
        statep = (lyr_state_t *)ddi_get_soft_state(lyr_statep, inst);
        if (statep == NULL) {
            cmn_err(CE_WARN, "%s: ddi_get_soft_state failed on 
                     "inst %d\n", myname, inst);
            goto FAIL;
        }
        statep->dip = dip;
        statep->minor = inst;
        if (ddi_create_minor_node(dip, "node", S_IFCHR, statep->minor, 
            DDI_PSEUDO, 0) != DDI_SUCCESS) {
            cmn_err(CE_WARN, "%s: ddi_create_minor_node failed on 
                     "inst %d\n", myname, inst);
            goto FAIL;
        }
        mutex_init(&statep->lock, NULL, MUTEX_DRIVER, NULL);
        return (DDI_SUCCESS);
    case DDI_RESUME:
    case DDI_PM_RESUME:
        default:
            break;
    }
return (DDI_FAILURE);
/*NOTREACHED*/
FAIL:
ddi_soft_state_free(lyr_statep, inst);
ddi_remove_minor_node(dip, NULL);
return (DDI_FAILURE);
}

static int
lyr_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
    int inst;
    lyr_state_t *statep;
    char *myname = "lyr_detach";

    inst = ddi_get_instance(dip);
    statep = ddi_get_soft_state(lyr_statep, inst);
    if (statep == NULL) {
        cmn_err(CE_WARN, "%s: get soft state failed on "
                "inst %d\n", myname, inst);
        return (DDI_FAILURE);
    }
    if (statep->dip != dip) {
        cmn_err(CE_WARN, "%s: soft state does not match devinfo "
                "on inst %d\n", myname, inst);
        return (DDI_FAILURE);
    }

    switch (cmd) {
    case DDI_DETACH:
        mutex_destroy(&statep->lock);
        ddi_soft_state_free(lyr_statep, inst);
        ddi_remove_minor_node(dip, NULL);
        return (DDI_SUCCESS);
    case DDI_SUSPEND:
    case DDI_PM_SUSPEND:
        break;
    default:
        break;
    }
    return (DDI_FAILURE);
}

/*
 * on this driver's open, we open the target specified by a property and store
 * the layered handle and ident in our soft state. a good target would be
 * "/dev/console" or more interestingly, a pseudo terminal as specified by the
 * tty command
 */
/*ARGSUSED*/
static int
lyr_open(dev_t *devtp, int oflag, int otyp, cred_t *credp)
{
    int rv, inst = getminor(*devtp);
    lyr_state_t *statep;
    char *myname = "lyr_open";
    dev_info_t *dip;
char *lyr_targ = NULL;

statep = (lyr_state_t *)ddi_get_soft_state(lyr_statep, inst);
if (statep == NULL) {
    cmn_err(CE_WARN, "%s: ddi_get_soft_state failed on "
    "inst %d\n", myname, inst);
    return (EIO);
}

dip = statep->dip;

/*
 * our target device to open should be specified by the "lyr_targ"
 * string property, which should be set in this driver’s .conf file
 */
if (ddi_prop_lookup_string(DDI_DEV_T_ANY, dip, DDI_PROP_NOTPROM,
    "lyr_targ", &lyr_targ) != DDI_PROP_SUCCESS) {
    cmn_err(CE_WARN, "%s: ddi_prop_lookup_string failed on "
    "inst %d\n", myname, inst);
    return (EIO);
}

/*
 * since we only have one pair of lh’s and li’s available, we don’t
 * allow multiple on the same instance
 */
mutex_enter(&statep->lock);
if (statep->flags & (LYR_OPENED | LYR_IDENTED)) {
    cmn_err(CE_WARN, "%s: multiple layered opens or idents "
    "from inst %d not allowed\n", myname, inst);
    mutex_exit(&statep->lock);
    ddi_prop_free(lyr_targ);
    return (EIO);
}

/*
 * allow multiple on the same instance
 */
mutex_enter(&statep->lock);
if (statep->flags & (LYR_OPENED | LYR_IDENTED)) {
    cmn_err(CE_WARN, "%s: ldi_ident_from_dev failed on inst %d\n",
    myname, inst);
    goto FAIL;
}

rv = ldi_ident_from_dev(*devtp, &statep->li);
if (rv != 0) {
    cmn_err(CE_WARN, "%s: ldi_ident_from_dev failed on inst %d\n",
    myname, inst);
    goto FAIL;
}

statep->flags |= LYR_IDENTED;
rv = ldi_open_by_name(lyr_targ, FREAD | FWRITE, credp, &statep->lh,
    statep->li);
if (rv != 0) {
    cmn_err(CE_WARN, "%s: ldi_open_by_name failed on inst %d\n",
    myname, inst);
    goto FAIL;
}

statep->flags |= LYR_OPENED;

FAIL:

/* cleanup on error */
if (rv != 0) {
    if (statep->flags & LYR_OPENED)
        (void)ldi_close(statep->lh, FREAD | FWRITE, credp);
}
if (statep->flags & LYR_IDENTED)
    ldi_ident_release(statep->li);
statep->flags &= ~(LYR_OPENED | LYR_IDENTED);
}
mutex_exit(&statep->lock);
if (lyr_targ != NULL)
    ddi_prop_free(lyr_targ);
return (rv);

/* on this driver’s close, we close the target indicated by the lh member
 * in our soft state and release the ident, li as well. in fact, we MUST do
 * both of these at all times even if close yields an error because the
 * device framework effectively closes the device, releasing all data
 * associated with it and simply returning whatever value the target’s
 * close(9E) returned. therefore, we must as well.
 */
/*ARGSUSED*/
static int
lyr_close(dev_t devt, int oflag, int otyp, cred_t *credp)
{
    int rv, inst = getminor(devt);
    lyr_state_t *statep;
    char *myname = "lyr_close";
    statep = (lyr_state_t *)ddi_get_soft_state(lyr_statep, inst);
    if (statep == NULL) {
        cmn_err(CE_WARN, "%s: ddi_get_soft_state failed on "
            "inst %d
            ", myname, inst);
        return (EIO);
    }
    mutex_enter(&statep->lock);
    rv = ldi_close(statep->lh, FREAD | FWRITE, credp);
    if (rv == 0) {
        cmn_err(CE_WARN, "%s: closed target successfully on "
            "inst %d\n", myname, inst);
    }
    ldi_ident_release(statep->li);
    if (rv == 0) {
        cmn_err(CE_CONT, "%s: closed target successfully on "
            "inst %d\n", myname, inst);
    }
    statep->flags &= ~(LYR_OPENED | LYR_IDENTED);
    mutex_exit(&statep->lock);
    return (rv);
}

/* echo the data we receive to the target */
/*ARGSUSED*/
static int
lyr_write(dev_t devt, struct uio *uiop, cred_t *credp)
{
    int rv, inst = getminor(devt);
    lyr_state_t *statep;
    char *myname = "lyr_write";
statep = (lyr_state_t *)ddi_get_soft_state(lyr_statep, inst);
if (statep == NULL) {
    cmn_err(CE_WARN, "%s: ddi_get_soft_state failed on "
            "inst %d\n", myname, inst);
    return (EIO);
} return (ldi_write(statep->lh, uiop, credp));

▼ How to Build and Load the Layered Driver

1 Compile the driver.
   Use the -D_KERNEL option to indicate that this is a kernel module.
   ▪ If you are compiling for a SPARC architecture, use the -xarch=v9 option:
     % cc -c -D_KERNEL -xarch=v9 lyr.c
   ▪ If you are compiling for a 32-bit x86 architecture, use the following command:
     % cc -c -D_KERNEL lyr.c

2 Link the driver.
   % ld -r -o lyr lyr.o

3 Install the configuration file.
   As user root, copy the configuration file to the kernel driver area of the machine:
   # cp lyr.conf /usr/kernel/drv

4 Install the driver binary.
   ▪ As user root, copy the driver binary to the sparcv9 driver area on a SPARC architecture:
     # cp lyr /usr/kernel/drv/sparcv9
   ▪ As user root, copy the driver binary to the drv driver area on a 32-bit x86 architecture:
     # cp lyr /usr/kernel/drv

5 Load the driver.
   As user root, use the add_drv(1M) command to load the driver.
   # add_drv lyr
   List the pseudo devices to confirm that the lyr device now exists:

   # ls /devices/pseudo | grep lyr
   lyr@0
   lyr@0:node
Test the Layered Driver

To test the \texttt{lyr} driver, write a message to the \texttt{lyr} device and verify that the message displays on the \texttt{lyr\_targ} device.

\textbf{EXAMPLE 14–3} \hspace{1em} Write a Short Message to the Layered Device

In this example, the \texttt{lyr\_targ} device is the console of the system where the \texttt{lyr} device is installed.

If the display you are viewing is also the display for the console device of the system where the \texttt{lyr} device is installed, note that writing to the console will corrupt your display. The console messages will appear outside your window system. You will need to redraw or refresh your display after testing the \texttt{lyr} driver.

If the display you are viewing is not the display for the console device of the system where the \texttt{lyr} device is installed, log into or otherwise gain a view of the display of the target console device.

The following command writes a very brief message to the \texttt{lyr} device:

\begin{verbatim}
# echo "\n\n\t===> Hello World!! <===
" > /devices/pseudo/lyr@1:node
\end{verbatim}

You should see the following messages displayed on the target console:

\begin{verbatim}
console login:

===> Hello World!! <===
\end{verbatim}

The messages from \texttt{lyr\_open()} and \texttt{lyr\_close()} come from the \texttt{cmn\_err(9F)} calls in the \texttt{lyr\_open()} and \texttt{lyr\_close()} entry points.

\textbf{EXAMPLE 14–4} \hspace{1em} Write a Longer Message to the Layered Device

The following command writes a longer message to the \texttt{lyr} device:

\begin{verbatim}
# cat lyr.conf > /devices/pseudo/lyr@1:node
\end{verbatim}

You should see the following messages displayed on the target console:

\begin{verbatim}
lyr:
lyr\_open: opened target \'/dev/console' successfully on inst 1
lyr:
lyr\_close: closed target successfully on inst 1

# Use is subject to license terms.
# \#pragma ident "%Z%%M% %I% %E% SMI"
\end{verbatim}
EXAMPLE 14–4  Write a Longer Message to the Layered Device  (Continued)

name="lyr" parent="pseudo" instance=1;
lyr_targ="/dev/console";
lyr:
lyr_close: closed target successfully on inst 1

EXAMPLE 14–5  Change the Target Device

To change the target device, edit /usr/kernel/drv/lyr.conf and change the value of the
lyr_targ property to be a path to a different target device. For example, the target device could
be the output of a tty command in a local terminal. An example of such a device path is
/dev/pts/4.

Make sure the lyr device is not in use before you update the driver to use the new target device.

# modinfo -c | grep lyr
174 3 lyr UNLOADED/UNINSTALLED

Use the update_drv(1M) command to reload the lyr.conf configuration file:

# update_drv lyr

Write a message to the lyr device again and verify that the message displays on the new
lyr_targ device.

User Interfaces

The LDI includes user-level library and command interfaces to report device layering and usage
information. "Device Information Library Interfaces" on page 251 discusses the
libdevinfo(3LIB) interfaces for reporting device layering information. "Print System
Configuration Command Interfaces" on page 253 discusses the prtconf(1M) interfaces for
reporting kernel device usage information. "Device User Command Interfaces" on page 256
discusses the fuser(1M) interfaces for reporting device consumer information.

Device Information Library Interfaces

The LDI includes libdevinfo(3LIB) interfaces that report a snapshot of device layering
information. Device layering occurs when one device in the system is a consumer of another
device in the system. Device layering information is reported only if both the consumer and the
target are bound to a device node that is contained within the snapshot.
Device layering information is reported by the `libdevinfo(3LIB)` interfaces as a directed graph. An *lnode* is an abstraction that represents a vertex in the graph and is bound to a device node. You can use `libdevinfo(3LIB)` interfaces to access properties of an *lnode*, such as the name and device number of the node.

The edges in the graph are represented by a link. A link has a source *lnode* that represents the device consumer. A link also has a target *lnode* that represents the target device.

The following describes the `libdevinfo(3LIB)` device layering information interfaces:

- **DINFOLYR**
  - Snapshot flag that enables you to capture device layering information.

- **di_link_t**
  - A directed link between two endpoints. Each endpoint is a `di_lnode_t`. An opaque structure.

- **di_lnode_t**
  - The endpoint of a link. An opaque structure. A `di_lnode_t` is bound to a `di_node_t`.

- **di_node_t**
  - Represents a device node. An opaque structure. A `di_node_t` is not necessarily bound to a `di_lnode_t`.

- **di_walk_link(3DEVINFO)**
  - Walk all links in the snapshot.

- **di_walk_lnode(3DEVINFO)**
  - Walk all *lnodes* in the snapshot.

- **di_link_next_by_node(3DEVINFO)**
  - Get a handle to the next link where the specified `di_node_t` node is either the source or the target.

- **di_link_next_by_lnode(3DEVINFO)**
  - Get a handle to the next link where the specified `di_lnode_t` *lnode* is either the source or the target.

- **di_link_to_lnode(3DEVINFO)**
  - Get the *lnode* that corresponds to the specified endpoint of a `di_link_t` link.

- **di_link_spectype(3DEVINFO)**
  - Get the link spectype. The spectype indicates how the target device is being accessed. The target device is represented by the target *lnode*.

- **di_lnode_next(3DEVINFO)**
  - Get a handle to the next occurrence of the specified `di_lnode_t` *lnode* associated with the specified `di_node_t` device node.

- **di_lnode_name(3DEVINFO)**
  - Get the name that is associated with the specified *lnode*.

- **di_lnode_devinfo(3DEVINFO)**
  - Get a handle to the device node that is associated with the specified *lnode*. 
Get the device number of the device node that is associated with the specified Inode.

The device layering information returned by the LDI can be quite complex. Therefore, the LDI provides interfaces to help you traverse the device tree and the device usage graph. These interfaces enable the consumer of a device tree snapshot to associate custom data pointers with different structures within the snapshot. For example, as an application traverses Inodes, the application can update the custom pointer associated with each Inode to mark which Inodes already have been seen.

The following describes the `libdevinfo(3LIB)` node and link marking interfaces:

- **di_lnode_private_set(3DEVINFO)**: Associate the specified data with the specified Inode. This association enables you to traverse Inodes in the snapshot.

- **di_lnode_private_get(3DEVINFO)**: Retrieve a pointer to data that was associated with an Inode through a call to `di_lnode_private_set(3DEVINFO)`.

- **di_link_private_set(3DEVINFO)**: Associate the specified data with the specified link. This association enables you to traverse links in the snapshot.

- **di_link_private_get(3DEVINFO)**: Retrieve a pointer to data that was associated with a link through a call to `di_link_private_set(3DEVINFO)`.

### Print System Configuration Command Interfaces

The `prtconf(1M)` command is enhanced to display kernel device usage information. The default `prtconf(1M)` output is not changed. Device usage information is displayed when you specify the verbose option (`-v`) with the `prtconf(1M)` command. Usage information about a particular device is displayed when you specify a path to that device on the `prtconf(1M)` command line.

- **`prtconf -v`**: Display device minor node and device usage information. Show kernel consumers and the minor nodes each kernel consumer currently has open.

- **`prtconf path`**: Display device usage information for the device specified by `path`.

- **`prtconf -a path`**: Display device usage information for the device specified by `path` and all device nodes that are ancestors of `path`.

- **`prtconf -c path`**: Display device usage information for the device specified by `path` and all device nodes that are children of `path`. 
EXAMPLE 14–6  Device Usage Information

When you want usage information about a particular device, the value of the path parameter can be any valid device path.

% prtconf /dev/cfg/c0
SUNW,isptwo, instance #0

EXAMPLE 14–7  Ancestor Node Usage Information

To display usage information about a particular device and all device nodes that are ancestors of that particular device, specify the -a flag with the prtconf(1M) command. Ancestors include all nodes up to the root of the device tree. If you specify the -a flag with the prtconf(1M) command, then you must also specify a device path name.

% prtconf -a /dev/cfg/c0
SUNW,Sun-Fire  
 ssm, instance #0
  pci, instance #0
    SUNW,isptwo, instance #0

EXAMPLE 14–8  Child Node Usage Information

To display usage information about a particular device and all device nodes that are children of that particular device, specify the -c flag with the prtconf(1M) command. If you specify the -c flag with the prtconf(1M) command, then you must also specify a device path name.

% prtconf -c /dev/cfg/c0
SUNW,isptwo, instance #0
  sd (driver not attached)
  st (driver not attached)
  sd, instance #1
  sd, instance #0
  sd, instance #6
  st, instance #1 (driver not attached)
  st, instance #0 (driver not attached)
  st, instance #2 (driver not attached)
  st, instance #3 (driver not attached)
  st, instance #4 (driver not attached)
  st, instance #5 (driver not attached)
  st, instance #6 (driver not attached)
  ses, instance #0 (driver not attached)
  ...

EXAMPLE 14–9  Layering and Device Minor Node Information – Keyboard

To display device layering and device minor node information about a particular device, specify the -v flag with the prtconf(1M) command.

% prtconf -v /dev/kbd
conskbd, instance #0
EXAMPLE 14–9  Layering and Device Minor Node Information – Keyboard  (Continued)

System properties:
...
Device Layered Over:
  mod=kb8042  dev=(101,0)
  dev_path=/isa/i8042@1,60/keyboard@
Device Minor Nodes:
  dev=(103,0)
    dev_path=/pseudo/conskbd@0:kbd
    spectype=chr  type=minor
    dev_link=/dev/kbd
  dev=(103,1)
    dev_path=/pseudo/conskbd@0:conskbd
    spectype=chr  type=internal
Device Minor Layered Under:
  mod=wc  accesstype=chr
  dev_path=/pseudo/wc@

This example shows that the /dev/kbd device is layered on top of the hardware keyboard device (/isa/i8042@1,60/keyboard@). This example also shows that the /dev/kbd device has two device minor nodes. The first minor node has a /dev link that can be used to access the node. The second minor node is an internal node that is not accessible through the file system. The second minor node has been opened by the wc driver, which is the workstation console. Compare the output from this example to the output from Example 14–12.

EXAMPLE 14–10  Layering and Device Minor Node Information – Network Device

This example shows which devices are using the currently plumbed network device.

% prtconf -v /dev/iprb0
pci1028,145, instance #0
  Hardware properties:
  ...
  Interrupt Specifications:
  ...
Device Minor Nodes:
  dev=(27,1)
    dev_path=/pci@0,0/pci8086,244e@1e/pci1028,145@:iprb0
    spectype=chr  type=minor
    alias=/dev/iprb0
  dev=(27,4098)
    dev_path=<clone>
Device Minor Layered Under:
  mod=udp6  accesstype=chr
  dev_path=/pseudo/udp6@
  dev=(27,4097)
    dev_path=<clone>
Device Minor Layered Under:
  mod=udp  accesstype=chr
  dev_path=/pseudo/udp@
  dev=(27,4096)
    dev_path=<clone>
Device Minor Layered Under:
This example shows that the iprb0 device has been linked under udp and udp6. Notice that no paths are shown to the minor nodes that udp and udp6 are using. No paths are shown in this case because the minor nodes were created through clone opens of the iprb driver, and therefore there are no file system paths by which these nodes can be accessed. Compare the output from this example to the output from Example 14–11.

### Device User Command Interfaces

The `fuser(1M)` command is enhanced to display device usage information. The `fuser(1M)` command displays device usage information only if `path` represents a device minor node. The `-d` flag is valid for the `fuser(1M)` command only if you specify a `path` that represents a device minor node.

- **fuser path**  
  Display information about application device consumers and kernel device consumers if `path` represents a device minor node.

- **fuser -d path**  
  Display all users of the underlying device that is associated with the device minor node represented by `path`.

Kernel device consumers are reported in one of the following four formats. Kernel device consumers always are surrounded by square brackets ([ ]).

- `[kernel_module_name]`
- `[kernel_module_name, dev_path=path]`
- `[kernel_module_name, dev=(major, minor)]`
- `[kernel_module_name, dev=(major, minor), dev_path=path]`

When the `fuser(1M)` command displays file or device users, the output consists of a process ID on `stdout` followed by a character on `stderr`. The character on `stderr` describes how the file or device is being used. All kernel consumer information is displayed to `stderr`. No kernel consumer information is displayed to `stdout`.

If you do not use the `-d` flag, then the `fuser(1M)` command reports consumers of only the device minor node that is specified by `path`. If you use the `-d` flag, then the `fuser(1M)` command reports consumers of the device node that underlies the minor node specified by `path`. The following example illustrates the difference in report output in these two cases.

**EXAMPLE 14–11**  
**Consumers of Underlying Device Nodes**

Most network devices clone their minor node when the device is opened. If you request device usage information for the clone minor node, the usage information might show that no process is using the device. If instead you request device usage information for the underlying device
EXAMPLE 14–11 Consumers of Underlying Device Nodes  

(Continued)

node, the usage information might show that a process is using the device. In this example, no device consumers are reported when only a device path is passed to the fuser(1M) command. When the -d flag is used, the output shows that the device is being accessed by udp and udp6.

% fuser /dev/iprb0
/dev/iprb0:
% fuser -d /dev/iprb0
/dev/iprb0: [udp, dev_path=/pseudo/udp@0] [udp6, dev_path=/pseudo/udp6@0]

Compare the output from this example to the output from Example 14–10.

EXAMPLE 14–12 Consumer of the Keyboard Device

In this example, a kernel consumer is accessing /dev/kbd. The kernel consumer that is accessing the /dev/kbd device is the workstation console driver.

% fuser -d /dev/kbd
/dev/kbd: [genunix] [wc, dev_path=/pseudo/wc@0]

Compare the output from this example to the output from Example 14–9.
PART II

Designing Specific Kinds of Device Drivers

The second part of the book provides design information that is specific to the type of driver:

- Chapter 16, “Drivers for Block Devices,” describes drivers for a block-oriented devices.
- Chapter 17, “SCSI Target Drivers,” outlines the Sun Common SCSI Architecture (SCSA) and the requirements for SCSI target drivers.
- Chapter 18, “SCSI Host Bus Adapter Drivers,” explains how to apply SCSA to SCSI Host Bus Adapter (HBA) drivers.
- Chapter 19, “Drivers for Network Devices,” describes the Generic LAN driver (GLD). The GLDv3 framework is a function calls-based interface of MAC plugins and MAC driver service routines and structures.
- Chapter 20, “USB Drivers,” describes how to write a client USB device driver using the USBA 2.0 framework.
- Chapter 21, “SR-IOV Drivers,” describes the SR-IOV device driver and the interfaces available to write an SR-IOV driver.
A character device does not have physically addressable storage media, such as tape drives or serial ports, where I/O is normally performed in a byte stream. This chapter describes the structure of a character device driver, focusing in particular on entry points for character drivers. In addition, this chapter describes the use of physio(9F) and aphsio(9F) in the context of synchronous and asynchronous I/O transfers.

This chapter provides information on the following subjects:
- “Overview of the Character Driver Structure” on page 261
- “Character Device Autoconfiguration” on page 263
- “Device Access (Character Drivers)” on page 264
- “I/O Request Handling” on page 266
- “Mapping Device Memory” on page 275
- “Multiplexing I/O on File Descriptors” on page 276
- “Miscellaneous I/O Control” on page 278
- “32-bit and 64-bit Data Structure Macros” on page 283

Overview of the Character Driver Structure

Figure 15–1 shows data structures and routines that define the structure of a character device driver. Device drivers typically include the following elements:

- Device-loadable driver section
- Device configuration section
- Character driver entry points

The shaded device access section in the following figure illustrates character driver entry points.
Associated with each device driver is a `dev_ops(9S)` structure, which in turn refers to a `cb_ops(9S)` structure. These structures contain pointers to the driver entry points:

- `open(9E)`
- `close(9E)`
- `read(9E)`
- `write(9E)`
- `ioctl(9E)`
- `chpoll(9E)`
- `aread(9E)`
- `awrite(9E)`
- `mmap(9E)`
- `devmap(9E)`
- `segmap(9E)`
- `prop_op(9E)`

**Note** – Some of these entry points can be replaced with `nodev(9F)` or `nulldev(9F)` as appropriate.
Character Device Autoconfiguration

The attach(9E) routine should perform the common initialization tasks that all devices require, such as:

- Allocating per-instance state structures
- Registering device interrupts
- Mapping the device’s registers
- Initializing mutex variables and condition variables
- Creating power-manageable components
- Creating minor nodes

See “attach(9E)EntryPoint” on page 104 for code examples of these tasks.

Character device drivers create minor nodes of type S_IFCHR. A minor node of S_IFCHR causes a character special file that represents the node to eventually appear in the /devices hierarchy.

The following example shows a typical attach(9E) routine for character drivers. Properties that are associated with the device are commonly declared in an attach() routine. This example uses a predefined Size property. Size is the equivalent of the Nblocks property for getting the size of partition in a block device. If, for example, you are doing character I/O on a disk device, you might use Size to get the size of a partition. Since Size is a 64-bit property, you must use a 64-bit property interface. In this case, you use ddi_prop_update_int64(9F). See “Device Properties” on page 77 for more information about properties.

EXAMPLE 15-1 Character Driver attach() Routine

```c
static int
xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    int instance = ddi_get_instance(dip);
    switch (cmd) {
    case DDI_ATTACH:
        /*
         * Allocate a state structure and initialize it.
         * Map the device’s registers.
         * Add the device driver’s interrupt handler(s).
         * Initialize any mutexes and condition variables.
         * Create power manageable components.
         *
         * Create the device’s minor node. Note that the node_type
         * argument is set to DDI_NT_TAPE.
         */
        if (ddi_create_minor_node(dip, minor_name, S_IFCHR,
                                  instance, DDI_NT_TAPE, 0) == DDI_FAILURE)
            {
            /* Free resources allocated so far. */
            /* Remove any previously allocated minor nodes. */
            ddi_remove_minor_node(dip, NULL);
            return (DDI_FAILURE);
        }
        /*
        */
    }
}
```
EXAMPLE 15–1  Character Driver attach() Routine  (Continued)

* Create driver properties like "Size." Use "Size"
* instead of "size" to ensure the property works
* for large bytecounts.
*/
   xsp->Size = size_of_device_in_bytes;
   maj_number = ddi_driver_major(dip);
   if (ddi_prop_update_int64(makedevice(maj_number, instance),
       dip, "Size", xsp->Size) != DDI_PROP_SUCCESS) {
       cmn_err(CE_CONT, "%s: cannot create Size property\n",
               ddi_get_name(dip));
       /* Free resources allocated so far. */
       return (DDI_FAILURE);
   }
   /* ... */
   return (DDI_SUCCESS);
}

Device Access (Character Drivers)

Access to a device by one or more application programs is controlled through the open(9E) and
close(9E) entry points. An open(2) system call to a special file that represents a character device
always causes a call to the open(9E) routine for the driver. For a particular minor device,
open(9E) can be called many times. The close(9E) routine is called only when the final
reference to a device is removed. If the device is accessed through file descriptors, the final call to
close(9E) can occur as a result of a close(2) or exit(2) system call. If the device is accessed
through memory mapping, the final call to close(9E) can occur as a result of a munmap(2)
system call.

open () Entry Point (Character Drivers)

The primary function of open() is to verify that the open request is allowed. The syntax for
open(9E) is as follows:

int xxopen(dev_t *devp, int flag, int otyp, cred_t *credp);

where:

devp Pointer to a device number. The open() routine is passed a pointer so that the driver
can change the minor number. With this pointer, drivers can dynamically create
minor instances of the device. An example would be a pseudo terminal driver that
creates a new pseudo-terminal whenever the driver is opened. A driver that
dynamically chooses the minor number normally creates only one minor device node in attach(9E) with ddi_create_minor_node(9F), then changes the minor number component of *devp using makedevice(9F) and getmajor(9F):

*devp = makedevice(getmajor(*devp), new_minor);

You do not have to call ddi_create_minor_node(9F) for the new minor. A driver must not change the major number of *devp. The driver must keep track of available minor numbers internally.

flag  Flag with bits to indicate whether the device is opened for reading (FREAD), writing (WRITE), or both. User threads issuing the open(2) system call can also request exclusive access to the device (FEXCL) or specify that the open should not block for any reason (FNDELAY), but the driver must enforce both cases. A driver for a write-only device such as a printer might consider an open(9E) for reading invalid.

otyp  Integer that indicates how open() was called. The driver must check that the value of otyp is appropriate for the device. For character drivers, otyp should be OTYP_CHR (see the open(9E) man page).

credp  Pointer to a credential structure containing information about the caller, such as the user ID and group IDs. Drivers should not examine the structure directly, but should instead use drv_priv(9F) to check for the common case of root privileges. In this example, only root or a user with the PRIV_SYS_DEVICES privilege is allowed to open the device for writing.

The following example shows a character driver open(9E) routine.

EXAMPLE 15-2  Character Driver open(9E) Routine

static int
xxopen(dev_t *devp, int flag, int otyp, cred_t *credp)
{
    minor_t instance;
    if (getminor(*devp) /* if device pointer is invalid */
        return (EINVAL);
    instance = getminor(*devp); /* one-to-one example mapping */
    /* Is the instance attached? */
    if (ddi_get_soft_state(statep, instance) == NULL)
        return (ENXIO);
    /* verify that otyp is appropriate */
    if (otyp != OTYP_CHR)
        return (EINVAL);
    if ((flag & FWRITE) && drv_priv(credp) == EPERM)
        return (EPERM);
    return (0);
}
close() Entry Point (Character Drivers)

The syntax for close(9E) is as follows:

```c
int xxclose(dev_t dev, int flag, int otyp, cred_t *credp);
```

close() should perform any cleanup necessary to finish using the minor device, and prepare the device (and driver) to be opened again. For example, the open routine might have been invoked with the exclusive access (FEXCL) flag. A call to close(9E) would allow additional open routines to continue. Other functions that close(9E) might perform are:

- Waiting for I/O to drain from output buffers before returning
- Rewinding a tape (tape device)
- Hanging up the phone line (modem device)

A driver that waits for I/O to drain could wait forever if draining stalls due to external conditions such as flow control. See "Threads Unable to Receive Signals" on page 75 for information about how to avoid this problem.

I/O Request Handling

This section discusses I/O request processing in detail.

User Addresses

When a user thread issues a write(2) system call, the thread passes the address of a buffer in user space:

```c
char buffer[] = "python";
count = write(fd, buffer, strlen(buffer) + 1);
```

The system builds a uio(9S) structure to describe this transfer by allocating an iovec(9S) structure and setting the iov_base field to the address passed to write(2), in this case, buffer. The uio(9S) structure is passed to the driver write(9E) routine. See "Vectored I/O" on page 267 for more information about the uio(9S) structure.

The address in the iovec(9S) is in user space, not kernel space. Thus, the address is neither guaranteed to be currently in memory nor to be a valid address. In either case, accessing a user address directly from the device driver or from the kernel could crash the system. Thus, device drivers should never access user addresses directly. Instead, a data transfer routine in the Oracle Solaris DDI/DKI should be used to transfer data into or out of the kernel. These routines can handle page faults. The DDI/DKI routines can bring in the proper user page to continue the copy transparently. Alternatively, the routines can return an error on an invalid access.
copyout(9F) can be used to copy data from kernel space to user space. copyin(9F) can copy data from user space to kernel space. ddi_copyout(9F) and ddi_copyin(9F) operate similarly but are to be used in the ioctl(9E) routine. copyin(9F) and copyout(9F) can be used on the buffer described by each iovec(9S) structure, or uiomove(9F) can perform the entire transfer to or from a contiguous area of driver or device memory.

**Vectored I/O**

In character drivers, transfers are described by a uio(9S) structure. The uio(9S) structure contains information about the direction and size of the transfer, plus an array of buffers for one end of the transfer. The other end is the device.

The uio(9S) structure contains the following members:

```c
iovec_t *uio_iov; /* base address of the iovec */
/* buffer description array */
int uio_iovcnt; /* the number of iovec structures */
on_t uio_offset; /* 32-bit offset into file where */
/* data is transferred from or to */
offset_t uio_loffset; /* 64-bit offset into file where */
/* data is transferred from or to */
uio_seg_t uio_segflg; /* identifies the type of I/O transfer */
/* UIO_SYSSPACE: kernel <-> kernel */
/* UIO_USERSPACE: kernel <-> user */
short uio_fmode; /* file mode flags (not driver settable) */
daddr_t uio_limit; /* 32-bit ulimit for file (maximum */
/* block offset). not driver settable. */
diskaddr_t uio_llimit; /* 64-bit ulimit for file (maximum block */
/* block offset). not driver settable. */
int uio_resid; /* amount (in bytes) not */
/* transferred on completion */
```

A uio(9S) structure is passed to the driver read(9E) and write(9E) entry points. This structure is generalized to support what is called **gather-write** and **scatter-read**. When writing to a device, the data buffers to be written do not have to be contiguous in application memory. Similarly, data that is transferred from a device into memory comes off in a contiguous stream but can go into noncontiguous areas of application memory. See the readv(2), writev(2), pread(2), and pwrite(2) man pages for more information on scatter-gather I/O.

Each buffer is described by an iovec(9S) structure. This structure contains a pointer to the data area and the number of bytes to be transferred.

```c
caddr_t iov_base; /* address of buffer */
int iov_len; /* amount to transfer */
```

The uio structure contains a pointer to an array of iovec(9S) structures. The base address of this array is held in uio_iov, and the number of elements is stored in uio_iovcnt.
The `uio_offset` field contains the 32-bit offset into the device at which the application needs to begin the transfer. `uio_offset` is used for 64-bit file offsets. If the device does not support the notion of an offset, these fields can be safely ignored. The driver should interpret either `uio_offset` or `uio_loffset`, but not both. If the driver has set the `D_64BIT` flag in the `cb_ops()` structure, that driver should use `uio_loffset`.

The `uio_resid` field starts out as the number of bytes to be transferred, that is, the sum of all the `iov_len` fields in `uio iov`. This field must be set by the driver to the number of bytes that were not transferred before returning. The `read(2)` and `write(2)` system calls use the return value from the `read(9E)` and `write(9E)` entry points to determine failed transfers. If a failure occurs, these routines return -1. If the return value indicates success, the system calls return the number of bytes requested minus `uio_resid`. If `uio_resid` is not changed by the driver, the `read(2)` and `write(2)` calls return 0. A return value of 0 indicates end-of-file, even though all the data has been transferred.

The support routines `uiomove(9F), physio(9F), and aphysio(9F)` update the `uio(9S)` structure directly. These support routines update the device offset to account for the data transfer. Neither the `uio_offset` or `uio_loffset` fields need to be adjusted when the driver is used with a seekable device that uses the concept of position. I/O performed to a device in this manner is constrained by the maximum possible value of `uio_offset` or `uio_loffset`. An example of such a usage is raw I/O on a disk.

If the device has no concept of position, the driver can take the following steps:

1. Save `uio_offset` or `uio_loffset`.
2. Perform the I/O operation.
3. Restore `uio_offset` or `uio_loffset` to the field's initial value.

I/O that is performed to a device in this manner is not constrained by the maximum possible value of `uio_offset` or `uio_loffset`. An example of this type of usage is I/O on a serial line.

The following example shows one way to preserve `uio_loffset` in the `read(9E)` function.

```c
static int
xxread(dev_t dev, struct uio *uio_p, cred_t *cred_p)
{
    offset_t off;
    /* ... */
    off = uio_p->uio_loffset; /* save the offset */
    /* do the transfer */
    uio_p->uio_loffset = off; /* restore it */
}
```
Differences Between Synchronous and Asynchronous I/O

Data transfers can be synchronous or asynchronous. The determining factor is whether the entry point that schedules the transfer returns immediately or waits until the I/O has been completed.

The `read(9E)` and `write(9E)` entry points are synchronous entry points. The transfer must not return until the I/O is complete. Upon return from the routines, the process knows whether the transfer has succeeded.

The `aread(9E)` and `awrite(9E)` entry points are asynchronous entry points. Asynchronous entry points schedule the I/O and return immediately. Upon return, the process that issues the request knows that the I/O is scheduled and that the status of the I/O must be determined later. In the meantime, the process can perform other operations.

With an asynchronous I/O request to the kernel, the process is not required to wait while the I/O is in process. A process can perform multiple I/O requests and allow the kernel to handle the data transfer details. Asynchronous I/O requests enable applications such as transaction processing to use concurrent programming methods to increase performance or response time. Any performance boost for applications that use asynchronous I/O, however, comes at the expense of greater programming complexity.

Data Transfer Methods

Data can be transferred using either programmed I/O or DMA. These data transfer methods can be used either by synchronous or by asynchronous entry points, depending on the capabilities of the device.

Programmed I/O Transfers

Programmed I/O devices rely on the CPU to perform the data transfer. Programmed I/O data transfers are identical to other read and write operations for device registers. Various data access routines are used to read or store values to device memory.

`uiomove(9F)` can be used to transfer data to some programmed I/O devices. `uiomove(9F)` transfers data between the user space, as defined by the `uio(9S)` structure, and the kernel. `uiomove()` can handle page faults, so the memory to which data is transferred need not be locked down. `uiomove()` also updates the `uio_resid` field in the `uio(9S)` structure. The following example shows one way to write a ramdisk `read(9E)` routine. It uses synchronous I/O and relies on the presence of the following fields in the ramdisk state structure:

```c
#include <sys/uio.h>

caddr_t ram; /* base address of ramdisk */
int ramsize; /* size of the ramdisk */
```
EXAMPLE 15-3  Ramdisk read(9E) Routine Using uiomove(9F)

```
static int
rd_read(dev_t dev, struct uio *uiop, cred_t *credp)
{
    rd_devstate_t *rsp;

    rsp = ddi_get_soft_state(rd_statep, getminor(dev));
    if (rsp == NULL)
        return (ENXIO);
    if (uiop->uio_offset >= rsp->ramsize)
        return (EINVAL);
    /* uiomove takes the offset into the kernel buffer,
    * the data transfer count (minimum of the requested and
    * the remaining data), the UIO_READ flag, and a pointer
    * to the uio structure.
    */
    return (uiomove(rsp->ram + uiop->uio_offset,
                    min(uiop->uio_resid, rsp->ramsize - uiop->uio_offset),
                    UIO_READ, uiop));
}
```

Another example of programmed I/O would be a driver that writes data one byte at a time directly to the device's memory. Each byte is retrieved from the uio(9S) structure by using uwritec(9F). The byte is then sent to the device. read(9E) can use ureadc(9F) to transfer a byte from the device to the area described by the uio(9S) structure.

EXAMPLE 15-4  Programmed I/O write(9E) Routine Using uwritec(9F)

```
static int
xxwrite(dev_t dev, struct uio *uiop, cred_t *credp)
{
    int value;
    struct xxstate *xsp;

    xsp = ddi_get_soft_state(statep, getminor(dev));
    if (xsp == NULL)
        return (ENXIO);
    /* if the device implements a power manageable component, do this: */
    pm_busy_component(xsp->dip, 0);
    if (xsp->pm_suspended)
        pm_raise_power(xsp->dip, normal power);
    while (uiop->uio_resid > 0) {
        /* do the programmed I/O access */
        value = uwritec(uiop);
        if (value == -1)
            return (EFAULT);
        ddi_put8(xsp->data_access_handle, &xsp->regp->data,
                 (uint8_t)value);
        ddi_put8(xsp->data_access_handle, &xsp->regp->csr,
                 START_TRANSFER);
    }
```
/*
 * this device requires a ten microsecond delay
 * between writes
 */
drv_usecwait(10);

pm_idle_component(xsp->dip, 0);
return (0);

DMA Transfers (Synchronous)

Character drivers generally use physio(9F) to do the setup work for DMA transfers in read(9E) and write(9E), as is shown in Example 15–5.

int physio(int (*strat)(struct buf *), struct buf **bp,
dev_t dev, int rw, void (*mincnt)(struct buf *),
struct uio **uio);

physio(9F) requires the driver to provide the address of a strategy(9E) routine. physio(9F) ensures that memory space is locked down, that is, memory cannot be paged out, for the duration of the data transfer. This lock-down is necessary for DMA transfers because DMA transfers cannot handle page faults. physio(9F) also provides an automated way of breaking a larger transfer into a series of smaller, more manageable ones. See “minphys() Entry Point” on page 273 for more information.

EXAMPLE 15–5  read(9E) and write(9E) Routines Using physio(9F)

static int
xxread(dev_t dev, struct uio *uiop, cred_t *credp)
{
    struct xxstate *xsp;
    int ret;
    xsp = ddi_get_soft_state(statep, getminor(dev));
    if (xsp == NULL)
        return (ENXIO);
    ret = physio(xxstrategy, NULL, dev, B_READ, xxminphys, uiop);
    return (ret);
}

static int
xxwrite(dev_t dev, struct uio *uiop, cred_t *credp)
{
    struct xxstate *xsp;
    int ret;
    xsp = ddi_get_soft_state(statep, getminor(dev));
    if (xsp == NULL)
        return (ENXIO);
    ret = physio(xxstrategy, NULL, dev, B_WRITE, xxminphys, uiop);
return (ret);
}

In the call to `physio(9F)`, `xxstrategy` is a pointer to the driver `strategy()` routine. Passing NULL as the `buf(9S)` structure pointer tells `physio(9F)` to allocate a `buf(9S)` structure. If the driver must provide `physio(9F)` with a `buf(9S)` structure, `getrbuf(9F)` should be used to allocate the structure. `physio(9F)` returns zero if the transfer completes successfully, or an error number on failure. After calling `strategy(9E)`, `physio(9F)` calls `biowait(9F)` to block until the transfer either completes or fails. The return value of `physio(9F)` is determined by the error field in the `buf(9S)` structure set by `bioerror(9F)`.

**DMA Transfers (Asynchronous)**

Character drivers that support `aread(9E)` and `awrite(9E)` use `aphysio(9F)` instead of `physio(9F)`.

```
int aphysio(int (*strat)(struct buf *), int (*cancel)(struct buf *),
    dev_t dev, int rw, void (*mincnt)(struct buf *),
    struct aio_req *aio_req);
```

**Note** – The address of `anocancel(9F)` is the only value that can currently be passed as the second argument to `aphysio(9F)`.

`aphysio(9F)` requires the driver to pass the address of a `strategy(9E)` routine. `aphysio(9F)` ensures that memory space is locked down, that is, cannot be paged out, for the duration of the data transfer. This lock-down is necessary for DMA transfers because DMA transfers cannot handle page faults. `aphysio(9F)` also provides an automated way of breaking a larger transfer into a series of smaller, more manageable ones. See “`minphys()` Entry Point” on page 273 for more information.

Example 15–5 and Example 15–6 demonstrate that the `aread(9E)` and `awrite(9E)` entry points differ only slightly from the `read(9E)` and `write(9E)` entry points. The difference is primarily in their use of `aphysio(9F)` instead of `physio(9F)`.

**EXAMPLE 15–6  aread(9E) and awrite(9E) Routines Using aphysio(9F)**

```
static int
xxaread(dev_t dev, struct aio_req *aio_p, cred_t *cred_p)
{
    struct xxstate *xsp;
    xsp = ddi_get_soft_state(statep, getminor(dev));
    if (xsp == NULL)
        return (ENXIO);
```
EXAMPLE 15–6    aread(9E) and awrite(9E) Routines Using aphysio(9F)  (Continued)

        return (aphysio(xxstrategy, anocancel, dev, B_READ,
        xxminphys, aiop));
    }

static int
xxawrite(dev_t dev, struct aio_req *aiop, cred_t *cred_p)
{
    struct xxstate *xsp;
    xsp = ddi_get_soft_state(statep, getminor(dev));
    if (xsp == NULL)
        return (ENXIO);
    return (aphysio(xxstrategy, anocancel, dev, B_WRITE,
        xxminphys, aiop));
}

In the call to aphysio(9F), xxstrategy() is a pointer to the driver strategy routine. aiop is a pointer to the aio_req(9S) structure. aiop is passed to aread(9E) and awrite(9E). aio_req(9S) describes where the data is to be stored in user space. aphysio(9F) returns zero if the I/O request is scheduled successfully or an error number on failure. After calling strategy(9E), aphysio(9F) returns without waiting for the I/O to complete or fail.

minphys() Entry Point

The minphys() entry point is a pointer to a function to be called by physio(9F) or aphysio(9F). The purpose of xxminphys is to ensure that the size of the requested transfer does not exceed a driver-imposed limit. If the user requests a larger transfer, strategy(9E) is called repeatedly, requesting no more than the imposed limit at a time. This approach is important because DMA resources are limited. Drivers for slow devices, such as printers, should be careful not to tie up resources for a long time.

Usually, a driver passes the address of the kernel function minphys(9F), but the driver can define its own xxminphys() routine instead. The job of xxminphys() is to keep the b_bcount field of the buf(9S) structure under a driver's limit. The driver should adhere to other system limits as well. For example, the driver's xxminphys() routine should call the system minphys(9F) routine after setting the b_bcount field and before returning.

EXAMPLE 15–7    minphys(9F) Routine
#define XXMINVAL (512 << 10) /* 512 KB */
static void
xxminphys(struct buf *bp)
{
    if (bp->b_bcount > XXMINVAL)
        bp->b_bcount = XXMINVAL;
    minphys(bp);
}
strategy() Entry Point

The strategy(9E) routine originated in block drivers. The strategy function got its name from implementing a strategy for efficient queuing of I/O requests to a block device. A driver for a character-oriented device can also use a strategy(9E) routine. In the character I/O model presented here, strategy(9E) does not maintain a queue of requests, but rather services one request at a time.

In the following example, the strategy(9E) routine for a character-oriented DMA device allocates DMA resources for synchronous data transfer. strategy() starts the command by programming the device register. See Chapter 9, “Direct Memory Access (DMA),” for a detailed description.

Note – strategy(9E) does not receive a device number (dev_t) as a parameter. Instead, the device number is retrieved from the b_edev field of the buf(9S) structure passed to strategy(9E).

EXAMPLE 15-8 strategy(9E) Routine

```c
static int
xxstrategy(struct buf *bp)
{
    minor_t instance;
    struct xxstate  *xsp;
    ddi_dma_cookie_t cookie;

    instance = getminor(bp->b_edev);
    xsp = ddi_get_soft_state(statep, instance);
    /* ... */
    /* If the device has power manageable components,
    * mark the device busy with pm_busy_components(9F),
    * and then ensure that the device is
    * powered up by calling pm_raise_power(9F).
    */
    /* Set up DMA resources with ddi_dma_alloc_handle(9F) and
    * ddi_dma_buf_bind_handle(9F).
    */
    xsp->bp = bp; /* remember bp */
    /* Program DMA engine and start command */
    return (0);
}
```

Note – Although strategy() is declared to return an int, strategy() must always return zero.

On completion of the DMA transfer, the device generates an interrupt, causing the interrupt routine to be called. In the following example, xxintr() receives a pointer to the state structure for the device that might have generated the interrupt.
EXAMPLE 15–9  Interrupt Routine

```c
static u_int
xxintr(caddr_t arg)
{
  struct xxstate *xsp = (struct xxstate *)arg;
  if ( /* device did not interrupt */ ) {
    return (DDI_INTR_UNCLAIMED);
  }
  if ( /* error */ ) {
    /* error handling */
  }
  /* Release any resources used in the transfer, such as DMA resources.
  * ddi_dma_unbind_handle(9F) and ddi_dma_free_handle(9F)
  * Notify threads that the transfer is complete.
  */
  biodone(xsp->bp);
  return (DDI_INTRCLAIMED);
}
```

The driver indicates an error by calling `bioerror(9F)`. The driver must call `biodone(9F)` when the transfer is complete or after indicating an error with `bioerror(9F)`.

### Mapping Device Memory

Some devices, such as frame buffers, have memory that is directly accessible to user threads by way of memory mapping. Drivers for these devices typically do not support the `read(9E)` and `write(9E)` interfaces. Instead, these drivers support memory mapping with the `devmap(9E)` entry point. For example, a frame buffer driver might implement the `devmap(9E)` entry point to enable the frame buffer to be mapped in a user thread.

The `devmap(9E)` entry point is called to export device memory or kernel memory to user applications. The `devmap()` function is called from `devmap_setup(9F)` inside `segmap(9E)` or on behalf of `ddi_devmap_segmap(9F)`.

The `segmap(9E)` entry point is responsible for setting up a memory mapping requested by an `mmap(2)` system call. Drivers for many memory-mapped devices use `ddi_devmap_segmap(9F)` as the entry point rather than defining their own `segmap(9E)` routine.

Multiplexing I/O on File Descriptors

A thread sometimes needs to handle I/O on more than one file descriptor. One example is an application program that needs to read the temperature from a temperature-sensing device and then report the temperature to an interactive display. A program that makes a read request with no data available should not block while waiting for the temperature before interacting with the user again.

The `poll(2)` system call provides users with a mechanism for multiplexing I/O over a set of file descriptors that reference open files. `poll(2)` identifies those file descriptors on which a program can send or receive data without blocking, or on which certain events have occurred.

To enable a program to poll a character driver, the driver must implement the `chpoll(9E)` entry point. The system calls `chpoll(9E)` when a user process issues a `poll(2)` system call on a file descriptor associated with the device. The `chpoll(9E)` entry point routine is used by non-STREAMS character device drivers that need to support polling.

The `chpoll(9E)` function uses the following syntax:

```c
int xxchpoll(dev_t dev, short events, int anyyet, short *reventsp,
    struct pollhead **phpp);
```

In the `chpoll(9E)` entry point, the driver must follow these rules:

- Implement the following algorithm when the `chpoll(9E)` entry point is called:

  ```c
  if ( /* events are satisfied now */ ) {
    *reventsp = mask_of_satisfied_events
  } else {
    *reventsp = 0;
    if (!anyyet)
      *phpp = &local_pollhead_structure;
  }
  return (0);
  ```

  See the `chpoll(9E)` man page for a discussion of events to check. The `chpoll(9E)` entry point should then return the mask of satisfied events by setting the return events in `*reventsp`.

  If no events have occurred, the return field for the events is cleared. If the `anyyet` field is not set, the driver must return an instance of the `pollhead` structure. The `pollhead` structure is usually allocated in a state structure. The `pollhead` structure should be treated as opaque by the driver. None of the `pollhead` fields should be referenced.

- Call `pollwakeup(9F)` whenever a device condition of type `events`, listed in Example 15–10, occurs. This function should be called only with one event at a time. You can call `pollwakeup(9F)` in the interrupt routine when the condition has occurred.
Example 15–10 and Example 15–11 show how to implement the polling discipline and how to use pollwakeup(9F).

The following example shows how to handle the POLLIN and POLLERR events. The driver first reads the status register to determine the current state of the device. The parameter events specifies which conditions the driver should check. If an appropriate condition has occurred, the driver sets that bit in *reventsp. If none of the conditions has occurred and if anyyet is not set, the address of the pollhead structure is returned in *phpp.

**EXAMPLE 15–10  chpoll(9E) Routine**

```c
static int
xxchpoll(dev_t dev, short events, int anyyet,
    short *reventsp, struct pollhead **phpp)
{
    uint8_t status;
    short revent;
    struct xxstate *xsp;
    xsp = ddi_get_soft_state(statep, getminor(dev));
    if (xsp == NULL)
        return (ENXIO);
    revent = 0;
    /*
    * Valid events are:
    * POLLIN | POLLOUT | POLLPRI | POLLHUP | POLLERR
    * This example checks only for POLLIN and POLLERR.
    */
    status = ddi_get8(xsp->data_access_handle, &xsp->regp->csr);
    if ((events & POLLIN) &&
        dataavailabletoread)
        revent |= POLLIN;
    if (status & DEVICE_ERROR) {
        revent |= POLLERR;
    } /* if nothing has occurred */
    if (revent == 0) {
        if (!anyyet) {
            *phpp = &xsp->pollhead;
        }
        *reventsp = revent;
    }
    return (0);
}
```

The following example shows how to use the pollwakeup(9F) function. The pollwakeup(9F) function usually is called in the interrupt routine when a supported condition has occurred. The interrupt routine reads the status from the status register and checks for the conditions. The routine then calls pollwakeup(9F) for each event to possibly notify polling threads that they should check again. Note that pollwakeup(9F) should not be called with any locks held, since deadlock could result if another routine tried to enter chpoll(9E) and grab the same lock.
EXAMPLE 15–11  Interrupt Routine Supporting chpoll(9E)

```c
static u_int
xxintr(caddr_t arg)
{
    struct xxstate *xsp = (struct xxstate *)arg;
    uint8_t status;
    /* normal interrupt processing */
    /* ... */
    status = ddi_get8(xsp->data_access_handle, &xsp->regp->csr);
    if (status & DEVICE_ERROR) {
        pollwakeup(&xsp->pollhead, POLLERR);
    }
    if ( /* just completed a read */ ) {
        pollwakeup(&xsp->pollhead, POLLIN);
    }
    /* ... */
    return (DDI_INTR_CLAIMED);
}
```

### Miscellaneous I/O Control

The `ioctl(9E)` routine is called when a user thread issues an `ioctl(2)` system call on a file descriptor associated with the device. The I/O control mechanism is a catchall for getting and setting device-specific parameters. This mechanism is frequently used to set a device-specific mode, either by setting internal driver software flags or by writing commands to the device. The control mechanism can also be used to return information to the user about the current device state. In short, the control mechanism can do whatever the application and driver need to have done.

### `ioctl()` Entry Point (Character Drivers)

```c
int xxioctl(dev_t dev, int cmd, intptr_t arg, int mode,
           cred_t *credp, int *rvalp);
```

The `cmd` parameter indicates which command `ioctl(9E)` should perform. By convention, the driver with which an I/O control command is associated is indicated in bits 8-15 of the command. Typically, the ASCII code of a character represents the driver. The driver-specific command in bits 0-7. The creation of some I/O commands is illustrated in the following example:

```c
#define XXIOC ('x' << 8) /* 'x' is a character that represents device xx */
#define XX_GET_STATUS (XXIOC | 1) /* get status register */
#define XX_SET_CMD (XXIOC | 2) /* send command */
```

The interpretation of `arg` depends on the command. I/O control commands should be documented in the driver documentation or a man page. The command should also be defined in a public header file, so that applications can determine the name of the command, what the
command does, and what the command accepts or returns as _arg_. Any data transfer of _arg_ into or out of the driver must be performed by the driver.

Certain classes of devices such as frame buffers or disks must support standard sets of I/O control requests. These standard I/O control interfaces are documented in the *Solaris 8 Reference Manual Collection*. For example, _fbio(7I)_ documents the I/O controls that frame buffers must support, and _dkio(7I)_ documents standard disk I/O controls. See “Miscellaneous I/O Control” on page 278 for more information on I/O controls.

Drivers must use _ddi_copyin(9F)_ to transfer _arg_ data from the user-level application to the kernel level. Drivers must use _ddi_copyout(9F)_ to transfer data from the kernel to the user level. Failure to use _ddi_copyin(9F)_ or _ddi_copyout(9F)_ can result in panics under two conditions. A panic occurs if the architecture separates the kernel and user address spaces, or if the user address has been swapped out.

_ioctl(9E) is usually a switch statement with a case for each supported ioctl(9E) request.

**EXAMPLE 15–12 ioctl(9E) Routine**

```c
static int
xxioctl(dev_t dev, int cmd, intptr_t arg, int mode,
    cred_t *credp, int *rvalp)
{
    uint8_t csr;
    struct xxstate *xsp;
    xsp = ddi_get_soft_state(statep, getminor(dev));
    if (xsp == NULL) {
        return (ENXIO);
    }
    switch (cmd) {
        case XX_GET_STATUS:
            csr = ddi_get8(xsp->data_access_handle, &xsp->regp->csr);
            if (ddi_copyout(&csr, (void *)arg, sizeof (uint8_t), mode) != 0) {
                return (EFAULT);
            }
            break;
        case XX_SET_CMD:
            if (ddi_copyin((void *)arg, &csr, sizeof (uint8_t), mode) != 0) {
                return (EFAULT);
            }
            ddi_put8(xsp->data_access_handle, &xsp->regp->csr, csr);
            break;
        default:
            /* generic "ioctl unknown" error */
            return (ENOTTY);
    }
    return (0);
}
```

The _cmd_ variable identifies a specific device control operation. A problem can occur if _arg_ contains a user virtual address. ioctl(9E) must call _ddi_copyin(9F)_ or _ddi_copyout(9F)_ to
transfer data between the data structure in the application program pointed to by arg and the driver. In **Example 15–12**, for the case of an XX_GET_STATUS request, the contents of xsp->regp->csr are copied to the address in arg. ioctl(9E) can store in *rvalp any integer value as the return value to the ioctl(2) system call that makes a successful request. Negative return values, such as -1, should be avoided. Many application programs assume that negative values indicate failure.

The following example demonstrates an application that uses the I/O controls discussed in the previous paragraph.

**EXAMPLE 15–13** Using ioctl(9E)

```c
#include <sys/types.h>
#include "xxio.h"  /* contains device's ioctl cmds and args */

int main(void)
{
  uint8_t status;
  /* ... */
  /* read the device status */
  if (ioctl(fd, XX_GET_STATUS, &status) == -1) {
    /* error handling */
  }
  printf("device status %x\n", status);
  exit(0);
}
```

**I/O Control Support for 64-Bit Capable Device Drivers**

The Oracle Solaris kernel runs in 64-bit mode on suitable hardware, supporting both 32-bit applications and 64-bit applications. A 64-bit device driver is required to support I/O control commands from programs of both sizes. The difference between a 32-bit program and a 64-bit program is the C language type model. A 32-bit program is ILP32, and a 64-bit program is LP64. See Appendix C, "Making a Device Driver 64-Bit Ready," for information on C data type models.

If data that flows between programs and the kernel is not identical in format, the driver must be able to handle the model mismatch. Handling a model mismatch requires making appropriate adjustments to the data.

To determine whether a model mismatch exists, the ioctl(9E) mode parameter passes the data model bits to the driver. As **Example 15–14** shows, the mode parameter is then passed to ddi_model_convert_from(9F) to determine whether any model conversion is necessary.

A flag subfield of the mode argument is used to pass the data model to the ioctl(9E) routine. The flag is set to one of the following:
FNATIVE is conditionally defined to match the data model of the kernel implementation. The FMODELS mask should be used to extract the flag from the mode argument. The driver can then examine the data model explicitly to determine how to copy the application data structure.

The DDI function ddi_model_convert_from(9F) is a convenience routine that can assist some drivers with their ioctl() calls. The function takes the data type model of the user application as an argument and returns one of the following values:

- DDI_MODEL_ILP32 – Convert from ILP32 application
- DDI_MODEL_NONE – No conversion needed

DDI_MODEL_NONE is returned if no data conversion is necessary, as occurs when the application and driver have the same data model. DDI_MODEL_ILP32 is returned to a driver that is compiled to the LP64 model and that communicates with a 32-bit application.

In the following example, the driver copies a data structure that contains a user address. The data structure changes size from ILP32 to LP64. Accordingly, the 64-bit driver uses a 32-bit version of the structure when communicating with a 32-bit application.

**EXAMPLE 15-14  ioctl(9E) Routine to Support 32-bit Applications and 64-bit Applications**

```c
struct args32 {
    uint32_t     addr;  /* 32-bit address in LP64 */
    int          len;
};
struct args {
    caddr_t     addr;  /* 64-bit address in LP64 */
    int          len;
};

static int
xxioctl(dev_t dev, int cmd, intptr_t arg, int mode,
        cred_t *credp, int *rvalp)
{
    struct xxstate  *xsp;
    struct args    a;
    xsp = ddi_get_soft_state(statep, getminor(dev));
    if (xsp == NULL) {
        return (ENXIO);
    } else {
        switch (cmd) {
        case XX_COPYIN_DATA:
            switch(ddi_model_convert_from(mode)) {
            case DDI_MODEL_ILP32:
                {
                    struct args32 a32;
                    /* copy 32-bit args data shape */
                    if (ddi_copyin((void *)arg, 6a32,
                                  sizeof (struct args32), mode) != 0) {
```
EXAMPLE 15–14  ioctl(9E) Routine to Support 32-bit Applications and 64-bit Applications
(Continued)

    return (EFAULT);

  } /* convert 32-bit to 64-bit args data shape */
  a.addr = a32.addr;
  a.len = a32.len;
  break;

  case DDI_MODEL_NONE:
    /* application and driver have same data model. */
    if (ddi_copyin((void *)arg, &a, sizeof (struct args),
      mode) != 0) { 
      return (EFAULT);
    }
    /* continue using data shape in native driver data model. */
    break;

  case XX_COPYOUT_DATA:
    /* copyout handling */
    break;
  default:
    /* generic "ioctl unknown" error */
    return (ENOTTY);
  }

return (0);

Handling copyout() Overflow

Sometimes a driver needs to copy out a native quantity that no longer fits in the 32-bit sized structure. In this case, the driver should return EOVERFLOW to the caller. EOVERFLOW serves as an indication that the data type in the interface is too small to hold the value to be returned, as shown in the following example.

EXAMPLE 15–15  Handling copyout(9F) Overflow

int
xxioctl(dev_t dev, int cmd, intptr_t arg, int mode,
  cred_t *cr, int *rval_p)
{
  struct resdata res;
  /* body of driver */
  switch (ddi_model_convert_from(mode & FMODELS)) {
    case DDI_MODEL_ILP32: {
      struct resdata32 res32;
      if (res.size > UINT_MAX)
        return (Eoverflow);
      res32.size = (size32_t)res.size;
      res32.flag = res.flag;
    }
EXAMPLE 15–15 Handling copyout(9F) Overflow (Continued)

```c
    if (ddi_copyout(&res32,
           (void *)&arg, sizeof (res32), mode))
       return (EFAULT);
} break;

case DDI_MODEL_NONE:
    if (ddi_copyout(&res, (void *)&arg, sizeof (res), mode))
       return (EFAULT);
   break;

return (0);
```

32-bit and 64-bit Data Structure Macros

The method in Example 15–15 works well for many drivers. An alternate scheme is to use the data structure macros that are provided in `<sys/model.h>` to move data between the application and the kernel. These macros make the code less cluttered and behave identically, from a functional perspective.

EXAMPLE 15–16 Using Data Structure Macros to Move Data

```c
int  
xxioctl(dev_t dev, int cmd, intptr_t arg, int mode,
        cred_t *cr, int *rval_p)
{
    STRUCT_DECL(opdata, op);

    if (cmd != OPONE)
       return (ENOTTY);

    STRUCT_INIT(op, mode);

    if (copyin((void *)&arg,
                  STRUCT_BUF(op), STRUCT_SIZE(op)))
       return (EFAULT);

    if (STRUCT_FGET(op, flag) != XXACTIVE ||
         STRUCT_FGET(op, size) > XXSIZE)
       return (EINVAL);
    xxdowork(device_state, STRUCT_FGET(op, size));
    return (0);
}
```
How Do the Structure Macros Work?

In a 64-bit device driver, structure macros enable the use of the same piece of kernel memory by data structures of both sizes. The memory buffer holds the contents of the native form of the data structure, that is, the LP64 form, and the ILP32 form. Each structure access is implemented by a conditional expression. When compiled as a 32-bit driver, only one data model, the native form, is supported. No conditional expression is used.

The 64-bit versions of the macros depend on the definition of a shadow version of the data structure. The shadow version describes the 32-bit interface with fixed-width types. The name of the shadow data structure is formed by appending ”32” to the name of the native data structure. For convenience, place the definition of the shadow structure in the same file as the native structure to ease future maintenance costs.

The macros can take the following arguments:

- **structname**: The structure name of the native form of the data structure as entered after the `struct` keyword.
- **umodel**: A flag word that contains the user data model, such as FILP32 or FLIP64, extracted from the mode parameter of `ioctl(9E)`.
- **handle**: The name used to refer to a particular instance of a structure that is manipulated by these macros.
- **fieldname**: The name of the field within the structure.

When to Use Structure Macros

Macros enable you to make in-place references only to the fields of a data item. Macros do not provide a way to take separate code paths that are based on the data model. Macros should be avoided if the number of fields in the data structure is large. Macros should also be avoided if the frequency of references to these fields is high.

Macros hide many of the differences between data models in the implementation of the macros. As a result, code written with this interface is generally easier to read. When compiled as a 32-bit driver, the resulting code is compact without needing clumsy #ifdefs, but still preserves type checking.

Declaring and Initializing Structure Handles

`STRUCT_DECL(9F)` and `STRUCT_INIT(9F)` can be used to declare and initialize a handle and space for decoding an `ioctl` on the stack. `STRUCT_HANDLE(9F)` and `STRUCT_SET_HANDLE(9F)` declare
and initialize a handle without allocating space on the stack. The latter macros can be useful if
the structure is very large, or is contained in some other data structure.

Note – Because the STRUCT DECL(9F) and STRUCT HANDLE(9F) macros expand to data structure
declarations, these macros should be grouped with such declarations in C code.

The macros for declaring and initializing structures are as follows:

STRUCT_DECL(structname, handle)
Declarations a structure handle that is called handle for a structname data structure.
STRUCT DECL allocates space for its native form on the stack. The native form is assumed to
be larger than or equal to the ILP32 form of the structure.

STRUCT_INIT(handle, umodel)
Initializes the data model for handle to umodel. This macro must be invoked before any
access is made to a structure handle declared with STRUCT DECL(9F).

STRUCT_HANDLE(structname, handle)
Declares a structure handle that is called handle. Contrast with STRUCT DECL(9F).

STRUCT_SET_HANDLE(handle, umodel, addr)
Initializes the data model for handle to umodel, and sets addr as the buffer used for
subsequent manipulation. Invoke this macro before accessing a structure handle declared
with STRUCT DECL(9F).

Operations on Structure Handles

The macros for performing operations on structures are as follows:

size_t STRUCT_SIZE(handle)
Returns the size of the structure referred to by handle, according to its embedded data
model.

typeof fieldname STRUCT_FGET(handle, fieldname)
Returns the indicated field in the data structure referred to by handle. This field is a
non-pointer type.

typeof fieldname STRUCT_FGETP(handle, fieldname)
Returns the indicated field in the data structure referred to by handle. This field is a pointer
type.

STRUCT_FSET(handle, fieldname, val)
Sets the indicated field in the data structure referred to by handle to value val. The type of val
should match the type of fieldname. The field is a non-pointer type.
STRUCT_FSETP(handle, fieldname, val)
Sets the indicated field in the data structure referred to by handle to value val. The field is a pointer type.

typedef fieldname *STRUCT_FADDR(handle, fieldname)
Returns the address of the indicated field in the data structure referred to by handle.

struct structname *STRUCT_BUF(handle)
Returns a pointer to the native structure described by handle.

Other Operations
Some miscellaneous structure macros follow:

size_t SIZEOF_STRUCT(struct_name, datamodel)
Returns the size of struct_name, which is based on the given data model.

size_t SIZEOF_PTR(datamodel)
Returns the size of a pointer based on the given data model.
Drivers for Block Devices

This chapter describes the structure of block device drivers. The kernel views a block device as a set of randomly accessible logical blocks. The file system uses a list of buf(9S) structures to buffer the data blocks between a block device and the user space. Only block devices can support a file system.

This chapter provides information on the following subjects:

- “Block Driver Structure Overview” on page 287
- “File I/O” on page 288
- “Block Device Autoconfiguration” on page 289
- “Controlling Device Access” on page 291
- “Synchronous Data Transfers (Block Drivers)” on page 295
- “Asynchronous Data Transfers (Block Drivers)” on page 299
- “dump() and print() Entry Points” on page 303
- “Disk Device Drivers” on page 304

Block Driver Structure Overview

Figure 16–1 shows data structures and routines that define the structure of a block device driver. Device drivers typically include the following elements:

- Device-loadable driver section
- Device configuration section
- Device access section

The shaded device access section in the following figure illustrates entry points for block drivers.
Associated with each device driver is a `dev_ops(9S)` structure, which in turn refers to a `cb_ops(9S)` structure. See Chapter 6, “Driver Autoconfiguration,” for details on driver data structures.

Block device drivers provide these entry points:

- `open(9E)`
- `close(9E)`
- `strategy(9E)`
- `print(9E)`

**Note** – Some of the entry points can be replaced by `nodev(9F)` or `nulldev(9F)` as appropriate.

## File I/O

A file system is a tree-structured hierarchy of directories and files. Some file systems, such as the UNIX File System (UFS), reside on block-oriented devices. File systems are created by `format(1M)` and `newfs(1M)`.

When an application issues a `read(2)` or `write(2)` system call to an ordinary file on the UFS file system, the file system can call the device driver `strategy(9E)` entry point for the block device on which the file system resides. The file system code can call `strategy(9E)` several times for a single `read(2)` or `write(2)` system call.
The file system code determines the logical device address, or *logical block number*, for each ordinary file block. A block I/O request is then built in the form of a *buf*(9S) structure directed at the block device. The driver *strategy*(9E) entry point then interprets the *buf*(9S) structure and completes the request.

### Block Device Autoconfiguration

*attach*(9E) should perform the common initialization tasks for each instance of a device:

- Allocating per-instance state structures
- Mapping the device’s registers
- Registering device interrupts
- Initializing mutex and condition variables
- Creating power manageable components
- Creating minor nodes

Block device drivers create minor nodes of type S_IFBLK. As a result, a block special file that represents the node appears in the /devices hierarchy.

Logical device names for block devices appear in the /dev/dsk directory, and consist of a controller number, bus-address number, disk number, and slice number. These names are created by the *devfsadm*(1M) program if the node type is set to DDI_NT_BLOCK or DDI_NT_BLOCK_CHAN. DDI_NT_BLOCK_CHAN should be specified if the device communicates on a channel, that is, a bus with an additional level of addressability. SCSI disks are a good example. DDI_NT_BLOCK_CHAN causes a bus-address field (tN) to appear in the logical name. DDI_NT_BLOCK should be used for most other devices.

A minor device refers to a partition on the disk. For each minor device, the driver must create an *nbblocks* or *Nblocks* property. This integer property gives the number of blocks supported by the minor device expressed in units of DEV_BSIZE, that is, 512 bytes. The file system uses the *nbblocks* and *Nblocks* properties to determine device limits. *Nblocks* is the 64-bit version of *nbblocks*. *Nblocks* should be used with storage devices that can hold over 1 Tbyte of storage per disk. See “Device Properties” on page 77 for more information.

Example 16–1 shows a typical *attach*(9E) entry point with emphasis on creating the device’s minor node and the *Nblocks* property. Note that because this example uses *Nblocks* and not *nbblocks*, *ddi_prop_update_int64*(9F) is called instead of *ddi_prop_update_int*(9F).

As a side note, this example shows the use of *makedevice*(9F) to create a device number for *ddi_prop_update_int64()*(). The *makedevice* function makes use of *ddi_driver_major*(9F), which generates a major number from a pointer to a dev_info_t structure. Using *ddi_driver_major()* is similar to using *getmajor*(9F), which gets a dev_t structure pointer.

**Example 16–1** Block Driver attach() Routine

```c
static int
xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
```
int instance = ddi_get_instance(dip);
switch (cmd) {
    case DDI_ATTACH:
        /*
         * allocate a state structure and initialize it
         * map the devices registers
         * add the device driver's interrupt handler(s)
         * initialize any mutexes and condition variables
         * read label information if the device is a disk
         * create power manageable components
         *
         * Create the device minor node. Note that the node_type
         * argument is set to DDI_NT_BLOCK.
         */
        if (ddi_create_minor_node(dip, "minor_name", S_IFBLK, instance, DDI_NT_BLOCK, 0) == DDI_FAILURE) {
            /* free resources allocated so far */
            /* Remove any previously allocated minor nodes */
            ddi_remove_minor_node(dip, NULL);
            return (DDI_FAILURE);
        }
        /*
         * Create driver properties like "Nblocks". If the device
         * is a disk, the Nblocks property is usually calculated from
         * information in the disk label. Use "Nblocks" instead of
         * "nblocks" to ensure the property works for large disks.
         */
        xsp->Nblocks = size;
        /* size is the size of the device in 512 byte blocks */
        maj_number = ddi_driver_major(dip);
        if (ddi_prop_update_int64(makedevice(maj_number, instance), dip, "Nblocks", xsp->Nblocks) != DDI_PROP_SUCCESS) {
            /* free resources allocated so far */
            return (DDI_FAILURE);
        }
        xsp->open = 0;
        xsp->nlayered = 0;
        /* ... */
        return (DDI_SUCCESS);
    case DDI_RESUME:
        /* For information, see Chapter 12, "Power Management," in this book. */
        default:
            return (DDI_FAILURE);
}
Controlling Device Access

This section describes the entry points for open() and close() functions in block device drivers. See Chapter 15, "Drivers for Character Devices," for more information on open(9E) and close(9E).

open() Entry Point (Block Drivers)

The open(9E) entry point is used to gain access to a given device. The open(9E) routine of a block driver is called when a user thread issues an open(2) or mount(2) system call on a block special file associated with the minor device, or when a layered driver calls open(9E). See "File I/O" on page 288 for more information.

The open() entry point should check for the following conditions:

- The device can be opened, that is, the device is online and ready.
- The device can be opened as requested. The device supports the operation. The device's current state does not conflict with the request.
- The caller has permission to open the device.

The following example demonstrates a block driver open(9E) entry point.

EXAMPLE 16-2  Block Driver open(9E) Routine

```c
static int
xxopen(dev_t *devp, int flags, int otyp, cred_t *credp)
{
    minor_t	instance;
    struct xxstate *xsp;

    instance = getminor(*devp);
    xsp = ddi_get_soft_state(statep, instance);
    if (xsp == NULL)
        return (ENXIO);
    mutex_enter(&xsp->mu);
    /*
    * only honor FEXCL. If a regular open or a layered open
    * is still outstanding on the device, the exclusive open
    * must fail.
    */
    if ((flags & FEXCL) && (xsp->open || xsp->nlayered)) {
        mutex_exit(&xsp->mu);
        return (EAGAIN);
    }
    switch (otyp) {
        case OTYP_LYR:
            xsp->nlayered++;
            break;
        case OTYP_BLK:
            xsp->open = 1;
```
The `otyp` argument is used to specify the type of open on the device. `OTYP_BLK` is the typical open type for a block device. A device can be opened several times with `otyp` set to `OTYP_BLK`. `close(9E)` is called only once when the final close of type `OTYP_BLK` has occurred for the device. `otyp` is set to `OTYP_LYR` if the device is being used as a layered device. For every open of type `OTYP_LYR`, the layering driver issues a corresponding close of type `OTYP_LYR`. The example keeps track of each type of open so the driver can determine when the device is not being used in `close(9E)`.

**c**lose( ) Entry **P**oint (B**lock** **D**rivers)**

The `close(9E)` entry point uses the same arguments as `open(9E)` with one exception. `dev` is the device number rather than a pointer to the device number.

The `close()` routine should verify `otyp` in the same way as was described for the `open(9E)` entry point. In the following example, `close()` must determine when the device can really be closed. Closing is affected by the number of block opens and layered opens.

**EXAMPLE 16–3  Block Device close(9E) Routine**

```c
static int xxclose(dev_t dev, int flag, int otyp, cred_t *credp)
{
    minor_t instance;
    struct xxstate *xsp;

    instance = getminor(dev);
    xsp = ddi_get_soft_state(statep, instance);
    if (xsp == NULL)
        return (ENXIO);
    mutex_enter(&xsp->mu);
    switch (otyp) {
        case OTYP_LYR:
            xsp->nlayered--;
            break;
        case OTYP_BLK:
            xsp->open = 0;
            break;
        default:
            mutex_exit(&xsp->mu);
    }
    mutex_exit(&xsp->mu);
    return (0);
}
```
return (EINVAL);
}

if (xsp->open || xsp->nlayered) {
  /* not done yet */
  mutex_exit(&xsp->mu);
  return (0);
}

/* cleanup (rewind tape, free memory, etc.) */
/* wait for I/O to drain */
mutex_exit(&xsp->mu);
return (0);
}

strategy() Entry Point

The strategy(9E) entry point is used to read and write data buffers to and from a block device. The name strategy refers to the fact that this entry point might implement some optimal strategy for ordering requests to the device.

strategy(9E) can be written to process one request at a time, that is, a synchronous transfer. strategy() can also be written to queue multiple requests to the device, as in an asynchronous transfer. When choosing a method, the abilities and limitations of the device should be taken into account.

The strategy(9E) routine is passed a pointer to a buf(9S) structure. This structure describes the transfer request, and contains status information on return. buf(9S) and strategy(9E) are the focus of block device operations.

buf Structure

The following buf structure members are important to block drivers:

```c
int       b_flags; /* Buffer status */
struct buf *av_forw; /* Driver work list link */
struct buf *av_back; /* Driver work list link */
size_t    b_bcount; /* # of bytes to transfer */
union {
  caddr_t   b_addr; /* Buffer’s virtual address */
} b_un;
daddr_t   b_blkno; /* Block number on device */
diskaddr_t b_lblkno; /* Expanded block number on device */
size_t    b_resid; /* # of bytes not transferred after error */
int       b_error; /* Expanded error field */
void      *b_private; /* "opaque" driver private area */
dev_t     b_edev; /* expanded dev field */
```
where:

`av_forw` and `av_back`  Pointers that the driver can use to manage a list of buffers by the driver. See “Asynchronous Data Transfers (Block Drivers)” on page 299 for a discussion of the `av_forw` and `av_back` pointers.

`b_bcount`  Specifies the number of bytes to be transferred by the device.

`b_un.b_addr`  The kernel virtual address of the data buffer. Only valid after `bp_mapin(9F)` call.

`b_blkno`  The starting 32-bit logical block number on the device for the data transfer, which is expressed in 512-byte `DEV_BSIZE` units. The driver should use either `b_blkno` or `b_lblkno` but not both.

`b_lblkno`  The starting 64-bit logical block number on the device for the data transfer, which is expressed in 512-byte `DEV_BSIZE` units. The driver should use either `b_blkno` or `b_lblkno` but not both.

`b_resid`  Set by the driver to indicate the number of bytes that were not transferred because of an error. See Example 16–7 for an example of setting `b_resid`. The `b_resid` member is overloaded. `b_resid` is also used by `disksort(9F)`.

`b_error`  Set to an error number by the driver when a transfer error occurs. `b_error` is set in conjunction with the `b_flags` `B_ERROR` bit. See the `Intro(9E)` man page for details about error values. Drivers should use `bioerror(9F)` rather than setting `b_error` directly.

`b_flags`  Flags with status and transfer attributes of the `buf` structure. If `B_READ` is set, the `buf` structure indicates a transfer from the device to memory. Otherwise, this structure indicates a transfer from memory to the device. If the driver encounters an error during data transfer, the driver should set the `B_ERROR` field in the `b_flags` member. In addition, the driver should provide a more specific error value in `b_error`. Drivers should use `bioerror(9F)` rather than setting `B_ERROR`.

`Caution` – Drivers should never clear `b_flags`.

`b_private`  For exclusive use by the driver to store driver-private data.

`b_edev`  Contains the device number of the device that was used in the transfer.
A buf structure pointer can be passed into the device driver's `strategy(9E)` routine. However, the data buffer referred to by `b_un.b_addr` is not necessarily mapped in the kernel's address space. Therefore, the driver cannot directly access the data. Most block-oriented devices have DMA capability and therefore do not need to access the data buffer directly. Instead, these devices use the DMA mapping routines to enable the device's DMA engine to do the data transfer. For details about using DMA, see Chapter 9, "Direct Memory Access (DMA)."

If a driver needs to access the data buffer directly, that driver must first map the buffer into the kernel's address space by using `bp_mapin(9F)`. `bp_mapout(9F)` should be used when the driver no longer needs to access the data directly.

Caution – `bp_mapout(9F)` should only be called on buffers that have been allocated and are owned by the device driver. `bp_mapout()` must not be called on buffers that are passed to the driver through the `strategy(9E)` entry point, such as a file system. `bp_mapin(9F)` does not keep a reference count. `bp_mapout(9F)` removes any kernel mapping on which a layer over the device driver might rely.

**Synchronous Data Transfers (Block Drivers)**

This section presents a simple method for performing synchronous I/O transfers. This method assumes that the hardware is a simple disk device that can transfer only one data buffer at a time by using DMA. Another assumption is that the disk can be spun up and spun down by software command. The device driver's `strategy(9E)` routine waits for the current request to be completed before accepting a new request. The device interrupts when the transfer is complete. The device also interrupts if an error occurs.

The steps for performing a synchronous data transfer for a block driver are as follows:

1. Check for invalid `buf(9S)` requests.
   - Check the `buf(9S)` structure that is passed to `strategy(9E)` for validity. All drivers should check the following conditions:
     - The request begins at a valid block. The driver converts the `b_blkno` field to the correct device offset and then determines whether the offset is valid for the device.
     - The request does not go beyond the last block on the device.
     - Device-specific requirements are met.
   - If an error is encountered, the driver should indicate the appropriate error with `bioerror(9F)`. The driver should then complete the request by calling `biodone(9F)`. `biodone()` notifies the caller of `strategy(9E)` that the transfer is complete. In this case, the transfer has stopped because of an error.
2. Check whether the device is busy.

Synchronous data transfers allow single-threaded access to the device. The device driver enforces this access in two ways:

- The driver maintains a busy flag that is guarded by a mutex.
- The driver waits on a condition variable with `cv_wait(9F)`, when the device is busy.

If the device is busy, the thread waits until the interrupt handler indicates that the device is not longer busy. The available status can be indicated by either the `cv_broadcast(9F)` or the `cv_signal(9F)` function. See Chapter 3, “Multithreading,” for details on condition variables.

When the device is no longer busy, the `strategy(9E)` routine marks the device as available. `strategy()` then prepares the buffer and the device for the transfer.

3. Set up the buffer for DMA.

Prepare the data buffer for a DMA transfer by using `ddi_dma_alloc_handle(9F)` to allocate a DMA handle. Use `ddi_dma_buf_bind_handle(9F)` to bind the data buffer to the handle. For information on setting up DMA resources and related data structures, see Chapter 9, “Direct Memory Access (DMA).”

4. Begin the transfer.

At this point, a pointer to the `buf(9S)` structure is saved in the state structure of the device. The interrupt routine can then complete the transfer by calling `biodone(9F)`.

The device driver then accesses device registers to initiate a data transfer. In most cases, the driver should protect the device registers from other threads by using mutexes. In this case, because `strategy(9E)` is single-threaded, guarding the device registers is not necessary. See Chapter 3, “Multithreading,” for details about data locks.

When the executing thread has started the device’s DMA engine, the driver can return execution control to the calling routine, as follows:

```c
static int
xxstrategy(struct buf *bp)
{
    struct xxstate *xsp;
    struct device_reg *regp;
    minor_t instance;
    ddi_dma_cookie_t cookie;
    instance = getminor(bp->b_edev);
    xsp = ddi_get_soft_state(statep, instance);
    if (xsp == NULL) {
        bioerror(bp, ENXIO);
        biodone(bp);
        return (0);
    }
    /* validate the transfer request */
    if ((bp->b_blkno >= xsp->Nblocks) || (bp->b_blkno < 0)) {
        bioerror(bp, EINVAL);
        biodone(bp);
        return (0);
    }
    /* validate the transfer request */
    if (/* conditions */) {
        bioerror(bp, EINVAL);
        biodone(bp);
        return (0);
    }
    /* prepare the buffer */
    if (/* conditions */) {
        bioerror(bp, EINVAL);
        biodone(bp);
        return (0);
    }
    /* start the transfer */
    if (/* conditions */) {
        bioerror(bp, EINVAL);
        biodone(bp);
        return (0);
    }
    /* complete the transfer */
    if (/* conditions */) {
        bioerror(bp, EINVAL);
        biodone(bp);
        return (0);
    }
    /* return control to the calling routine */
    return (0);
}
```
mutex_enter(&xsp->mu);
while (xsp->busy) {
    cv_wait(&xsp->cv, &xsp->mu);
}
mutex_exit(&xsp->mu);

*/
* If the device has power manageable components,
* mark the device busy with pm_busy_components(9F),
* and then ensure that the device
* is powered up by calling pm_raise_power(9F).
*
* Set up DMA resources with ddi_dma_alloc_handle(9F) and
* ddi_dma_buf_bind_handle(9F).
*
/*
 xsp->bp = bp;
 regp = xsp->regp;
 ddi_put32(xsp->data_access_handle, &regp->dma_addr,
          cookie.dmac_address);
 ddi_put32(xsp->data_access_handle, &regp->dma_size,
          (uint32_t)cookie.dmac_size);
 ddi_put8(xsp->data_access_handle, &regp->csr,
         ENABLE_INTERRUPTS | START_TRANSFER);
return (0);
}

5. Handle the interrupting device.

When the device finishes the data transfer, the device generates an interrupt, which
eventually results in the driver’s interrupt routine being called. Most drivers specify the state
structure of the device as the argument to the interrupt routine when registering interrupts.
See the ddi_add_intr(9F) man page and “Registering Interrupts” on page 128. The
interrupt routine can then access the buf(9S) structure being transferred, plus any other
information that is available from the state structure.

The interrupt handler should check the device’s status register to determine whether the
transfer completed without error. If an error occurred, the handler should indicate the
appropriate error with bioerror(9F). The handler should also clear the pending interrupt
for the device and then complete the transfer by calling biodone(9F).

As the final task, the handler clears the busy flag. The handler then calls cv_signal(9F) or
cv_broadcast(9F) on the condition variable, signaling that the device is no longer busy.
This notification enables other threads waiting for the device in strategy(9E) to proceed
with the next data transfer.

The following example shows a synchronous interrupt routine.

EXAMPLE 16-4  Synchronous Interrupt Routine for Block Drivers

static u_int
xxintr(caddr_t arg)
Synchronous Interrupt Routine for Block Drivers (Continued)

```c
{
    struct xxstate *xsp = (struct xxstate *)arg;
    struct buf *bp;
    uint8_t status;
    mutex_enter(&xsp->mu);
    status = ddi_get8(xsp->data_access_handle, &xsp->regp->csr);
    if (!(status & INTERRUPTING)) {
        mutex_exit(&xsp->mu);
        return (DDI_INTR_UNCLAIMED);
    }
    /* Get the buf responsible for this interrupt */
    bp = xsp->bp;
    xsp->bp = NULL;
    /* This example is for a simple device which either
     * succeeds or fails the data transfer, indicated in the
     * command/status register.
     */
    if (status & DEVICE_ERROR) {
        /* failure */
        bp->b_resid = bp->b_bcount;
        bioerror(bp, EIO);
    } else {
        /* success */
        bp->b_resid = 0;
    }
    ddi_put8(xsp->data_access_handle, &xsp->regp->csr,
             CLEAR_INTERRUPT);
    /* The transfer has finished, successfully or not */
    biodone(bp);
    /*
     * If the device has power manageable components that were
     * marked busy in strategy(9F), mark them idle now with
     * pm_idle_component(9F)
     * Release any resources used in the transfer, such as DMA
     * resources ddi_dma_unbind_handle(9F) and
     * ddi_dma_free_handle(9F).
     *
     * Let the next I/O thread have access to the device.
     */
    xsp->busy = 0;
    cv_signal(&xsp->cv);
    mutex_exit(&xsp->mu);
    return (DDI_INTRCLAIMED);
}
```
Asynchronous Data Transfers (Block Drivers)

This section presents a method for performing asynchronous I/O transfers. The driver queues the I/O requests and then returns control to the caller. Again, the assumption is that the hardware is a simple disk device that allows one transfer at a time. The device interrupts when a data transfer has completed. An interrupt also takes place if an error occurs. The basic steps for performing asynchronous data transfers are:

1. Check for invalid buf(9S) requests.
2. Enqueue the request.
3. Start the first transfer.
4. Handle the interrupting device.

Checking for Invalid buf Requests

As in the synchronous case, the device driver should check the buf(9S) structure passed to strategy(9E) for validity. See “Synchronous Data Transfers (Block Drivers)” on page 295 for more details.

Enqueuing the Request

Unlike synchronous data transfers, a driver does not wait for an asynchronous request to complete. Instead, the driver adds the request to a queue. The head of the queue can be the current transfer. The head of the queue can also be a separate field in the state structure for holding the active request, as in Example 16–5.

If the queue is initially empty, then the hardware is not busy and strategy(9E) starts the transfer before returning. Otherwise, if a transfer completes with a non-empty queue, the interrupt routine begins a new transfer. Example 16–5 places the decision of whether to start a new transfer into a separate routine for convenience.

The driver can use the av_forw and the av_back members of the buf(9S) structure to manage a list of transfer requests. A single pointer can be used to manage a singly linked list, or both pointers can be used together to build a doubly linked list. The device hardware specification specifies which type of list management, such as insertion policies, is used to optimize the performance of the device. The transfer list is a per-device list, so the head and tail of the list are stored in the state structure.

The following example provides multiple threads with access to the driver shared data, such as the transfer list. You must identify the shared data and must protect the data with a mutex. See Chapter 3, ”Multithreading,” for more details about mutex locks.
EXAMPLE 16-5  Enqueuing Data Transfer Requests for Block Drivers

static int
xxstrategy(struct buf *bp)
{
    struct xxstate *xsp;
    minor_t instance;
    instance = getminor(bp->b_edev);
    xsp = ddi_get_soft_state(statep, instance);
    /* ... */
    /* validate transfer request */
    /* ... */
    /* Add the request to the end of the queue. Depending on the device, a sorting
    * algorithm, such as disksort(9F) can be used if it improves the
    * performance of the device.
    */
    mutex_enter(&xsp->mu);
    bp->av_forw = NULL;
    if (xsp->list_head) {
        /* Non-empty transfer list */
        xsp->list_tail->av_forw = bp;
        xsp->list_tail = bp;
    } else {
        /* Empty Transfer list */
        xsp->list_head = bp;
        xsp->list_tail = bp;
    }
    mutex_exit(&xsp->mu);
    /* Start the transfer if possible */
    (void) xxstart((caddr_t)xsp);
    return (0);
}

Starting the First Transfer

Device drivers that implement queuing usually have a start() routine. start() dequeues the
next request and starts the data transfer to or from the device. In this example, start() processes all requests regardless of the state of the device, whether busy or free.

Note – start() must be written to be called from any context. start() can be called by both the strategy routine in kernel context and the interrupt routine in interrupt context.

start() is called by strategy(9E) every time strategy() queues a request so that an idle
device can be started. If the device is busy, start() returns immediately.

start() is also called by the interrupt handler before the handler returns from a claimed
interrupt so that a nonempty queue can be serviced. If the queue is empty, start() returns immediately.
Because `start()` is a private driver routine, `start()` can take any arguments and can return any type. The following code sample is written to be used as a DMA callback, although that portion is not shown. Accordingly, the example must take a `caddr_t` as an argument and return an `int`. See "Handling Resource Allocation Failures" on page 167 for more information about DMA callback routines.

**EXAMPLE 16-6  Starting the First Data Request for a Block Driver**

```c
static int
xxstart(caddr_t arg)
{
    struct xxstate *xsp = (struct xxstate *)arg;
    struct buf *bp;

    mutex_enter(&xsp->mu);
    /*
    * If there is nothing more to do, or the device is
    * busy, return.
    */
    if (xsp->list_head == NULL || xsp->busy) {
        mutex_exit(&xsp->mu);
        return (0);
    }
    xsp->busy = 1;
    /* Get the first buffer off the transfer list */
    bp = xsp->list_head;
    /* Update the head and tail pointer */
    xsp->list_head = xsp->list_head->av_forw;
    if (xsp->list_head == NULL)
        xsp->list_tail = NULL;
    bp->av_forw = NULL;
    mutex_exit(&xsp->mu);
    /*
    * If the device has power manageable components,
    * mark the device busy with pm_busy_components(9F),
    * and then ensure that the device
    * is powered up by calling pm_raise_power(9F).
    *
    * Set up DMA resources with ddi_dma_alloc_handle(9F) and
    * ddi_dma_buf_bind_handle(9F).
    */
    xsp->bp = bp;
    ddi_put32(xsp->data_access_handle, &xsp->regp->dma_addr,
              cookie.dmac_address);
    ddi_put32(xsp->data_access_handle, &xsp->regp->dma_size,
              (uint32_t)cookie.dmac_size);
    ddi_put8(xsp->data_access_handle, &xsp->regp->csr,
              ENABLE_INTERRUPTS | START_TRANSFER);
    return (0);
}
```
Handling the Interrupting Device

The interrupt routine is similar to the asynchronous version, with the addition of the call to `start()` and the removal of the call to `cv_signal(9F).

**EXAMPLE 16–7  Block Driver Routine for Asynchronous Interrupts**

```c
static u_int
xxintr(caddr_t arg)
{
    struct xxstate *xsp = (struct xxstate *)arg;
    struct buf *bp;
    uint8_t status;
    mutex_enter(&xsp->mu);
    status = ddi_get8(xsp->data_access_handle, &xsp->regp->csr);
    if (!(status & INTERRUPTING)) {
        mutex_exit(&xsp->mu);
        return (DDI_INTR_UNCLAIMED);
    }
    /* Get the buf responsible for this interrupt */
    bp = xsp->bp;
    xsp->bp = NULL;
    /* This example is for a simple device which either 
    * succeeds or fails the data transfer, indicated in the 
    * command/status register. */
    if (status & DEVICE_ERROR) {
        /* failure */
        bp->b_resid = bp->b_bcount;
        bioerror(bp, EIO);
    } else {
        /* success */
        bp->b_resid = 0;
    }
    ddi_put8(xsp->data_access_handle, &xsp->regp->csr,
             CLEAR_INTERRUPT);
    /* The transfer has finished, successfully or not */
    biodone(bp);
    /*
    * If the device has power manageable components that were 
    * marked busy in strategy(9F), mark them idle now with 
    * pm_idle_component(9F)
    * Release any resources used in the transfer, such as DMA 
    * resources (ddi_dma_unbind_handle(9F) and 
    * ddi_dma_free_handle(9F)).
    * Let the next I/O thread have access to the device.
    */
    xsp->busy = 0;
    mutex_exit(&xsp->mu);
    (void) xxstart((caddr_t)xsp);
    return (DDI_INTRCLAIMED);
}
```
dump() and print() Entry Points

This section discusses the dump(9E) and print(9E) entry points.

**dump() Entry Point (Block Drivers)**

The dump(9E) entry point is used to copy a portion of virtual address space directly to the specified device in the case of a system failure. dump() is also used to copy the state of the kernel out to disk during a checkpoint operation. See the cpr(7) and dump(9E) man pages for more information. The entry point must be capable of performing this operation without the use of interrupts, because interrupts are disabled during the checkpoint operation.

```c
int dump(dev_t dev, caddr_t addr, daddr_t blkno, int nblk)
```

where:

- `dev` Device number of the device to receive the dump.
- `addr` Base kernel virtual address at which to start the dump.
- `blkno` Block at which the dump is to start.
- `nblk` Number of blocks to dump.

The dump depends upon the existing driver working properly.

**print() Entry Point (Block Drivers)**

```c
int print(dev_t dev, char *str)
```

The print(9E) entry point is called by the system to display a message about an exception that has been detected. print(9E) should call cmn_err(9F) to post the message to the console on behalf of the system. The following example demonstrates a typical print() entry point.

```c
static int
xxprint(dev_t dev, char *str)
{
    cmn_err(CE_CONT, "xx: %s\n", str);
    return (0);
}
```
Disk Device Drivers

Disk devices represent an important class of block device drivers.

Disk ioctls

Oracle Solaris disk drivers need to support a minimum set of ioctl commands specific to Oracle Solaris disk drivers. These I/O controls are specified in the dkioc(7) manual page. Disk I/O controls transfer disk information to or from the device driver. An Oracle Solaris disk device is supported by disk utility commands such as format(1M) and newfs(1M). The mandatory Sun disk I/O controls are as follows:

- **DKIOCINFO**: Returns information that describes the disk controller
- **DKIOCGAPART**: Returns a disk's partition map
- **DKIOCSAPART**: Sets a disk's partition map
- **DKIOCGGEOEM**: Returns a disk's geometry
- **DKIOCSGEOEM**: Sets a disk's geometry
- **DKIOCGVTOC**: Returns a disk's Volume Table of Contents
- **DKIOCSVTOC**: Sets a disk's Volume Table of Contents

Disk Performance

The Oracle Solaris DDI/DKI provides facilities to optimize I/O transfers for improved file system performance. A mechanism manages the list of I/O requests so as to optimize disk access for a file system. See "Asynchronous Data Transfers (Block Drivers)" on page 299 for a description of enqueuing an I/O request.

The diskhd structure is used to manage a linked list of I/O requests.

```c
struct diskhd {
    long b_flags; /* not used, needed for consistency*/
    struct buf *b_forw, *b_back; /* queue of unit queues */
    struct buf *av_forw, *av_back; /* queue of bufs for this unit */
    long b_bcount; /* active flag */
};
```

The diskhd data structure has two buf pointers that the driver can manipulate. The av_forw pointer points to the first active I/O request. The second pointer, av_back, points to the last active request on the list.
A pointer to this structure is passed as an argument to `disksort(9F)`, along with a pointer to the current buf structure being processed. The `disksort()` routine sorts the buf requests to optimize disk seek. The routine then inserts the buf pointer into the diskhd list. The `disksort()` program uses the value that is in `b_resid` of the buf structure as a sort key. The driver is responsible for setting this value. Most Sun disk drivers use the cylinder group as the sort key. This approach optimizes the file system read-ahead accesses.

When data has been added to the diskhd list, the device needs to transfer the data. If the device is not busy processing a request, the `xxstart()` routine pulls the first buf structure off the diskhd list and starts a transfer.

If the device is busy, the driver should return from the `xxstrategy()` entry point. When the hardware is done with the data transfer, an interrupt is generated. The driver's interrupt routine is then called to service the device. After servicing the interrupt, the driver can then call the `start()` routine to process the next buf structure in the diskhd list.
SCSI Target Drivers

The Oracle Solaris DDI/DKI divides the software interface to SCSI devices into two major parts: target drivers and host bus adapter (HBA) drivers. Target refers to a driver for a device on a SCSI bus, such as a disk or a tape drive. Host bus adapter refers to the driver for the SCSI controller on the host machine. SCSA defines the interface between these two components. This chapter discusses target drivers only. See Chapter 18, “SCSI Host Bus Adapter Drivers,” for information on host bus adapter drivers.

Note – The terms “host bus adapter” and “HBA” are equivalent to “host adapter,” which is defined in SCSI specifications.

This chapter provides information on the following subjects:

- “Introduction to Target Drivers” on page 307
- “Sun Common SCSI Architecture Overview” on page 308
- “Hardware Configuration File” on page 311
- “Declarations and Data Structures” on page 311
- “Autoconfiguration for SCSI Target Drivers” on page 315
- “Resource Allocation” on page 320
- “Building and Transporting a Command” on page 323
- “SCSI Options” on page 329

Introduction to Target Drivers

Target drivers can be either character or block device drivers, depending on the device. Drivers for tape drives are usually character device drivers, while disks are handled by block device drivers. This chapter describes how to write a SCSI target driver. The chapter discusses the additional requirements that SCSA places on block and character drivers for SCSI target devices.
The following reference documents provide supplemental information needed by the designers of target drivers and host bus adapter drivers.


Refer also to the SCSI command specification for the target device, provided by the hardware vendor.

**Sun Common SCSI Architecture Overview**

The Sun Common SCSI Architecture (SCSA) is the Solaris DDI/DKI programming interface for the transmission of SCSI commands from a target driver to a host bus adapter driver. This interface is independent of the type of host bus adapter hardware, the platform, the processor architecture, and the SCSI command being transported across the interface.

Conforming to the SCSA enables the target driver to pass SCSI commands to target devices without knowledge of the hardware implementation of the host bus adapter.

The SCSA conceptually separates building the SCSI command from transporting the command with data across the SCSI bus. The architecture defines the software interface between high-level and low-level software components. The higher level software component consists of one or more SCSI target drivers, which translate I/O requests into SCSI commands appropriate for the peripheral device. The following example illustrates the SCSI architecture.
The lower-level software component consists of a SCSA interface layer and one or more host bus adapter drivers. The target driver is responsible for the generation of the proper SCSI commands required to execute the desired function and for processing the results.

**General Flow of Control**

Assuming no transport errors occur, the following steps describe the general flow of control for a read or write request.

1. The target driver’s `read(9E)` or `write(9E)` entry point is invoked. `physio(9F)` is used to lock down memory, prepare a `buf` structure, and call the strategy routine.

2. The target driver’s `strategy(9E)` routine checks the request. `strategy()` then allocates a `scsi_pkt(9S)` by using `scsi_init_pkt(9F)`. The target driver initializes the packet and sets the SCSI command descriptor block (CDB) using the `scsi_setup_cdb(9F)` function. The target driver also specifies a timeout. Then, the driver provides a pointer to a callback function. The callback function is called by the host bus adapter driver on completion of the command. The `buf(9S)` pointer should be saved in the SCSI packet’s target-private space.

3. The target driver submits the packet to the host bus adapter driver by using `scsi_transport(9F)`. The target driver is then free to accept other requests. The target driver should not access the packet while the packet is in transport. If either the host bus adapter driver or the target supports queueing, new requests can be submitted while the packet is in transport.

4. As soon as the SCSI bus is free and the target not busy, the host bus adapter driver selects the target and passes the CDB. The target driver executes the command. The target then performs the requested data transfers.
5. After the target sends completion status and the command completes, the host bus adapter driver notifies the target driver. To perform the notification, the host calls the completion function that was specified in the SCSI packet. At this time the host bus adapter driver is no longer responsible for the packet, and the target driver has regained ownership of the packet.

6. The SCSI packet’s completion routine analyzes the returned information. The completion routine then determines whether the SCSI operation was successful. If a failure has occurred, the target driver retries the command by calling `scsi_transport(9F)` again. If the host bus adapter driver does not support auto request sense, the target driver must submit a request sense packet to retrieve the sense data in the event of a check condition.

7. After successful completion or if the command cannot be retried, the target driver calls `scsi_destroy_pkt(9F)`. `scsi_destroy_pkt()` synchronizes the data. `scsi_destroy_pkt()` then frees the packet. If the target driver needs to access the data before freeing the packet, `scsi_sync_pkt(9F)` is called.

8. Finally, the target driver notifies the requesting application that the read or write transaction is complete. This notification is made by returning from the `read(9E)` entry point in the driver for character devices. Otherwise, notification is made indirectly through `biodone(9F)`.

SCSA allows the execution of many of such operations, both overlapped and queued, at various points in the process. The model places the management of system resources on the host bus adapter driver. The software interface enables the execution of target driver functions on host bus adapter drivers by using SCSI bus adapters of varying degrees of sophistication.

**SCSA Functions**

SCSA defines functions to manage the allocation and freeing of resources, the sensing and setting of control states, and the transport of SCSI commands. These functions are listed in the following table.

**TABLE 17–1 Standard SCSA Functions**

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>scsi_abort(9F)</code></td>
<td>Error handling</td>
</tr>
<tr>
<td><code>scsi_alloc_consistent_buf(9F)</code></td>
<td></td>
</tr>
<tr>
<td><code>scsi_destroy_pkt(9F)</code></td>
<td></td>
</tr>
<tr>
<td><code>scsi_dmafree(9F)</code></td>
<td></td>
</tr>
<tr>
<td><code>scsi_free_consistent_buf(9F)</code></td>
<td></td>
</tr>
<tr>
<td><code>scsi_ifgetcap(9F)</code></td>
<td>Transport information and control</td>
</tr>
</tbody>
</table>
TABLE 17–1 Standard SCSA Functions *(Continued)*

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>scsi_ifsetcap(9F)</td>
<td>Resource management</td>
</tr>
<tr>
<td>scsi_init_pkt(9F)</td>
<td>Polled I/O</td>
</tr>
<tr>
<td>scsi_poll(9F)</td>
<td>Probe functions</td>
</tr>
<tr>
<td>scsi_probe(9F)</td>
<td></td>
</tr>
<tr>
<td>scsi_reset(9F)</td>
<td></td>
</tr>
<tr>
<td>scsi_setup_cdb(9F)</td>
<td>CDB initialization function</td>
</tr>
<tr>
<td>scsi_sync_pkt(9F)</td>
<td>Command transport</td>
</tr>
<tr>
<td>scsi_transport(9F)</td>
<td></td>
</tr>
<tr>
<td>scsi_unprobe(9F)</td>
<td></td>
</tr>
</tbody>
</table>

Note – If your driver needs to work with a SCSI-1 device, use the makecom(9F).

Hardware Configuration File

Because SCSI devices are not self-identifying, a hardware configuration file is required for a target driver. See the driver.conf(4) and scsi_free_consistent_buf(9F) man pages for details. The following is a typical configuration file:

```
name="xx" class="scsi" target=2 lun=0;
```

The system reads the file during autoconfiguration. The system uses the class property to identify the driver’s possible parent. Then, the system attempts to attach the driver to any parent driver that is of class scsi. All host bus adapter drivers are of this class. Using the class property rather than the parent property is preferred. This approach enables any host bus adapter driver that finds the expected device at the specified target and lun IDs to attach to the target. The target driver is responsible for verifying the class in its probe(9E) routine.

Declarations and Data Structures

Target drivers must include the header file <sys/scsi/scsi.h>.

SCSI target drivers must use the following command to generate a binary module:

```
ld -r xx xx.o -N"misc/scsi"
```
**scsi_device Structure**

The host bus adapter driver allocates and initializes a `scsi_device(9S)` structure for the target driver before either the `probe(9E)` or `attach(9E)` routine is called. This structure stores information about each SCSI logical unit, including pointers to information areas that contain both generic and device-specific information. One `scsi_device(9S)` structure exists for each logical unit that is attached to the system. The target driver can retrieve a pointer to this structure by calling `ddi_get_driver_private(9F)`.

**Caution** – Because the host bus adapter driver uses the private field in the target device's `dev_info` structure, target drivers must not use `ddi_set_driver_private(9F)`.

The `scsi_device(9S)` structure contains the following fields:

```c
struct scsi_device {
    struct scsi_address sd_address; /* opaque address */
    dev_info_t *sd_dev; /* device node */
    kmutex_t sd_mutex;
    void *sd_reserved;
    struct scsi_inquiry *sd_inq;
    struct scsi_extended_sense *sd_sense;
    caddr_t sd_private;
};
```

where:

- **sd_address** Data structure that is passed to the routines for SCSI resource allocation.
- **sd_dev** Pointer to the target's `dev_info` structure.
- **sd_mutex** Mutex for use by the target driver. This mutex is initialized by the host bus adapter driver and can be used by the target driver as a per-device mutex. Do not hold this mutex across a call to `scsi_transport(9F)` or `scsi_poll(9F)`. See Chapter 3, "Multithreading," for more information on mutexes.
- **sd_inq** Pointer for the target device's SCSI inquiry data. The `scsi_probe(9F)` routine allocates a buffer, fills the buffer in with inquiry data, and attaches the buffer to this field.
- **sd_sense** Pointer to a buffer to contain SCSI request sense data from the device. The target driver must allocate and manage this buffer. See "attach() Entry Point (SCSI Target Drivers)" on page 317.
- **sd_private** Pointer field for use by the target driver. This field is commonly used to store a pointer to a private target driver state structure.
scsi_pkt Structure (Target Drivers)

The scsi_pkt structure contains the following fields:

```c
struct scsi_pkt {
    opaque_t pkt_ha_private; /* private data for host adapter */
    struct scsi_address pkt_address; /* destination packet is for */
    opaque_t pkt_private; /* private data for target driver */
    void (*pkt_comp)(struct scsi_pkt *); /* completion routine */
    uint_t pkt_flags; /* flags */
    int pkt_time; /* time allotted to complete command */
    uchar_t *pkt_scbp; /* pointer to status block */
    uchar_t *pkt_cdbp; /* pointer to command block */
    ssize_t pkt_resid; /* data bytes not transferred */
    uint_t pkt_state; /* state of command */
    uint_t pkt_statistics; /* statistics */
    uchar_t pkt_reason; /* reason completion called */
};
```

where:

- **pkt_address**  Target device’s address set by `scsi_init_pkt(9F)`.
- **pkt_private** Place to store private data for the target driver. `pkt_private` is commonly used to save the `buf(9S)` pointer for the command.
- **pkt_comp** Address of the completion routine. The host bus adapter driver calls this routine when the driver has transported the command. Transporting the command does not mean that the command succeeded. The target might have been busy. Another possibility is that the target might not have responded before the time out period elapsed. See the description for `pkt_time` field. The target driver must supply a valid value in this field. This value can be `NULL` if the driver does not want to be notified.

**Note**  Two different SCSI callback routines are provided. The `pkt_comp` field identifies a *completion callback* routine, which is called when the host bus adapter completes its processing. A *resource callback* routine is also available, which is called when currently unavailable resources are likely to be available. See the `scsi_init_pkt(9F)` man page.

- **pkt_flags** Provides additional control information, for example, to transport the command without disconnect privileges (FLAG_NODISCON) or to disable callbacks (FLAG_NOINTR). See the `scsi_pkt(9S)` man page for details.
- **pkt_time** Time out value in seconds. If the command is not completed within this time, the host bus adapter calls the completion routine with `pkt_reason` set to `CMD_TIMEOUT`. The target driver should set this field to longer than...
the maximum time the command might take. If the timeout is zero, no
timeout is requested. Timeout starts when the command is transmitted on
the SCSI bus.

(pkt_scbp) Pointer to the block for SCSI status completion. This field is filled in by the
host bus adapter driver.

(pkt_cdbp) Pointer to the SCSI command descriptor block, the actual command to be
sent to the target device. The host bus adapter driver does not interpret
this field. The target driver must fill the field in with a command that the
target device can process.

(pkt_resid) Residual of the operation. The pkt_resid field has two different uses
depending on how pkt_resid is used. When pkt_resid is used to allocate
DMA resources for a command scsi_init_pkt(9F), pkt_resid indicates
the number of unallocable bytes. DMA resources might not be allocated
due to DMA hardware scatter-gather or other device limitations. After
command transport, pkt_resid indicates the number of non-transferable
data bytes. The field is filled in by the host bus adapter driver before the
completion routine is called.

(pkt_state) Indicates the state of the command. The host bus adapter driver fills in
this field as the command progresses. One bit is set in this field for each of
the five following command states:
  ■ STATE_GOT_BUS – Acquired the bus
  ■ STATE_GOT_TARGET – Selected the target
  ■ STATE_SENT_CMD – Sent the command
  ■ STATE_XFERRED_DATA – Transferred data, if appropriate
  ■ STATE_GOT_STATUS – Received status from the device

(pkt_statistics) Contains transport-related statistics set by the host bus adapter driver.

(pkt_reason) Gives the reason the completion routine was called. The completion
routine decodes this field. The routine then takes the appropriate action.
If the command completes, that is, no transport errors occur, this field is
set to CMD_CMPLT. Other values in this field indicate an error. After a
command is completed, the target driver should examine the pkt_scbp
field for a check condition status. See the scsi_pkt(9S) man page for
more information.
Autoconfiguration for SCSI Target Drivers

SCSI target drivers must implement the standard autoconfiguration routines _init(9E), _fini(9E), and _info(9E). See “Loadable Driver Interfaces” on page 97 for more information.

The following routines are also required, but these routines must perform specific SCSI and SCsA processing:

- probe(9E)
- attach(9E)
- detach(9E)
- getinfo(9E)

probe ( ) Entry Point (SCSI Target Drivers)

SCSI target devices are not self-identifying, so target drivers must have a probe(9E) routine. This routine must determine whether the expected type of device is present and responding.

The general structure and the return codes of the probe(9E) routine are the same as the structure and return codes for other device drivers. SCSI target drivers must use the scsi探ue(9F) routine in their probe(9E) entry point. scsi探ue(9F) sends a SCSI inquiry command to the device and returns a code that indicates the result. If the SCSI inquiry command is successful, scsi探ue(9F) allocates a scsi_inquiry(9S) structure and fills the structure in with the device’s inquiry data. Upon return from scsi探ue(9F), the sd_inq field of the scsi_device(9S) structure points to this scsi_inquiry(9S) structure.

Because probe(9E) must be stateless, the target driver must call scsi_unprobe(9F) before probe(9E) returns, even if scsi探ue(9F) fails.

Example 17–1 shows a typical probe(9E) routine. The routine in the example retrieves the scsi_device(9S) structure from the private field of its dev_info structure. The routine also retrieves the device’s SCSI target and logical unit numbers for printing in messages. The probe(9E) routine then calls scsi探ue(9F) to verify that the expected device, a printer in this case, is present.

If successful, scsi探ue(9F) attaches the device’s SCSI inquiry data in a scsi_inquiry(9S) structure to the sd_inq field of the scsi_device(9S) structure. The driver can then determine whether the device type is a printer, which is reported in the inq_dtype field. If the device is a printer, the type is reported with scsi_log(9F), using scsi_dname(9F) to convert the device type into a string.

EXAMPLE 17–1  SCSI Target Driver probe(9E) Routine

static int
xxprobe(dev_info_t *dip)
EXAMPLE 17–1  SCSI Target Driver probe(9E) Routine  (Continued)

{
    struct scsi_device *sdp;
    int rval, target, lun;
    /*
     * Get a pointer to the scsi_device(9S) structure
     */
    sdp = (struct scsi_device *)ddi_get_driver_private(dip);

    target = sdp->sd_address.a_target;
    lun = sdp->sd_address.a_lun;
    /*
     * Call scsi_probe(9F) to send the Inquiry command. It will
     * fill in the sd_inq field of the scsi_device structure.
     */
    switch (scsi_probe(sdp, NULL_FUNC)) {
      case SCSIPROBE_FAILURE:
      case SCSIPROBE_NORESP:
      case SCSIPROBE_NOMEM:
        /*
         * In these cases, device might be powered off,
         * in which case we might be able to successfully
         * probe it at some future time - referred to
         * as 'deferred attach'.
         */
        rval = DDI_PROBE_PARTIAL;
        break;
      case SCSIPROBE_NONCSE:
        default:
        /*
         * Device isn't of the type we can deal with,
         * and/or it will never be usable.
         */
        rval = DDI_PROBE_FAILURE;
        break;
      case SCSIPROBE_EXISTS:
        /*
         * There is a device at the target/lun address. Check
         * inq dtype to make sure that it is the right device
         * type. See scsi_inquiry(9S)for possible device types.
         */
        switch (sdp->sd_inq->inq_dtype) {
          case DTYPE_PRINTER:
            scsi_log(sdp, "xx", SCSI_DEBUG,
                "found %s device at target%d, lun%d\n",
                scsi_dname((int)sdp->sd_inq->inq_dtype),
                target, lun);
            rval = DDI_PROBE_SUCCESS;
            break;
          case DTYPE_NOTPRESENT:
            default:
            rval = DDI_PROBE_FAILURE;
            break;
        }
    }
    scsi_unprobe(sdp);
A more thorough probe routine could check `scsi_inquiry` to make sure that the device is of the type expected by a particular driver.

### attach() Entry Point (SCSI Target Drivers)

After the `probe` routine has verified that the expected device is present, `attach` is called. `attach()` performs these tasks:

- Allocates and initializes any per-instance data.
- Creates minor device node information.
- Restores the hardware state of a device after a suspension of the device or the system. See "attach() Entry Point" on page 104 for details.

A SCSI target driver needs to call `scsi_probe` again to retrieve the device's inquiry data. The driver must also create a SCSI request sense packet. If the attach is successful, the attach() function should not call `scsi_unprobe`.

Three routines are used to create the request sense packet: `scsi_alloc_consistent_buf`, `scsi_init_pkt`, and `scsi_setup_cdb`. `scsi_alloc_consistent_buf` allocates a buffer that is suitable for consistent DMA. `scsi_alloc_consistent_buf()` then returns a pointer to a `buf` structure. The advantage of a consistent buffer is that no explicit synchronization of the data is required. In other words, the target driver can access the data after the callback. The `sd_sense` element of the device's `scsi_device` structure must be initialized with the address of the sense buffer. `scsi_init_pkt()` creates and partially initializes a `scsi_pkt` structure. `scsi_setup_cdb()` creates a SCSI command descriptor block, in this case by creating a SCSI request sense command.

Note that a SCSI device is not self-identifying and does not have a reg property. As a result, the driver must set the `pm-hardware-state` property. Setting `pm-hardware-state` informs the framework that this device needs to be suspended and then resumed.

The following example shows the SCSI target driver's `attach()` routine.

```c
EXAMPLE 17–2  SCSI Target Driver attach(9E) Routine

static int
xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    struct xxstate *xsp;
    struct scsi_pkt *rqpkt = NULL;
    return (rval);
}
```
struct scsi_device  *sdp;
struct buf         *bp = NULL;
int                instance;
instance = ddi_get_instance(dip);
switch (cmd) {
    case DDI_ATTACH:
        break;
    case DDI_RESUME:
        /* For information, see the "Directory Memory Access (DMA)" */
        /* chapter in this book. */
        default:
            return (DDI_FAILURE);
    }
/*
* Allocate a state structure and initialize it.
*/
xsp = ddi_get_soft_state(statep, instance);
sdp = (struct scsi_device *)ddi_get_driver_private(dip);
/*
* Cross-link the state and scsi_device(9S) structures.
*/
if (sdp->sd_private = (caddr_t)xsp;
    xsp->sdp = sdp;
/*
* Call scsi_probe(9F) again to get and validate inquiry data.
* Allocate a request sense buffer. The buf(9S) structure
* is set to NULL to tell the routine to allocate a new one.
* The callback function is set to NULL_FUNC to tell the
* routine to return failure immediately if no
* resources are available.
*/
    bp = scsi_alloc_consistent_buf(&sdp->sd_address, NULL,
        SENSE_LENGTH, B_READ, NULL_FUNC, NULL);
    if (bp == NULL)
        goto failed;
/*
* Create a Request Sense scsi_pkt(9S) structure.
*/
rqpkt = scsi_init_pkt(&sdp->sd_address, NULL, bp,
    CDB_GROUP0, 1, 0, PKT_CONSISTENT, NULL_FUNC, NULL);
    if (rqpkt == NULL)
        goto failed;
/*
* scsi_alloc_consistent_buf(9F) returned a buf(9S) structure.
* The actual buffer address is in b.un.b_addr.
*/
sdp->sd_sense = (struct scsi_extended_sense *)bp->b_un.b_addr;
/*
* Create a Group0 CDB for the Request Sense command
*/
if (scsi_setup_cdb((union scsi_cdb *)rqpkt->pkt_cdbp,
    SCMD_REQUEST_SENSE, 0, SENSE__LENGTH, 0) == 0)
    goto failed;
/*
* Fill in the rest of the scsi_pkt structure.
*/
* `xxcallback()` is the private command completion routine.
*/
rqpkt->pkt_comp = xxcallback;
rqpkt->pkt_time = 30; /* 30 second command timeout */
rqpkt->pkt_flags |= FLAG_SENSING;
xsp->rqs = rqpkt;
xsp->rqsbuf = bp;
;/*
 * Create minor nodes, report device, and do any other initialization. */
 * Since the device does not have the 'reg' property,
 * cpr will not call its DDI_SUSPEND/DDI_RESUME entries.
 * The following code is to tell cpr that this device
 * needs to be suspended and resumed.
 */
(void) ddi_prop_update_string(device, dip,
 "pm-hardware-state", "needs-suspend-resume");
xsp->open = 0;
return (DDI_SUCCESS);
failed:
if (bp)
    scsi_free_consistent_buf(bp);
if (rqpkt)
    scsi_destroy_pkt(rqpkt);
sdp->sd_private = (caddr_t)NULL;
sdp->sd_sense = NULL;
scsi_unprobe(sdp);
/* Free any other resources, such as the state structure. */
return (DDI_FAILURE);
}

detach() Entry Point (SCSI Target Drivers)

The `detach()` entry point is the inverse of `attach()`. `detach()` must free all resources that were allocated in `attach()`. If successful, the `detach` should call `scsi_unprobe()`. The following example shows a target driver `detach()` routine.

```c
static int
xxdetach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
    struct xxstate *xsp;
    switch (cmd) {
    case DDI_DETACH:
        /*
         * Normal `detach()` operations, such as getting a
         * pointer to the state structure
         */
        scsi_free_consistent_buf(xsp->rqsbuf);
        scsi_destroy_pkt(xsp->rqs);
        xsp->sdp->sd_private = (caddr_t)NULL;
```
EXAMPLE 17–3  SCSI Target Driver detach(9E) Routine  (Continued)

```c
  xsp->sdp->sd_sense = NULL;
  scsi_unprobe(xsp->sdp);
  /*
   * Remove minor nodes.
   * Free resources, such as the state structure and properties.
   */
  return (DDI_SUCCESS);
```

case DDI_SUSPEND:
  /* For information, see the "Directory Memory Access (DMA)" */
  /* chapter in this book. */
default:
  return (DDI_FAILURE);
}

getinfo() Entry Point (SCSI Target Drivers)

The getinfo(9E) routine for SCSI target drivers is much the same as for other drivers (see "getinfo() Entry Point" on page 110 for more information on DDI_INFO_DEVT2INSTANCE case). However, in the DDI_INFO_DEVT2DEVINFO case of the getinfo() routine, the target driver must return a pointer to its dev_info node. This pointer can be saved in the driver state structure or can be retrieved from the sd_dev field of the scsi_device(9S) structure. The following example shows an alternative SCSI target driver getinfo() code fragment.

EXAMPLE 17–4  Alternative SCSI Target Driver getinfo() Code Fragment

```c
  case DDI_INFO_DEVT2DEVINFO:
    dev = (dev_t)arg;
    instance = getminor(dev);
    xsp = ddi_get_soft_state(statep, instance);
    if (xsp == NULL)
      return (DDI_FAILURE);
    *result = (void *)xsp->sdp->sd_dev;
    return (DDI_SUCCESS);
```

Resource Allocation

To send a SCSI command to the device, the target driver must create and initialize a `scsi_pkt(9S)` structure. This structure must then be passed to the host bus adapter driver.

scsi_init_pkt() Function

The `scsi_init_pkt(9F)` routine allocates and zeroes a `scsi_pkt(9S)` structure. `scsi_init_pkt()` also sets pointers to `pkt_private`, `*pkt_scbp`, and `*pkt_cdbp`. Additionally,
scsi_init_pkt() provides a callback mechanism to handle the case where resources are not available. This function has the following syntax:

```c
struct scsi_pkt *scsi_init_pkt(struct scsi_address *ap,
    struct scsi_pkt *pktp, struct buf *bp, int cmdlen,
    int statuslen, int privatelen, int flags,
    int (*callback)(caddr_t), caddr_t arg)
```

where:

- **ap**
  Pointer to a scsi_address structure. ap is the sd_address field of the device's scsi_device(9S) structure.

- **pktp**
  Pointer to the scsi_pkt(9S) structure to be initialized. If this pointer is set to NULL, a new packet is allocated.

- **bp**
  Pointer to a buf(9S) structure. If this pointer is not null and has a valid byte count, DMA resources are allocated.

- **cmdlen**
  Length of the SCSI command descriptor block in bytes.

- **statuslen**
  Required length of the SCSI status completion block in bytes.

- **privatelen**
  Number of bytes to allocate for the pkt_private field.

- **flags**
  Set of flags:
  - PKT_CONSISTENT – This bit must be set if the DMA buffer was allocated using scsi_alloc_consistent_buf(9F). In this case, the host bus adapter driver guarantees that the data transfer is properly synchronized before performing the target driver’s command completion callback.
  - PKT_DMA_PARTIAL – This bit can be set if the driver accepts a partial DMA mapping. If set, scsi_init_pkt(9F) allocates DMA resources with the DDI_DMA_PARTIAL flag set. The pkt_resid field of the scsi_pkt(9S) structure can be returned with a nonzero residual. A nonzero value indicates the number of bytes for which scsi_init_pkt(9F) was unable to allocate DMA resources.

- **callback**
  Specifies the action to take if resources are not available. If set to NULL_FUNC, scsi_init_pkt(9F) returns the value NULL immediately. If set to SLEEP_FUNC, scsi_init_pkt() does not return until resources are available. Any other valid kernel address is interpreted as the address of a function to be called when resources are likely to be available.

- **arg**
  Parameter to pass to the callback function.

The scsi_init_pkt() routine synchronizes the data prior to transport. If the driver needs to access the data after transport, the driver should call scsi_sync_pkt(9F) to flush any intermediate caches. The scsi_sync_pkt() routine can be used to synchronize any cached data.
**scsi_sync_pkt() Function**

If the target driver needs to resubmit the packet after changing the data, `scsi_sync_pkt(9F)` must be called before calling `scsi_transport(9F)`. However, if the target driver does not need to access the data, `scsi_sync_pkt()` does not need to be called after the transport.

**scsi_destroy_pkt() Function**

The `scsi_destroy_pkt(9F)` routine synchronizes any remaining cached data that is associated with the packet, if necessary. The routine then frees the packet and associated command, status, and target driver-private data areas. This routine should be called in the command completion routine.

**scsi_alloc_consistent_buf() Function**

For most I/O requests, the data buffer passed to the driver entry points is not accessed directly by the driver. The buffer is just passed on to `scsi_init_pkt(9F)`. If a driver sends SCSI commands that operate on buffers that the driver itself examines, the buffers should be DMA consistent. The SCSI request sense command is a good example. The `scsi_alloc_consistent_buf(9F)` routine allocates a `buf(9S)` structure and a data buffer that is suitable for DMA-consistent operations. The HBA performs any necessary synchronization of the buffer before performing the command completion callback.

*Note* — `scsi_alloc_consistent_buf(9F)` uses scarce system resources. Thus, you should use `scsi_alloc_consistent_buf()` sparingly.

**scsi_free_consistent_buf() Function**

`scsi_free_consistent_buf(9F)` releases a `buf(9S)` structure and the associated data buffer allocated with `scsi_alloc_consistent_buf(9F)`. See “attach() Entry Point (SCSI Target Drivers)” on page 317 and “detach() Entry Point (SCSI Target Drivers)” on page 319 for examples.
Building and Transporting a Command

The host bus adapter driver is responsible for transmitting the command to the device. Furthermore, the driver is responsible for handling the low-level SCSI protocol. The `scsi_transport(9F)` routine hands a packet to the host bus adapter driver for transmission. The target driver has the responsibility to create a valid `scsi_pkt(9S)` structure.

Building a Command

The routine `scsi_init_pkt(9F)` allocates space for a SCSI CDB, allocates DMA resources if necessary, and sets the `pkt_flags` field, as shown in this example:

```c
pkt = scsi_init_pkt(&sdp->sd_address, NULL, bp, 
    CDB_GROUP0, 1, 0, 0, SLEEP_FUNC, NULL);
```

This example creates a new packet along with allocating DMA resources as specified in the passed `buf(9S)` structure pointer. A SCSI CDB is allocated for a Group 0 (6-byte) command. The `pkt_flags` field is set to zero, but no space is allocated for the `pkt_private` field. This call to `scsi_init_pkt(9F)`, because of the `SLEEP_FUNC` parameter, waits indefinitely for resources if no resources are currently available.

The next step is to initialize the SCSI CDB, using the `scsi_setup_cdb(9F)` function:

```c
if (scsi_setup_cdb((union scsi_cdb *)pkt->pkt_cdbp, 
    SCMD_READ, bp->b_blkno, bp->b_bcount >> DEV_BSHIFT, 0) == 0) 
    goto failed;
```

This example builds a Group 0 command descriptor block. The example fills in the `pkt_cdbp` field as follows:

- The command itself is in byte 0. The command is set from the parameter `SCMD_READ`.
- The address field is in bits 0-4 of byte 1 and bytes 2 and 3. The address is set from `bp->b_blkno`.
- The count field is in byte 4. The count is set from the last parameter. In this case, `count` is set to `bp->b_bcount >> DEV_BSHIFT`, where `DEV_BSHIFT` is the byte count of the transfer converted to the number of blocks.

**Note** — `scsi_setup_cdb(9F)` does not support setting a target device’s logical unit number (LUN) in bits 5-7 of byte 1 of the SCSI command block. This requirement is defined by SCSI-1. For SCSI-1 devices that require the LUN bits set in the command block, use `makecom_g0(9F)` or some equivalent rather than `scsi_setup_cdb(9F)`. 
After initializing the SCSI CDB, initialize three other fields in the packet and store as a pointer to the packet in the state structure.

```c
pkt->pkt_private = (opaque_t)bp;
pkt->pkt_comp = xxcallback;
pkt->pkt_time = 30;
xsp->pkt = pkt;
```

The `buf(9S)` pointer is saved in the `pkt_private` field for later use in the completion routine.

## Setting Target Capabilities

The target drivers use `scsi_ifsetcap(9F)` to set the capabilities of the host adapter driver. A `cap` is a name-value pair, consisting of a null-terminated character string and an integer value. The current value of a capability can be retrieved using `scsi_ifgetcap(9F)`. `scsi_ifsetcap(9F)` allows capabilities to be set for all targets on the bus.

In general, however, setting capabilities of targets that are not owned by the target driver is not recommended. This practice is not universally supported by HBA drivers. Some capabilities, such as disconnect and synchronous, can be set by default by the HBA driver. Other capabilities might need to be set explicitly by the target driver. Wide-xfer and tagged-queueing must be set by the target drive, for example.

## Transporting a Command

After the `scsi_pkt(9S)` structure is filled in, use `scsi_transport(9F)` to hand the structure to the bus adapter driver:

```c
if (scsi_transport(pkt) != TRAN_ACCEPT) {
    bp->b_resid = bp->b_bcount;
    bioerror(bp, EIO);
    biodone(bp);
}
```

The other return values from `scsi_transport(9F)` are as follows:

- `TRAN_BUSY` – A command for the specified target is already in progress.
- `TRAN_BADPKT` – The DMA count in the packet was too large, or the host adapter driver rejected this packet for other reasons.
- `TRAN_FATAL_ERROR` – The host adapter driver is unable to accept this packet.

---

**Note** – The mutex `sd_mutex` in the `scsi_device(9S)` structure must not be held across a call to `scsi_transport(9F)`. 

---
If `scsi_transport(9F)` returns `TRAN_ACCEPT`, the packet becomes the responsibility of the host bus adapter driver. The packet should not be accessed by the target driver until the command completion routine is called.

**Synchronous `scsi_transport()` Function**

If `FLAG_NOINTR` is set in the packet, then `scsi_transport(9F)` does not return until the command is complete. No callback is performed.

**Note** – Do not use `FLAG_NOINTR` in interrupt context.

**Command Completion**

When the host bus adapter driver is through with the command, the driver invokes the packet’s completion callback routine. The driver then passes a pointer to the `scsi_pkt(9S)` structure as a parameter. After decoding the packet, the completion routine takes the appropriate action.

**Example 17–5** presents a simple completion callback routine. This code checks for transport failures. In case of failure, the routine gives up rather than retrying the command. If the target is busy, extra code is required to resubmit the command at a later time.

If the command results in a check condition, the target driver needs to send a request sense command unless auto request sense has been enabled.

Otherwise, the command succeeded. At the end of processing for the command, the command destroys the packet and calls `biodone(9F)`.

In the event of a transport error, such as a bus reset or parity problem, the target driver can resubmit the packet by using `scsi_transport(9F)`. No values in the packet need to be changed prior to resubmitting.

The following example does not attempt to retry incomplete commands.

**Note** – Normally, the target driver’s callback function is called in interrupt context. Consequently, the callback function should never sleep.
EXAMPLE 17–5  Completion Routine for a SCSI Driver  (Continued)

```c
struct scsi_status *ssp;
/*
 * Get a pointer to the buf(9S) structure for the command
 * and to the per-instance data structure.
 */
bp = (struct buf *)pkt->pkt_private;
instance = getminor(bp->b_edev);
xsp = ddi_get_soft_state(statep, instance);
/*
 * Figure out why this callback routine was called
 */
if (pkt->pkt_reason != CMP_CMPLT) {
    bp->b_resid = bp->b_bcount;
    bioerror(bp, EIO);
    scsi_destroy_pkt(pkt); /* Release resources */
    biodone(bp); /* Notify waiting threads */
} else {
    /* Command completed, check status.
     * See scsi_status(9S)
     */
    ssp = (struct scsi_status *)pkt->pkt_scbp;
    if (ssp->sts_busy) {
        /* error, target busy or reserved */
    } else if (ssp->sts_chk) {
        /* Send a request sense command. */
    } else {
        bp->b_resid = pkt->pkt_resid; /* Packet completed OK */
        scsi_destroy_pkt(pkt);
        biodone(bp);
    }
}
```

### Reuse of Packets

A target driver can reuse packets in the following ways:

- Resubmit the packet unchanged.
- Use `scsi_sync_pkt(9F)` to synchronize the data. Then, process the data in the driver. Finally, resubmit the packet.
- Free DMA resources, using `scsi_dmafree(9F)`, and pass the `pkt` pointer to `scsi_init_pkt(9F)` for binding to a new `bp`. The target driver is responsible for reinitializing the packet. The CDB has to have the same length as the previous CDB.
- If only partial DMA is allocated during the first call to `scsi_init_pkt(9F)`, subsequent calls to `scsi_init_pkt(9F)` can be made for the same packet. Calls can be made to `bp` as well to adjust the DMA resources to the next portion of the transfer.
Auto-Request Sense Mode

Auto-request sense mode is most desirable if queuing is used, whether the queuing is tagged or untagged. A contingent allegiance condition is cleared by any subsequent command and, consequently, the sense data is lost. Most HBA drivers start the next command before performing the target driver callback. Other HBA drivers can use a separate, lower-priority thread to perform the callbacks. This approach might increase the time needed to notify the target driver that the packet completed with a check condition. In this case, the target driver might not be able to submit a request sense command in time to retrieve the sense data.

To avoid this loss of sense data, the HBA driver, or controller, should issue a request sense command if a check condition has been detected. This mode is known as auto-request sense mode. Note that not all HBA drivers are capable of auto-request sense mode, and some drivers can only operate with auto-request sense mode enabled.

A target driver enables auto-request-sense mode by using `scsi_ifsetcap(9F)`. The following example shows auto-request sense enabling.

**EXAMPLE 17–6  Enabling Auto-Request Sense Mode**

```c
static int
xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    struct xxstate *xsp;
    struct scsi_device *sdp = (struct scsi_device *)
        ddi_get_driver_private(dip);
    /*
    * Enable auto-request-sense. An auto-request-sense command might
    * fail due to a BUSY condition or transport error. Therefore,
    * it is recommended to allocate a separate request sense
    * packet as well.
    * Note that scsi_ifsetcap(9F) can return -1, 0, or 1
    */
    xsp->sdp_arq_enabled =
        ((scsi_ifsetcap(ROUTE, "auto-rqsense", 1, 1) == 1) ? 1 : 0);
    /*
    * If the HBA driver supports auto request sense then the
    * status blocks should be sizeof (struct scsi_arq_status).
    * Else, one byte is sufficient.
    */
    xsp->sdp_cmd_stat_size = (xsp->sdp_arq_enabled
               ? sizeof (struct scsi_arq_status) : 1);
    /* ... */
}
```

If a packet is allocated using `scsi_init_pkt(9F)` and auto-request sense is desired on this packet, additional space is needed. The target driver must request this space for the status block to hold the auto-request sense structure. The sense length used in the request sense command is `sizeof`, from `struct scsi_extended_sense`. Auto-request sense can be disabled per individual packet by allocating `sizeof`, from `struct scsi_status`, for the status block.
The packet is submitted using `scsi_transport(9F)` as usual. When a check condition occurs on this packet, the host adapter driver takes the following steps:

- Issues a request sense command if the controller does not have auto-request sense capability
- Obtains the sense data
- Fills in the `scsi_arq_status` information in the packet's status block
- Sets `STATE_ARQ_DONE` in the packet's `pkt_state` field
- Calls the packet's callback handler (`pkt_comp()`)

The target driver's callback routine should verify that sense data is available by checking the `STATE_ARQ_DONE` bit in `pkt_state`. `STATE_ARQ_DONE` implies that a check condition has occurred and that a request sense has been performed. If auto-request sense has been temporarily disabled in a packet, subsequent retrieval of the sense data cannot be guaranteed.

The target driver should then verify whether the auto-request sense command completed successfully and decode the sense data.

### Dump Handling

The `dump(9E)` entry point copies a portion of virtual address space directly to the specified device in the case of system failure or checkpoint operation. See the `cpr(7)` and `dump(9E)` man pages. The `dump(9E)` entry point must be capable of performing this operation without the use of interrupts.

The arguments for `dump()` are as follows:

- `dev` Device number of the dump device
- `addr` Kernel virtual address at which to start the dump
- `blkno` First destination block on the device
- `nblk` Number of blocks to dump

**EXAMPLE 17-7** `dump(9E)` Routine

```c
static int
xxdump(dev_t dev, caddr_t addr, daddr_t blkno, int nblk)
{
    struct xxstate  *xsp;
    struct buf      *bp;
    struct scsi_pkt *pkt;
    int      rval;
    int      instance;

    instance = getminor(dev);
    xsp = ddi_get_soft_state(statep, instance);
```
EXAMPLE 17–7  dump(9E) Routine (Continued)

    if (tgt->suspended) {
        (void) pm_raise_power(DEVINFO(tgt), 0, 1);
    }

    bp = getrbuf(KM_NOSLEEP);
    if (bp == NULL) {
        return (EIO);
    }

    /* Calculate block number relative to partition. */
    bp->b.un.b_addr = addr;
    bp->b.e.dev = dev;
    bp->b.b.count = nblk * DEV_BSIZE;
    bp->b.flags = B_WRITE | B_BUSY;
    bp->b.blkno = blkno;

    pkt = scsi_init_pkt(ROUTE(tgt), NULL, bp, CDB_GROUP1,
                           sizeof (struct scsi_arq_status),
                           sizeof (struct bst_pkt_private), 0, NULL_FUNC, NULL);
    if (pkt == NULL) {
        freerbuf(bp);
        return (EIO);
    }
    (void) scsi_setup_cdb((union scsi_cdb *)pkt->pkt_cdbp,
                          SCMD_WRITE_G1, blkno, nblk, 0);

    /* While dumping in polled mode, other cmds might complete
    * and these should not be resubmitted. we set the
    * dumping flag here which prevents requeueing cmds.
    */
    tgt->dumping = 1;
    rval = scsi_poll(pkt);
    tgt->dumping = 0;

    scsi_destroy_pkt(pkt);
    freerbuf(bp);

    if (rval != DDI_SUCCESS) {
        rval = EIO;
    }
    return (rval);

SCSI Options

SCSA defines a global variable, scsi_options, for control and debugging. The defined bits in
scsi_options can be found in the file <sys/scsi/conf/autoconf.h>. The scsi_options uses the
bits as follows:

SCSI_OPTIONS_DR   Enables global disconnect or reconnect.
<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCSI_OPTIONS_FAST</td>
<td>Enables global FAST SCSI support: 10 Mbytes/sec transfers. The HBA should</td>
</tr>
<tr>
<td></td>
<td>not operate in FAST SCSI mode unless the SCSI_OPTIONS_FAST (0x100) bit is</td>
</tr>
<tr>
<td></td>
<td>set.</td>
</tr>
<tr>
<td>SCSI_OPTIONS_FAST20</td>
<td>Enables global FAST20 SCSI support: 20 Mbytes/sec transfers. The HBA</td>
</tr>
<tr>
<td></td>
<td>should not operate in FAST20 SCSI mode unless the SCSI_OPTIONS_FAST20 (0x400)</td>
</tr>
<tr>
<td></td>
<td>bit is set.</td>
</tr>
<tr>
<td>SCSI_OPTIONS_FAST40</td>
<td>Enables global FAST40 SCSI support: 40 Mbytes/sec transfers. The HBA</td>
</tr>
<tr>
<td></td>
<td>should not operate in FAST40 SCSI mode unless the SCSI_OPTIONS_FAST40 (0x800)</td>
</tr>
<tr>
<td></td>
<td>bit is set.</td>
</tr>
<tr>
<td>SCSI_OPTIONS_FAST80</td>
<td>Enables global FAST80 SCSI support: 80 Mbytes/sec transfers. The HBA</td>
</tr>
<tr>
<td></td>
<td>should not operate in FAST80 SCSI mode unless the SCSI_OPTIONS_FAST80 (0x1000)</td>
</tr>
<tr>
<td></td>
<td>bit is set.</td>
</tr>
<tr>
<td>SCSI_OPTIONS_FAST160</td>
<td>Enables global FAST160 SCSI support: 160 Mbytes/sec transfers. The HBA</td>
</tr>
<tr>
<td></td>
<td>should not operate in FAST160 SCSI mode unless the SCSI_OPTIONS_FAST160 (0x2000)</td>
</tr>
<tr>
<td></td>
<td>bit is set.</td>
</tr>
<tr>
<td>SCSI_OPTIONS_FAST320</td>
<td>Enables global FAST320 SCSI support: 320 Mbytes/sec transfers. The HBA</td>
</tr>
<tr>
<td></td>
<td>should not operate in FAST320 SCSI mode unless the SCSI_OPTIONS_FAST320 (0x4000)</td>
</tr>
<tr>
<td></td>
<td>bit is set.</td>
</tr>
<tr>
<td>SCSI_OPTIONS_LINK</td>
<td>Enables global link support.</td>
</tr>
<tr>
<td>SCSI_OPTIONS_PARITY</td>
<td>Enables global parity support.</td>
</tr>
<tr>
<td>SCSI_OPTIONS_QAS</td>
<td>Enables the Quick Arbitration Select feature. QAS is used to decrease</td>
</tr>
<tr>
<td></td>
<td>protocol overhead when devices arbitrate for and access the bus. QAS is</td>
</tr>
<tr>
<td></td>
<td>only supported on Ultra4 (FAST160) SCSI devices, although not all such</td>
</tr>
<tr>
<td></td>
<td>devices support QAS. The HBA should not operate in QAS SCSI mode unless the</td>
</tr>
<tr>
<td></td>
<td>SCSI_OPTIONS_QAS (0x100000) bit is set. Consult the appropriate Oracle</td>
</tr>
<tr>
<td></td>
<td>hardware documentation to determine whether your machine supports QAS.</td>
</tr>
<tr>
<td>SCSI_OPTIONS_SYNC</td>
<td>Enables global synchronous transfer capability.</td>
</tr>
<tr>
<td>SCSI_OPTIONS_TAG</td>
<td>Enables global tagged queuing support.</td>
</tr>
<tr>
<td>SCSI_OPTIONS_WIDE</td>
<td>Enables global WIDE SCSI.</td>
</tr>
</tbody>
</table>

**Note** – The setting of *scsi_options* affects all host bus adapter drivers and all target drivers that are present on the system. Refer to the *scsi_hba_attach(9F)* man page for information on controlling these options for a particular host adapter.
This chapter contains information on creating SCSI host bus adapter (HBA) drivers. The chapter provides sample code illustrating the structure of a typical HBA driver. The sample code shows the use of the HBA driver interfaces that are provided by the Sun Common SCSI Architecture (SCSA). This chapter provides information on the following subjects:

- “Introduction to Host Bus Adapter Drivers” on page 331
- “SCSI Interface” on page 332
- “SCSA HBA Interfaces” on page 334
- “HBA Driver Dependency and Configuration Issues” on page 344
- “Entry Points for SCSA HBA Drivers” on page 350
- “SCSI HBA Driver Specific Issues” on page 376
- “Support for Queuing” on page 379

**Introduction to Host Bus Adapter Drivers**

As described in Chapter 17, “SCSI Target Drivers,” the DDI/DKI divides the software interface to SCSI devices into two major parts:

- Target devices and drivers
- Host bus adapter devices and drivers

*Target device* refers to a device on a SCSI bus, such as a disk or a tape drive. *Target driver* refers to a software component installed as a device driver. Each target device on a SCSI bus is controlled by one instance of the target driver.

*Host bus adapter device* refers to HBA hardware, such as an SBus or PCI SCSI adapter card. *Host bus adapter driver* refers to a software component that is installed as a device driver. Some examples are the esp driver on a SPARC machine, the ncrs driver on an x86 machine, and the isp driver, which works on both architectures. An instance of the HBA driver controls each of its host bus adapter devices that are configured in the system.
The Sun Common SCSI Architecture (SCSA) defines the interface between the target and HBA components.

**Note** – Understanding SCSI target drivers is an essential prerequisite to writing effective SCSI HBA drivers. For information on SCSI target drivers, see Chapter 17, “SCSI Target Drivers.” Target driver developers can also benefit from reading this chapter.

The host bus adapter driver is responsible for performing the following tasks:

- Managing host bus adapter hardware
- Accepting SCSI commands from the SCSI target driver
- Transporting the commands to the specified SCSI target device
- Performing any data transfers that the command requires
- Collecting status
- Handling auto-request sense (optional)
- Informing the target driver of command completion or failure

**SCSI Interface**

SCSA is the DDI/DKI programming interface for the transmission of SCSI commands from a target driver to a host adapter driver. By conforming to the SCSA, the target driver can easily pass any combination of SCSI commands and sequences to a target device. Knowledge of the hardware implementation of the host adapter is not necessary. Conceptually, SCSA separates the building of a SCSI command from the transporting of the command with data to the SCSI bus. SCSA manages the connections between the target and HBA drivers through an HBA transport layer, as shown in the following figure.
The HBA transport layer is a software and hardware layer that is responsible for transporting a SCSI command to a SCSI target device. The HBA driver provides resource allocation, DMA management, and transport services in response to requests made by SCSI target drivers through SCSA. The host adapter driver also manages the host adapter hardware and the SCSI protocols necessary to perform the commands. When a command has been completed, the HBA driver calls the target driver's SCSI pkt command completion routine.

The following example illustrates this flow, with emphasis on the transfer of information from target drivers to SCSA to HBA drivers. The figure also shows typical transport entry points and function calls.
SCSA HBA Interfaces

SCSA HBA interfaces include HBA entry points, HBA data structures, and an HBA framework.

SCSA HBA Entry Point Summary

SCSA defines a number of HBA driver entry points. These entry points are listed in the following table. The entry points are called by the system when a target driver instance connected to the HBA driver is configured. The entry points are also called when the target driver makes a SCSA request. See “Entry Points for SCSA HBA Drivers” on page 350 for more information.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Called as a Result of</th>
</tr>
</thead>
<tbody>
<tr>
<td>tran_abort(9E)</td>
<td>Target driver calling scsi_abort(9F)</td>
</tr>
<tr>
<td>tran_bus_reset(9E)</td>
<td>System resetting bus</td>
</tr>
<tr>
<td>tran_destroy_pkt(9E)</td>
<td>Target driver calling scsi_destroy_pkt(9F)</td>
</tr>
</tbody>
</table>
### TABLE 18–1  SCSA HBA Entry Point Summary (Continued)

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Called as a Result of</th>
</tr>
</thead>
<tbody>
<tr>
<td>tran_dmafree(9E)</td>
<td>Target driver calling scsi_dmafree(9F)</td>
</tr>
<tr>
<td>tran_getcap(9E)</td>
<td>Target driver calling scsi_ifgetcap(9F)</td>
</tr>
<tr>
<td>tran_init_pkt(9E)</td>
<td>Target driver calling scsi_init_pkt(9F)</td>
</tr>
<tr>
<td>tran_quiesce(9E)</td>
<td>System quiescing bus</td>
</tr>
<tr>
<td>tran_reset(9E)</td>
<td>Target driver calling scsi_reset(9F)</td>
</tr>
<tr>
<td>tran_reset_notify(9E)</td>
<td>Target driver calling scsi_reset_notify(9F)</td>
</tr>
<tr>
<td>tran_setcap(9E)</td>
<td>Target driver calling scsi_ifsetcap(9F)</td>
</tr>
<tr>
<td>tran_start(9E)</td>
<td>Target driver calling scsi_transport(9F)</td>
</tr>
<tr>
<td>tran_sync_pkt(9E)</td>
<td>Target driver calling scsi_sync_pkt(9F)</td>
</tr>
<tr>
<td>tran_tgt_free(9E)</td>
<td>System detaching target device instance</td>
</tr>
<tr>
<td>tran_tgt_init(9E)</td>
<td>System attaching target device instance</td>
</tr>
<tr>
<td>tran_tgt_probe(9E)</td>
<td>Target driver calling scsi_probe(9F)</td>
</tr>
<tr>
<td>tran_unquiesce(9E)</td>
<td>System resuming activity on bus</td>
</tr>
</tbody>
</table>

### SCSA HBA Data Structures

SCSA defines data structures to enable the exchange of information between the target and HBA drivers. The following data structures are included:

- `scsi_hba_tran(9S)`
- `scsi_address(9S)`
- `scsi_device(9S)`
- `scsi_pkt(9S)`

#### `scsi_hba_tran()` Structure

Each instance of an HBA driver must allocate a `scsi_hba_tran(9S)` structure by using the `scsi_hba_tran_alloc(9F)` function in the `attach(9E)` entry point. The `scsi_hba_tran_alloc()` function initializes the `scsi_hba_tran` structure. The HBA driver must initialize specific vectors in the transport structure to point to entry points within the HBA driver. After the `scsi_hba_tran` structure is initialized, the HBA driver exports the transport structure to SCSA by calling the `scsi_hbaAttach_setup(9F)` function.
Caution – Because SCSA keeps a pointer to the transport structure in the driver-private field on the devinfo node, HBA drivers must not use ddi_set_driver_private(9F). HBA drivers can, however, use ddi_get_driver_private(9F) to retrieve the pointer to the transport structure.

The SCSA interfaces require the HBA driver to supply a number of entry points that are callable through the scsi_hba_tran structure. See "Entry Points for SCSA HBA Drivers" on page 350 for more information.

The scsi_hba_tran structure contains the following fields:

```c
struct scsi_hba_tran {
    dev_info_t tran_hba_dip; /* HBAs dev_info pointer */
    void *tran_hba_private; /* HBA softstate */
    void *tran_tgt_private; /* HBA target private pointer */
    struct scsi_device *tran_sd; /* scsi_device */
    int (*tran_tgt_init)(); /* Transport target */
        /* Initialization */
    int (*tran_tgt_probe)(); /* Transport target probe */
    int (*tran_start)(); /* Transport start */
    int (*tran_reset)(); /* Transport reset */
    int (*tran_abort)(); /* Transport abort */
    int (*tran_getcap)(); /* Capability retrieval */
    int (*tran_setcap)(); /* Capability establishment */
    struct scsi_pkt *(*tran_init_pkt)(); /* Packet and DMA allocation */
    void (*tran_destroy_pkt)(); /* Packet and DMA deallocation */
    void (*tran_dma_free)(); /* DMA deallocation */
    void (*tran_sync_pkt)(); /* Sync DMA */
    void (*tran_reset_notify)(); /* Bus reset notification */
    int (*tran_bus_reset)(); /* Reset bus only */
    int (*tran_quiesce)(); /* Quiesce a bus */
    int (*tran_unquiesce)(); /* Unquiesce a bus */
    int tran_interconnect_type; /* transport interconnect */
};
```

The following descriptions give more information about these scsi_hba_tran structure fields:

- **tran_hba_dip** Pointer to the HBA device instance dev_info structure. The function scsi_hba_attach_setup(9F) sets this field.

- **tran_hba_private** Pointer to private data maintained by the HBA driver. Usually, tran_hba_private contains a pointer to the state structure of the HBA driver.

- **tran_tgt_private** Pointer to private data maintained by the HBA driver when using cloning. By specifying SCSI_HBA_TRAN_CLONE when calling scsi_hba_attach_setup(9F), the scsi_hba_tran(9S) structure is cloned once per target. This approach enables the HBA to initialize this field to point to a per-target instance data structure in the tran_tgt_init(9E) entry point. If
SCSI_HBA_TRAN_CLONE is not specified, tran_tgt_private is NULL, and tran_tgt_private must not be referenced. See “Transport Structure Cloning” on page 342 for more information.

**tran_sd**
Point to a per-target instance *scsi_device*(9S) structure used when cloning. If SCSI_HBA_TRAN_CLONE is passed to *scsi_hbaattach_setup*(9F), tran_sd is initialized to point to the per-target *scsi_device* structure. This initialization takes place before any HBA functions are called on behalf of that target. If SCSI_HBA_TRAN_CLONE is not specified, tran_sd is NULL, and tran_sd must not be referenced. See “Transport Structure Cloning” on page 342 for more information.

**tran_tgt_init**
Pointer to the HBA driver entry point that is called when initializing a target device instance. If no per-target initialization is required, the HBA can leave tran_tgt_init set to NULL.

**tran_tgt_probe**
Pointer to the HBA driver entry point that is called when a target driver instance calls *scsi_probe*(9F). This routine is called to probe for the existence of a target device. If no target probing customization is required for this HBA, the HBA should set tran_tgt_probe to *scsi_hba_probe*(9F).

**tran_tgt_free**
Pointer to the HBA driver entry point that is called when a target device instance is destroyed. If no per-target deallocation is necessary, the HBA can leave tran_tgt_free set to NULL.

**tran_start**
Pointer to the HBA driver entry point that is called when a target driver calls *scsi_transport*(9F).

**tran_reset**
Pointer to the HBA driver entry point that is called when a target driver calls *scsi_reset*(9F).

**tran_abort**
Pointer to the HBA driver entry point that is called when a target driver calls *scsi_abort*(9F).

**tran_getcap**
Pointer to the HBA driver entry point that is called when a target driver calls *scsi_ifgetcap*(9F).

**tran_setcap**
Pointer to the HBA driver entry point that is called when a target driver calls *scsi_ifsetcap*(9F).

**tran_init_pkt**
Pointer to the HBA driver entry point that is called when a target driver calls *scsi_init_pkt*(9F).

**tran_destroy_pkt**
Pointer to the HBA driver entry point that is called when a target driver calls *scsi_destroy_pkt*(9F).
tran_dmafree  
Pointer to the HBA driver entry point that is called when a target driver calls `scsi_dmafree(9F)`.

tran_sync_pkt  
Pointer to the HBA driver entry point that is called when a target driver calls `scsi_sync_pkt(9F)`.

tran_reset_notify  
Pointer to the HBA driver entry point that is called when a target driver calls `tran_reset_notify(9E)`.

tran_bus_reset  
The function entry that resets the SCSI bus without resetting targets.

tran_quiesce  
The function entry that waits for all outstanding commands to complete and blocks (or queues) any I/O requests issued.

tran_unquiesce  
The function entry that allows I/O activities to resume on the SCSI bus.

tran_interconnect_type  
Integer value denoting interconnect type of the transport as defined in the `services.h` header file.

**scsi_address Structure**

The `scsi_address(9S)` structure provides transport and addressing information for each SCSI command that is allocated and transported by a target driver instance.

The `scsi_address` structure contains the following fields:

```c
struct scsi_address {
    struct scsi_hba_tran *a_hba_tran; /* Transport vectors */
    ushort_t a_target; /* Target identifier */
    uchar_t a_lun; /* LUN on that target */
    uchar_t a_sublun; /* Sub LUN on that LUN */
};
```

- `a_hba_tran`: Pointer to the `scsi_hba_tran(9S)` structure, as allocated and initialized by the HBA driver. If `SCSI_HBA_TRAN_CLONE` was specified as the flag to `scsi_hba_attach_setup(9F)`, `a_hba_tran` points to a copy of that structure.

- `a_target`: Identifies the SCSI target on the SCSI bus.

- `a_lun`: Identifies the SCSI logical unit on the SCSI target.

**scsi_device Structure**

The HBA framework allocates and initializes a `scsi_device(9S)` structure for each instance of a target device. The allocation and initialization occur before the framework calls the HBA driver's `tran_tgt_init(9E)` entry point. This structure stores information about each SCSI
logical unit, including pointers to information areas that contain both generic and device-specific information. One `scsi_device(9S)` structure exists for each target device instance that is attached to the system.

If the per-target initialization is successful, the HBA framework sets the target driver's per-instance private data to point to the `scsi_device(9S)` structure, using `ddi_set_driver_private(9F)`. Note that an initialization is successful if `tran_tgt_init()` returns success or if the vector is null.

The `scsi_device(9S)` structure contains the following fields:

```c
struct scsi_device {
    struct scsi_address sd_address; /* routing information */
    dev_info_t *sd_dev; /* device dev_info node */
    kmutex_t sd_mutex; /* mutex used by device */
    void sd_reserved;
    struct scsi_inquiry *sd_inq;
    struct scsi_extended_sense *sd_sense;
    caddr_t sd_private; /* for driver’s use */
};
```

where:

- **sd_address**: Data structure that is passed to the routines for SCSI resource allocation.
- **sd_dev**: Pointer to the target’s dev_info structure.
- **sd_mutex**: Mutex for use by the target driver. This mutex is initialized by the HBA framework. The mutex can be used by the target driver as a per-device mutex. This mutex should not be held across a call to `scsi_transport(9F)` or `scsi_poll(9F)`. See Chapter 3, “Multithreading,” for more information on mutexes.
- **sd_inq**: Pointer for the target device’s SCSI inquiry data. The `scsi_probe(9F)` routine allocates a buffer, fills the buffer in, and attaches the buffer to this field.
- **sd_sense**: Pointer to a buffer to contain request sense data from the device. The target driver must allocate and manage this buffer itself. See the target driver’s `attach(9E)` routine in “attach() Entry Point” on page 104 for more information.
- **sd_private**: Pointer field for use by the target driver. This field is commonly used to store a pointer to a private target driver state structure.

**scsi_pkt Structure (HBA)**

To execute SCSI commands, a target driver must first allocate a `scsi_pkt(9S)` structure for the command. The target driver must then specify its own private data area length, the command status, and the command length. The HBA driver is responsible for implementing the packet
allocation in the `tran_init_pkt(9E)` entry point. The HBA driver is also responsible for freeing the packet in its `tran_destroy_pkt(9E)` entry point. See “`scsi_pkt Structure (Target Drivers)`” on page 313 for more information.

The `scsi_pkt(9S)` structure contains these fields:

```c
struct scsi_pkt {
    opaque_t pkt_ha_private; /* private data for host adapter */
    struct scsi_address pkt_address; /* destination address */
    opaque_t pkt_private; /* private data for target driver */
    void (*pkt_comp)(struct scsi_pkt *); /* completion routine */
    uint_t pkt_flags; /* flags */
    int pkt_time; /* time allotted to complete command */
    uchar_t *pkt_scbp; /* pointer to status block */
    uchar_t *pkt_cdbp; /* pointer to command block */
    ssize_t pkt_resid; /* data bytes not transferred */
    uint_t pkt_state; /* state of command */
    uint_t pkt_statistics; /* statistics */
    uchar_t pkt_reason; /* reason completion called */
};
```

where:

- **pkt_ha_private** Pointer to per-command HBA-driver private data.
- **pkt_address** Pointer to the `scsi_address(9S)` structure providing address information for this command.
- **pkt_private** Pointer to per-packet target-driver private data.
- **pkt_comp** Pointer to the target-driver completion routine called by the HBA driver when the transport layer has completed this command.
- **pkt_flags** Flags for the command.
- **pkt_time** Specifies the completion timeout in seconds for the command.
- **pkt_scbp** Pointer to the status completion block for the command.
- **pkt_cdbp** Pointer to the command descriptor block (CDB) for the command.
- **pkt_resid** Count of the data bytes that were not transferred when the command completed. This field can also be used to specify the amount of data for which resources have not been allocated. The HBA must modify this field during transport.
- **pkt_state** State of the command. The HBA must modify this field during transport.
- **pkt_statistics** Provides a history of the events that the command experienced while in the transport layer. The HBA must modify this field during transport.
- **pkt_reason** Reason for command completion. The HBA must modify this field during transport.
Per-Target Instance Data

An HBA driver must allocate a `scsi_hba_tran(9S)` structure during `attach(9E)`. The HBA driver must then initialize the vectors in this transport structure to point to the required entry points for the HBA driver. This `scsi_hba_tran` structure is then passed into `scsi_hba_attach_setup(9F)`.

The `scsi_hba_tran` structure contains a `tran_hba_private` field, which can be used to refer to the HBA driver's per-instance state.

Each `scsi_address(9S)` structure contains a pointer to the `scsi_hba_tran` structure. In addition, the `scsi_address` structure provides the target, that is, a `target` and logical unit (a `lun`) addresses for the particular target device. Each entry point for the HBA driver is passed a pointer to the `scsi_address` structure, either directly or indirectly through the `scsi_device(9S)` structure. As a result, the HBA driver can reference its own state. The HBA driver can also identify the target device that is addressed.

The following figure illustrates the HBA data structures for transport operations.

FIGURE 18-3  HBA Transport Structures

- `scsi_address` structure
- `scsi_device` structure
- `scsi_hba_tran` structure
- HBA private data pointer
- Transport vectors
- HBA soft state structure
- `Per HBA device instance`
- HBA driver entry points
- SCSI Bus
- One SCSI device structure per target device instance
- `Per target device instance`
- `Per HBA device instance`
- Target devices
Transport Structure Cloning

Cloning can be useful if an HBA driver needs to maintain per-target private data in the `scsi_hba_tran(9S)` structure. Cloning can also be used to maintain a more complex address than is provided in the `scsi_address(9S)` structure.

In the cloning process, the HBA driver must still allocate a `scsi_hba_tran` structure at `attach(9E)` time. The HBA driver must also initialize the `tran_hba_private` soft state pointer and the entry point vectors for the HBA driver. The difference occurs when the framework begins to connect an instance of a target driver to the HBA driver. Before calling the HBA driver's `tran_tgt_init(9E)` entry point, the framework clones the `scsi_hba_tran` structure that is associated with that instance of the HBA. Accordingly, each `scsi_address` structure that is allocated and initialized for a particular target device instance points to a per-target instance `copy` of the `scsi_hba_tran` structure. The `scsi_address` structures do not point to the `scsi_hba_tran` structure that is allocated by the HBA driver at `attach()` time.

An HBA driver can use two important pointers when cloning is specified. These pointers are contained in the `scsi_hba_tran` structure. The first pointer is the `tran_tgt_private` field, which the driver can use to point to per-target HBA private data. The `tran_tgt_private` pointer is useful, for example, if an HBA driver needs to maintain a more complex address than `a_target` and `a_lun` provide. The second pointer is the `tran_sd` field, which is a pointer to the `scsi_device(9S)` structure referring to the particular target device.

When specifying cloning, the HBA driver must allocate and initialize the per-target data. The HBA driver must then initialize the `tran_tgt_private` field to point to this data during its `tran_tgt_init(9E)` entry point. The HBA driver must free this per-target data during its `tran_tgt_free(9E)` entry point.

When cloning, the framework initializes the `tran_sd` field to point to the `scsi_device` structure before the HBA driver `tran_tgt_init()` entry point is called. The driver requests cloning by passing the `SCSI_HBA_TRAN_CLONE` flag to `scsi_hba_attach_setup(9F)`. The following figure illustrates the HBA data structures for cloning transport operations.
SCSA HBA Functions

SCSA also provides a number of functions. The functions are listed in the following table, for use by HBA drivers.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Called by Driver Entry Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>scsi_hba_init(9F)</td>
<td>_init(9E)</td>
</tr>
<tr>
<td>scsi_hba_fini(9F)</td>
<td>_fini(9E)</td>
</tr>
<tr>
<td>scsi_hba_attach_setup(9F)</td>
<td>attach(9E)</td>
</tr>
<tr>
<td>scsi_hba_detach(9F)</td>
<td>detach(9E)</td>
</tr>
<tr>
<td>scsi_hba_tran_alloc(9F)</td>
<td>attach(9E)</td>
</tr>
</tbody>
</table>
HBA Driver Dependency and Configuration Issues

In addition to incorporating SCSA HBA entry points, structures, and functions into a driver, a developer must deal with driver dependency and configuration issues. These issues involve configuration properties, dependency declarations, state structure and per-command structure, entry points for module initialization, and autoconfiguration entry points.

Declarations and Structures

HBA drivers must include the following header files:

```c
#include <sys/scsi/scsi.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

To inform the system that the module depends on SCSA routines, the driver binary must be generated with the following command. See “SCSA HBA Interfaces” on page 334 for more information on SCSA routines.

```bash
% ld -r xx.o -o xx -N "misc/scsi"
```

The code samples are derived from a simplified isp driver for the QLogic Intelligent SCSI Peripheral device. The isp driver supports WIDE SCSI, with up to 15 target devices and 8 logical units (LUNs) per target.

Per-Command Structure

An HBA driver usually needs to define a structure to maintain state for each command submitted by a target driver. The layout of this per-command structure is entirely up to the device driver writer. The layout needs to reflect the capabilities and features of the hardware and the software algorithms that are used in the driver.

The following structure is an example of a per-command structure. The remaining code fragments of this chapter use this structure to illustrate the HBA interfaces.

---

### TABLE 18–2  
SCSA HBA Functions *(Continued)*

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Called by Driver Entry Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>scsi_hba_tran_free(9F)</td>
<td>detach(9E)</td>
</tr>
<tr>
<td>scsi_hba_probe(9F)</td>
<td>tran_tgt_probe(9E)</td>
</tr>
<tr>
<td>scsi_hba_pkt_alloc(9F)</td>
<td>tran_init_pkt(9E)</td>
</tr>
<tr>
<td>scsi_hba_pkt_free(9F)</td>
<td>tran_destroy_pkt(9E)</td>
</tr>
<tr>
<td>scsi_hba_lookup_capstr(9F)</td>
<td>tran_getcap(9E) and tran_setcap(9E)</td>
</tr>
</tbody>
</table>
struct isp_cmd {
    struct isp_request    cmd_isp_request;
    struct isp_response    cmd_isp_response;
    struct scsi_pkt *cmd_pkt;
    struct isp_cmd *cmd_forw;
    uint32_t    cmd_dmacount;
    ddi_dma_handle_t    cmd_dmahandle;
    uint_t    cmd_cookie;
    uint_t    cmd_ncookies;
    uint_t    cmd_cookiecnt;
    uint_t    cmd_nwin;
    uint_t    cmd_curwin;
    off_t    cmd_dma_offset;
    uint_t    cmd_dma_len;
    ddi_dma_cookie_t cmd_dmacookies[ISP_NDATASEGS];
    u_int    cmd_flags;
    u_short    cmd_slot;
    u_int    cmd_cdblen;
    u_int    cmd_scblen;
};

Entry Points for Module Initialization

This section describes the entry points for operations that are performed by SCSI HBA drivers.

The following code for a SCSI HBA driver illustrates a representative dev_ops(9S) structure. The driver must initialize the devo_bus_ops field in this structure to NULL. A SCSI HBA driver can provide leaf driver interfaces for special purposes, in which case the devo_cb_ops field might point to a cb_ops(9S) structure. In this example, no leaf driver interfaces are exported, so the devo_cb_ops field is initialized to NULL.

_init() Entry Point (SCSI HBA Drivers)

The _init(9E) function initializes a loadable module. _init() is called before any other routine in the loadable module.

In a SCSI HBA, the _init() function must call scsi_hba_init(9F) to inform the framework of the existence of the HBA driver before calling mod_install(9F). If scsi_hba__init() returns a nonzero value, _init() should return this value. Otherwise, _init() must return the value returned by mod_install(9F).

The driver should initialize any required global state before calling mod_install(9F).

If mod_install() fails, the _init() function must free any global resources allocated. _init() must call scsi_hba_fini(9F) before returning.

The following example uses a global mutex to show how to allocate data that is global to all instances of a driver. The code declares global mutex and soft-state structure information. The global mutex and soft state are initialized during _init().
fini() _ EntryPoint (SCSI HBA Drivers)

The fini(9E) function is called when the system is about to try to unload the SCSI HBA driver. The fini() function must call mod_remove(9F) to determine whether the driver can be unloaded. If mod_remove() returns 0, the module can be unloaded. The HBA driver must deallocate any global resources allocated in _init(9E). The HBA driver must also call scsi_hba_fini(9F).

_fini() must return the value returned by mod_remove().

Note – The HBA driver must not free any resources or call scsi_hba_fini(9F) unless mod_remove(9F) returns 0.

Example 18–1 shows module initialization for SCSI HBA.

EXA M P L E 18–1  Module Initialization for SCSI HBA

static struct dev_ops isp_dev_ops = {
  DEVO_REV, /* devo_rev */
  0, /* refcnt */
  isp_getinfo, /* getinfo */
  nulldev, /* identify */
  nulldev, /* probe */
  isp_attach, /* attach */
  isp_detach, /* detach */
  nodev, /* reset */
  NULL, /* driver operations */
  NULL, /* bus operations */
  isp_power, /* power management */
  isp_quiesce, /* quiesce */
};

/* * Local static data */
static kmutex_t isp_global_mutex;
static void *isp_state;

int _init(void)
{
  int err;

  if ((err = ddi_soft_state_init(&isp_state,
        sizeof (struct isp), 0)) != 0) {
    return (err);
  }
  if ((err = scsi_hba_init(&modlinkage)) == 0) {
    mutex_init(&isp_global_mutex, "isp global mutex",
               M UT EX _ Driver, NULL);
    if ((err = mod_install(&modlinkage)) != 0) {
      mutex_destroy(&isp_global_mutex);
      scsi_hba_fini(&modlinkage);
    }
  }
  return (0);
}
EXAMPLE 18–1  Module Initialization for SCSI HBA  (Continued)

    ddi_soft_state_fini(&isp_state);
  }
  return (err);
}

int
_fini(void)
{
  int   err;
  if ((err = mod_remove(&modlinkage)) == 0) {
    mutex_destroy(&isp_global_mutex);
    scsi_hba_fini(&modlinkage);
    ddi_soft_state_fini(&isp_state);
  }
  return (err);
}

Autoconfiguration Entry Points

Associated with each device driver is a dev_ops(9S) structure, which enables the kernel to locate
the autoconfiguration entry points of the driver. A complete description of these
autoconfiguration routines is given in Chapter 6, “Driver Autoconfiguration.” This section
describes only those entry points associated with operations performed by SCSI HBA drivers.
These entry points include attach(9E) and detach(9E).

attach() Entry Point (SCSI HBA Drivers)

The attach(9E) entry point for a SCSI HBA driver performs several tasks when configuring and
attaching an instance of the driver for the device. For a typical driver of real devices, the
following operating system and hardware concerns must be addressed:

- Soft-state structure
- DMA
- Transport structure
- Attaching an HBA driver
- Register mapping
- Interrupt specification
- Interrupt handling
- Create power manageable components
- Report attachment status

Soft-State Structure

When allocating the per-device-instance soft-state structure, a driver must clean up carefully if
an error occurs.
DMA

The HBA driver must describe the attributes of its DMA engine by properly initializing the `ddi_dma_attr_t` structure.

```c
static ddi_dma_attr_t isp_dma_attr = {
    DMA_ATTR_V0, /* ddi_dma_attr version */
    0, /* low address */
    0xffffffff, /* high address */
    0x00ffffff, /* counter upper bound */
    1, /* alignment requirements */
    0x3f, /* burst sizes */
    1, /* minimum DMA access */
    0xffffffff, /* maximum DMA access */
    (1<<24)-1, /* segment boundary restrictions */
    1, /* scatter-gather list length */
    512, /* device granularity */
    0 /* DMA flags */
};
```

The driver, if providing DMA, should also check that its hardware is installed in a DMA-capable slot:

```c
if (ddi_slaveonly(dip) == DDI_SUCCESS) {
    return (DDI_FAILURE);
}
```

Transport Structure

The driver should further allocate and initialize a transport structure for this instance. The `tran_hba_private` field is set to point to this instance’s soft-state structure. The `tran_tgt_probe` field can be set to `NULL` to achieve the default behavior, if no special probe customization is needed.

```c
tran = scsi_hba_tran_alloc(dip, SCSI_HBA_CANSLEEP);
isp->isp_tran = tran;
isp->isp_dip = dip;
tran->tran_hba_private = isp;
tran->tran_tgt_private = NULL;
tran->tran_tgt_init = isp_tran_tgt_init;
tran->tran_tgt_probe = scsi_hba_probe;
tran->tran_tgt_free = (void (*)())NULL;
tran->tran_start = isp_scsi_start;
tran->tran_abort = isp_scsi_abort;
tran->tran_reset = isp_scsi_reset;
tran->tran_getcap = isp_scsi_getcap;
tran->tran_setcap = isp_scsi_setcap;
tran->tran_init_pkt = isp_scsi_init_pkt;
tran->tran_destroy_pkt = isp_scsi_destroy_pkt;
tran->tran_dmafree = isp_scsi_dmafree;
tran->tran_sync_pkt = isp_scsi_sync_pkt;
```
<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>tran-&gt;tran_reset_notify</td>
<td>isp_scsi_reset_notify</td>
</tr>
<tr>
<td>tran-&gt;tran_bus_quiesce</td>
<td>isp_tran_bus_quiesce</td>
</tr>
<tr>
<td>tran-&gt;tran_bus_unquiesce</td>
<td>isp_tran_bus_unquiesce</td>
</tr>
<tr>
<td>tran-&gt;tran_bus_reset</td>
<td>isp_tran_bus_reset</td>
</tr>
<tr>
<td>tran-&gt;tran_interconnect_type</td>
<td>isp_tran_interconnect_type</td>
</tr>
</tbody>
</table>

**Attaching an HBA Driver**

The driver should attach this instance of the device, and perform error cleanup if necessary.

```c
i = scsi_hba_attach_setup(dip, &isp_dma_attr, tran, 0);
if (i != DDI_SUCCESS) {
    /* do error recovery */
    return (DDI_FAILURE);
}
```

**Register Mapping**

The driver should map in its device's registers. The driver need to specify the following items:

- Register set index
- Data access characteristics of the device
- Size of the register to be mapped

```c
ddi_device_acc_attr_t dev_attributes;

    dev_attributes.devacc_attr_version = DDI_DEVICE_ATTR_V0;
    dev_attributes.devacc_attr_dataorder = DDI_STRICTORDER_ACC;
    dev_attributes.devacc_attr_endian_flags = DDI_STRUCTURE_LE_ACC;

    if (ddi_regs_map_setup(dip, 0, (caddr_t *)&isp->isp_reg,
                      0, sizeof (struct ispregs), &dev_attributes,
                      &isp->isp_acc_handle) != DDI_SUCCESS) {
        /* do error recovery */
        return (DDI_FAILURE);
    }
```

**Adding an Interrupt Handler**

The driver must first obtain the *iblock cookie* to initialize any mutexes that are used in the driver handler. Only after those mutexes have been initialized can the interrupt handler be added.

```c
i = ddi_get_iblock_cookie(dip, 0, &isp->iblock_cookie);
if (i != DDI_SUCCESS) {
    /* do error recovery */
    return (DDI_FAILURE);
}
mutex_init(&isp->mutex, "isp_mutex", MUTEX_DRIVER,
            (void *)&isp->iblock_cookie);
i = ddi_add_intr(dip, 0, &isp->iblock_cookie,
                 0, isp_intr, (caddr_t)isp);
if (i != DDI_SUCCESS) {
```
/* do error recovery */
return (DDI_FAILURE);
}

If a high-level handler is required, the driver should be coded to provide such a handler.
Otherwise, the driver must be able to fail the attach. See “Handling High-Level Interrupts” on
page 147 for a description of high-level interrupt handling.

Create Power Manageable Components

With power management, if the host bus adapter only needs to power down when all target
adapters are at power level 0, the HBA driver only needs to provide a power(9E) entry point.
Refer to Chapter 12, “Power Management.” The HBA driver also needs to create a
pm-components(9P) property that describes the components that the device implements.

Nothing more is necessary, since the components will default to idle, and the power
management framework’s default dependency processing will ensure that the host bus adapter
will be powered up whenever an target adapter is powered up. Provided that automatic power
management is enabled automatically, the processing will also power down the host bus adapter
when all target adapters are powered down ()

Report Attachment Status

Finally, the driver should report that this instance of the device is attached and return success.

ddi_report_dev(dip);
    return (DDI_SUCCESS);

detach() Entry Point (SCSI HBA Drivers)
The driver should perform standard detach operations, including calling
scsi_hba_detach(9F).

Entry Points for SCSA HBA Drivers

An HBA driver can work with target drivers through the SCSA interface. The SCSA interfaces
require the HBA driver to supply a number of entry points that are callable through the
scsi_hba_tran(9S) structure.

These entry points fall into five functional groups:

■ Target driver instance initialization
■ Resource allocation and deallocation
■ Command transport
■ Capability management
■ Abort and reset handling
Dynamic reconfiguration

The following table lists the entry points for SCSA HBA by function groups.

**TABLE 18-3 SCSA Entry Points**

<table>
<thead>
<tr>
<th>Function Groups</th>
<th>Entry Points Within Group</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target Driver Instance Initialization</td>
<td><code>tran_tgt_init(9E)</code></td>
<td>Performs per-target initialization (optional)</td>
</tr>
<tr>
<td></td>
<td><code>tran_tgt_probe(9E)</code></td>
<td>Probes SCSI bus for existence of a target (optional)</td>
</tr>
<tr>
<td></td>
<td><code>tran_tgt_free(9E)</code></td>
<td>Performs per-target deallocation (optional)</td>
</tr>
<tr>
<td>Resource Allocation</td>
<td><code>tran_init_pkt(9E)</code></td>
<td>Allocates SCSI packet and DMA resources</td>
</tr>
<tr>
<td></td>
<td><code>tran_destroy_pkt(9E)</code></td>
<td>Frees SCSI packet and DMA resources</td>
</tr>
<tr>
<td></td>
<td><code>tran_sync_pkt(9E)</code></td>
<td>Synchronizes memory before and after DMA</td>
</tr>
<tr>
<td></td>
<td><code>tran_dma_free(9E)</code></td>
<td>Frees DMA resources</td>
</tr>
<tr>
<td>Command Transport</td>
<td><code>tran_start(9E)</code></td>
<td>Transports a SCSI command</td>
</tr>
<tr>
<td>Capability Management</td>
<td><code>tran_getcap(9E)</code></td>
<td>Inquires about a capability’s value</td>
</tr>
<tr>
<td></td>
<td><code>tran_setcap(9E)</code></td>
<td>Sets a capability’s value</td>
</tr>
<tr>
<td>Abort and Reset</td>
<td><code>tran_abort(9E)</code></td>
<td>Aborts outstanding SCSI commands</td>
</tr>
<tr>
<td></td>
<td><code>tran_reset(9E)</code></td>
<td>Resets a target device or the SCSI bus</td>
</tr>
<tr>
<td></td>
<td><code>tran_bus_reset(9E)</code></td>
<td>Resets the SCSI bus</td>
</tr>
<tr>
<td></td>
<td><code>tran_reset_notify(9E)</code></td>
<td>Request to notify target of bus reset (optional)</td>
</tr>
<tr>
<td>Dynamic Reconfiguration</td>
<td><code>tran_quiesce(9E)</code></td>
<td>Stops activity on the bus</td>
</tr>
<tr>
<td></td>
<td><code>tran_unquiesce(9E)</code></td>
<td>Resumes activity on the bus</td>
</tr>
</tbody>
</table>

**Target Driver Instance Initialization**

The following sections describe target entry points.
**tran_tgt_init() Entry Point**

The `tran_tgt_init()` entry point enables the HBA to allocate and initialize any per-target resources. `tran_tgt_init()` also enables the HBA to qualify the device's address as valid and supportable for that particular HBA. By returning `DDI_FAILURE`, the instance of the target driver for that device is not probed or attached.

`tran_tgt_init()` is not required. If `tran_tgt_init()` is not supplied, the framework attempts to probe and attach all possible instances of the appropriate target drivers.

```c
static int
isp_tran_tgt_init(
    dev_info_t *hba_dip,
    dev_info_t *tgt_dip,
    scsi_hba_tran_t *tran,
    struct scsi_device *sd)
{
    return ((sd->sd_address.a_target < N_ISP_TARGETS_WIDE &&
            sd->sd_address.a_lun < 8) ? DDI_SUCCESS : DDI_FAILURE);
}
```

**tran_tgt_probe() Entry Point**

The `tran_tgt_probe()` entry point enables the HBA to customize the operation of `scsi_probe()`, if necessary. This entry point is called only when the target driver calls `scsi_probe()`.

The HBA driver can retain the normal operation of `scsi_probe()` by calling `scsi_hba_probe()` and returning its return value.

This entry point is not required, and if not needed, the HBA driver should set the `tran_tgt_probe` vector in the `scsi_hba_tran` structure to point to `scsi_hba_probe()`.

`scsi_probe()` allocates a `scsi_inquiry` structure and sets the `sd_inq` field of the `scsi_device` structure to point to the data in `scsi_inquiry`. `scsi_hba_probe()` handles this task automatically. `scsi_unprobe()` then frees the `scsi_inquiry` data.

Except for the allocation of `scsi_inquiry` data, `tran_tgt_probe()` must be stateless, because the same SCSI device might call `tran_tgt_probe()` several times. Normally, allocation of `scsi_inquiry` data is handled by `scsi_hba_probe()`.

**Note** – The allocation of the `scsi_inquiry` structure is handled automatically by `scsi_hba_probe()`. This information is only of concern if you want custom `scsi_probe()` handling.

```c
static int
isp_tran_tgt_probe(
    struct scsi_device *sd,
```
 tran_tgt_free() Entry Point
The tran_tgt_free(9E) entry point enables the HBA to perform any deallocation or clean-up procedures for an instance of a target. This entry point is optional.

static void isp_tran_tgt_free(
    dev_info_t *hba_dip,
    dev_info_t *tgt_dip,
    scsi_hba_tran_t *hba_tran,
    struct scsi_device *sd)
{
    /*
    * Undo any special per-target initialization done
    * earlier in tran_tgt_init(9F) and tran_tgt_probe(9F)
    */
}

Resource Allocation
The following sections discuss resource allocation.

tran_init_pkt() Entry Point
The tran_init_pkt(9E) entry point allocates and initializes a scsi_pkt(9S) structure and DMA resources for a target driver request.

The tran_init_pkt(9E) entry point is called when the target driver calls the SCSA function scsi_init_pkt(9F).

Each call of the tran_init_pkt(9E) entry point is a request to perform one or more of three possible services:

- Allocation and initialization of a scsi_pkt(9S) structure
- Allocation of DMA resources for data transfer
- Reallocation of DMA resources for the next portion of the data transfer

Allocation and Initialization of a scsi_pkt(9S) Structure
The tran_init_pkt(9E) entry point must allocate a scsi_pkt(9S) structure through scsi_hba_pkt_alloc(9F) if pkt is NULL.

scsi_hba_pkt_alloc(9F) allocates space for the following items:
The **scsi_pkt**(9S) structure members, including **pkt**, must be initialized to zero except for the following members:

- **pkt_scbp** – Status completion
- **pkt_cdbp** – CDB
- **pkt_ha_private** – HBA driver private data
- **pkt_private** – Target driver private data

These members are pointers to memory space where the values of the fields are stored, as shown in the following figure. For more information, refer to "**scsi_pkt Structure (HBA)**" on page 339.

**FIGURE 18–5 scsi_pkt(9S) Structure Pointers**

The following example shows allocation and initialization of a **scsi_pkt** structure.

**EXAMPLE 18–2 HBA Driver Initialization of a SCSI Packet Structure**

```c
static struct scsi_pkt
isp_scsi_init_pkt(
    struct scsi_address *ap,
    struct scsi_pkt *pkt,
    struct buf *bp,
    int cmdlen,
```
int statuslen,
int tgtlen,
int flags,
int (*callback)(),
caddr_t arg)
{
    struct isp_cmd *sp;
    struct isp *isp;
    struct scsi_pkt *new_pkt;

    ASSERT(callback == NULL_FUNC || callback == SLEEP_FUNC);

    isp = (struct isp *)__ap->a_hba_tran->tran_hba_private;
    /*
     * First step of isp_scsi_init_pkt: pkt allocation
     */
    if (pkt == NULL) {
        pkt = scsi_hba_pkt_alloc(isp->isp_dip, ap, cmdlen,
                                   statuslen, tgtlen, sizeof (struct isp_cmd),
                                   callback, arg);
        if (pkt == NULL) {
            return (NULL);
        }
    }

    sp = (struct isp_cmd *)pkt->pkt_ha_private;
    /*
     * Initialize the new pkt
     */
    sp->cmd_pkt = pkt;
    sp->cmd_flags = 0;
    sp->cmd_scblen = statuslen;
    sp->cmd_cdblen = cmdlen;
    sp->cmd_dmahandle = NULL;
    sp->cmd_ncookies = 0;
    sp->cmd_cookie = 0;
    sp->cmd_cookiecnt = 0;
    sp->cmd_nwin = 0;
    pkt->pkt_address = *ap;
    pkt->pkt_comp = (void (*)())NULL;
    pkt->pkt_flags = 0;
    pkt->pkt_time = 0;
    pkt->pkt_resid = 0;
    pkt->pkt_statistics = 0;
    pkt->pkt_reason = 0;
    new_pkt = pkt;
} else {
    sp = (struct isp_cmd *)pkt->pkt_ha_private;
    new_pkt = NULL;
}

/*
 * Second step of isp_scsi_init_pkt: dma allocation/move
*/
if ((bp & bp->b_cflag) != 0) {
    if (sp->cmd_dmahandle == NULL) {
        if (isp_i_dma_alloc(isp, pkt, bp, flags, callback) == 0) {
            if (new_pkt) {
EXAMPLE 18–2 HBA Driver Initialization of a SCSI Packet Structure  
(Continued)

```c
    scsi_hba_pkt_free(ap, new_pkt);
    return ((struct scsi_pkt *)NULL);
} else {
    ASSERT(new_pkt == NULL);
    if (isp_i_dma_move(isp, pkt, bp) == 0) {
        return ((struct scsi_pkt *)NULL);
    } else {
        ASSERT(new_pkt == NULL);
        if (isp_i_dma_move(isp, pkt, bp) == 0) {
            return ((struct scsi_pkt *)NULL);
        }
    }
    return (pkt);
}
```

Allocation of DMA Resources

The `tran_init_pkt(9E)` entry point must allocate DMA resources for a data transfer if the following conditions are true:

- `bp` is not null.
- `bp->b_bcount` is not zero.
- DMA resources have not yet been allocated for this `scsi_pkt(9S)`.

The HBA driver needs to track how DMA resources are allocated for a particular command. This allocation can take place with a flag bit or a DMA handle in the per-packet HBA driver private data.

The `PKT_DMA_PARTIAL` flag in the `pkt` enables the target driver to break up a data transfer into multiple SCSI commands to accommodate the complete request. This approach is useful when the HBA hardware scatter-gather capabilities or system DMA resources cannot complete a request in a single SCSI command.

The `PKT_DMA_PARTIAL` flag enables the HBA driver to set the `DDI_DMA_PARTIAL` flag. The `DDI_DMA_PARTIAL` flag is useful when the DMA resources for this SCSI command are allocated. For example the `ddi_dma_buf_bind_handle(9F)` command can be used to allocate DMA resources. The DMA attributes used when allocating the DMA resources should accurately describe any constraints placed on the ability of the HBA hardware to perform DMA. If the system can only allocate DMA resources for part of the request, `ddi_dma_buf_bind_handle(9F)` returns `DDI_DMA_PARTIAL_MAP`.

The `tran_init_pkt(9E)` entry point must return the amount of DMA resources not allocated for this transfer in the field `pkt_resid`.

A target driver can make one request to `tran_init_pkt(9E)` to simultaneously allocate both a `scsi_pkt(9S)` structure and DMA resources for that `pkt`. In this case, if the HBA driver is unable to allocate DMA resources, that driver must free the allocated `scsi_pkt(9S)` before returning. The `scsi_pkt(9S)` must be freed by calling `scsi_hba_pkt_free(9F)`.
The target driver might first allocate the `scsi_pkt(9S)` and allocate DMA resources for this pkt at a later time. In this case, if the HBA driver is unable to allocate DMA resources, the driver must not free pkt. The target driver in this case is responsible for freeing the pkt.

**EXAMPLE 18-3  HBA Driver Allocation of DMA Resources**

```c
static int isp_i_dma_alloc(
    struct isp *isp,
    struct scsi_pkt *pkt,
    struct buf *bp,
    int flags,
    int (*callback)())
{
    struct isp_cmd *sp = (struct isp_cmd *)pkt->pkt_ha_private;
    int dma_flags;
    ddi_dma_attr_t tmp_dma_attr;
    int i;

    ASSERT(callback == NULL_FUNC || callback == SLEEP_FUNC);

    if (bp->b_flags & B_READ) {
        sp->cmd_flags &= ~CFLAG_DMASEND;
        dma_flags = DDI_DMA_READ;
    } else {
        sp->cmd_flags |= CFLAG_DMASEND;
        dma_flags = DDI_DMA_WRITE;
    }
    if (flags & PKT_CONSISTENT) {
        sp->cmd_flags |= CFLAG_CMDIOPB;
        dma_flags |= DDI_DMA_CONSISTENT;
    }
    if (flags & PKT_DMA_PARTIAL) {
        dma_flags |= DDI_DMA_PARTIAL;
    }

    tmp_dma_attr = isp_dma_attr;
    tmp_dma_attr.dma_attr_burstsizes = isp->isp_burst_size;
    cb = (callback == NULL_FUNC) ? DDI_DMA_DONTWAIT : DDI_DMA_SLEEP;

    if ((i = ddi_dma_alloc_handle(isp->isp_dip, &tmp_dma_attr,
        cb, 0, &sp->cmd_dmahandle)) != DDI_SUCCESS) {
        switch (i) {
            case DDI_DMA_BADATTR:
                bioerror(bp, EFAULT);
                return (0);
            case DDI_DMA_NORESOURCES:
                bioerror(bp, 0);
                return (0);
        }
    }

    i = ddi_dma_buf_bind_handle(sp->cmd_dmahandle, bp, dma_flags,
        cb, 0, &sp->cmd_dmacookies[0], &sp->cmd_ncookies);

    switch (i) {
```
case DDI_DMA_PARTIAL_MAP:
    if (ddi_dma_numwin(sp->cmd_dmahandle, &sp->cmd_nwin) == DDI_FAILURE) {
        cmn_err(CE_PANIC, "ddi_dma_numwin() failed\n");
    }

    if (ddi_dma_getwin(sp->cmd_dmahandle, sp->cmd_curwin, 
        &sp->cmd_dma_offset, &sp->cmd_dma_len, &sp->cmd_dmacookies[0], 
        &sp->cmd_ncookies) == DDI_FAILURE) {
        cmn_err(CE_PANIC, "ddi_dma_getwin() failed\n");
    }
    goto get_dma_cookies;

case DDI_DMA_MAPPED:
    sp->cmd_nwin = 1;
    sp->cmd_dma_len = 0;
    sp->cmd_dma_offset = 0;

get_dma_cookies:
    i = 0;
    sp->cmd_dmacount = 0;
    for (;;) {
        sp->cmd_dmacount += sp->cmd_dmacookies[i++].dmac_size;
        if (i == ISP_NDATASEGS || i == sp->cmd_ncookies) 
            break;
        ddi_dma_nextcookie(sp->cmd_dmahandle, 
            &sp->cmd_dmacookies[i]);
    }

    sp->cmd_cookie = i;
    sp->cmd_cookiecnt = i;
    sp->cmd_flags |= CFLAG_DMAVALID;
    pkt->pkt_resid = bp->b_bcount - sp->cmd_dmacount;
    return (1);

case DDI_DMA_NORESOURCES:
    bioerror(bp, 0);
    break;

case DDI_DMA_NOMAPPING:
    bioerror(bp, EFAULT);
    break;

case DDI_DMA_TOOBIG:
    bioerror(bp, EINVAL);
    break;

case DDI_DMA_INUSE:
    cmn_err(CE_PANIC, "ddi_dma_buf_bind_handle:" 
        "DDI_DMA_INUSE impossible\n");
    default:
        cmn_err(CE_PANIC, "ddi_dma_buf_bind_handle:" 
            "0x%lx impossible\n", i);
    }

    ddi_dma_free_handle(&sp->cmd_dmahandle);
    sp->cmd_dmahandle = NULL;
    sp->cmd_flags &= ~CFLAG_DMAVALID;
Reallocation of DMA Resources for Data Transfer

For a previously allocated packet with data remaining to be transferred, the `tran_init_pkt(9E)` entry point must reallocate DMA resources when the following conditions apply:

- Partial DMA resources have already been allocated.
- A non-zero `pkt_resid` was returned in the previous call to `tran_init_pkt(9E)`.
- `bp` is not null.
- `bp->b_bcount` is not zero.

When reallocating DMA resources to the next portion of the transfer, `tran_init_pkt(9E)` must return the amount of DMA resources not allocated for this transfer in the field `pkt_resid`.

If an error occurs while attempting to move DMA resources, `tran_init_pkt(9E)` must not free the `scsi_pkt(9S)`. The target driver in this case is responsible for freeing the packet.

If the callback parameter is `NULL_FUNC`, the `tran_init_pkt(9E)` entry point must not sleep or call any function that might sleep. If the callback parameter is `SLEEP_FUNC` and resources are not immediately available, the `tran_init_pkt(9E)` entry point should sleep. Unless the request is impossible to satisfy, `tran_init_pkt()` should sleep until resources become available.

```c
static int isp_i_dma_move(
    struct isp *isp,
    struct scsi_pkt *pkt,
    struct buf *bp)
{
    struct isp_cmd *sp = (struct isp_cmd *)pkt->pkt_ha_private;
    int i;

    ASSERT(sp->cmd_flags & CFLAG_COMPLETED);
    sp->cmd_flags &= ~CFLAG_COMPLETED;
    /*
     * If there are no more cookies remaining in this window,
     * must move to the next window first.
     */
    if (sp->cmd_cookie == sp->cmd_ncookies) {
        /*
         * For small pkts, leave things where they are
         */
        if (sp->cmd_curwin == sp->cmd_nwin && sp->cmd_nwin == 1)
            return (1);
        /*
         * At last window, cannot move
         */
        if (++sp->cmd_curwin >= sp->cmd_nwin)
```
ByExample for SCSA HBA Drivers

EXAMPLE 18–4 DMA Resource Reallocation for HBA Drivers (Continued)

```c
return (0);
if (ddi_dma_getwin(sp->cmd_dmahandle, sp->cmd_curwin,
    &sp->cmd_dma_offset, &sp->cmd_dma_len,
    &sp->cmd_dmacookies[0], &sp->cmd_ncookies) == DDI_FAILURE)
    return (0);
sp->cmd_cookie = 0;
} else {
    /*
    * Still more cookies in this window - get the next one
    */
    ddi_dma_nextcookie(sp->cmd_dmahandle, &sp->cmd_dmacookies[0]);
}
/*
* Get remaining cookies in this window, up to our maximum
*/
i = 0;
for (;;) {
    sp->cmd_dmacount += sp->cmd_dmacookies[i++].dmac_size;
    sp->cmd_cookie++;
    if (i == ISP_NDATASEGS || sp->cmd_cookie == sp->cmd_ncookies)
        break;
    ddi_dma_nextcookie(sp->cmd_dmahandle, &sp->cmd_dmacookies[i]);
}
sp->cmd_cookiecnt = i;
pkt->pkt_resid = bp->b_bcount - sp->cmd_dmacount;
return (1);
}
```

tran_destroy_pkt() Entry Point

The `tran_destroy_pkt()` entry point is the HBA driver function that deallocates `scsi_pkt` structures. The `tran_destroy_pkt()` entry point is called when the target driver calls `scsi_destroy_pkt(9F)`.

The `tran_destroy_pkt()` entry point must free any DMA resources that have been allocated for the packet. An implicit DMA synchronization occurs if the DMA resources are freed and any cached data remains after the completion of the transfer. The `tran_destroy_pkt()` entry point frees the SCSI packet by calling `scsi_hba_pkt_free(9F)`.

EXAMPLE 18–5 HBA Driver tran_destroy_pkt(9E) Entry Point

```c
static void
isp_scsi_destroy_pkt(
    struct scsi_address *ap,
    struct scsi_pkt *pkt)
{
    struct isp_cmd *sp = (struct isp_cmd *)pkt->pkt_ha_private;
    /*
    * Free the DMA, if any
    */
    if (sp->cmd_flags & CFLAG_DMAVALID) {
        sp->cmd_flags &= ~CFLAG_DMAVALID;
        (void) ddi_dma_unbind_handle(sp->cmd_dmahandle);
```


**tran_destroy_pkt() Entry Point**

The `tran_destroy_pkt()` entry point synchronizes the DMA object allocated for the `scsi_pkt` structure before or after a DMA transfer. The `tran_destroy_pkt()` entry point is called when the target driver calls `scsi_destroy_pkt()`. 

If the data transfer direction is a DMA read from device to memory, `tran_destroy_pkt()` must synchronize the CPU's view of the data. If the data transfer direction is a DMA write from memory to device, `tran_destroy_pkt()` must synchronize the device's view of the data.

**tran_sync_pkt() Entry Point**

The `tran_sync_pkt()` entry point synchronizes the DMA object allocated for the `scsi_pkt` structure before or after a DMA transfer. The `tran_sync_pkt()` entry point is called when the target driver calls `scsi_sync_pkt()`.

If the data transfer direction is a DMA read from device to memory, `tran_sync_pkt()` must synchronize the CPU's view of the data. If the data transfer direction is a DMA write from memory to device, `tran_sync_pkt()` must synchronize the device's view of the data.

**tran_dmafree() Entry Point**

The `tran_dmafree()` entry point deallocates DMA resources that have been allocated for a `scsi_pkt` structure. The `tran_dmafree()` entry point is called when the target driver calls `scsi_dmafree()`. 

`tran_dmafree()` must free only DMA resources allocated for a `scsi_pkt` structure, not the `scsi_pkt` itself. When DMA resources are freed, a DMA synchronization is implicitly performed.
Note – The `scsi_pkt(9S)` is freed in a separate request to `tran_destroy_pkt(9E)`. Because `tran_destroy_pkt()` must also free DMA resources, the HBA driver must keep accurate note of whether `scsi_pkt()` structures have DMA resources allocated.

EXAMPLE 18–7 HBA Driver `tran_dmafree(9E)` Entry Point

```c
static void
isp_scsi_dmafree(
    struct scsi_address  *ap,
    struct scsi_pkt      *pkt)
{
    struct isp_cmd       *sp = (struct isp_cmd *)pkt->pkt_ha_private;

    if (sp->cmd_flags & CFLAG_DMAVALID) {
        sp->cmd_flags &= ~CFLAG_DMAVALID;
        (void)ddi_dma_unbind_handle(sp->cmd_dmahandle);
        ddi_dma_free_handle(&sp->cmd_dmahandle);
        sp->cmd_dmahandle = NULL;
    }
}
```

Command Transport

An HBA driver goes through the following steps as part of command transport:

1. Accept a command from the target driver.
2. Issue the command to the device hardware.
3. Service any interrupts that occur.
4. Manage time outs.

`tran_start()` Entry Point

The `tran_start(9E)` entry point for a SCSI HBA driver is called to transport a SCSI command to the addressed target. The SCSI command is described entirely within the `scsi_pkt(9S)` structure, which the target driver allocated through the HBA driver’s `tran_init_pkt(9E)` entry point. If the command involves a data transfer, DMA resources must also have been allocated for the `scsi_pkt(9S)` structure.

The `tran_start()` entry point is called when a target driver calls `scsi_transport(9F)`.

`tran_start()` should perform basic error checking along with any initialization that is required by the command. The `FLAG_NOINTR` flag in the `pkt_flags` field of the `scsi_pkt(9S)` structure can affect the behavior of `tran_start()`. If `FLAG_NOINTR` is not set, `tran_start()` must queue the command for execution on the hardware and return immediately. Upon completion of the command, the HBA driver should call the `pkt` completion routine.

If the `FLAG_NOINTR` is set, then the HBA driver should not call the `pkt` completion routine.
The following example demonstrates how to handle the `tran_start(9E)` entry point. The ISP hardware provides a queue per-target device. For devices that can manage only one active outstanding command, the driver is typically required to manage a per-target queue. The driver then starts up a new command upon completion of the current command in a round-robin fashion.

**EXAMPLE 18-8  HBA Driver tran_start(9E) Entry Point**

```c
static int isp_scsi_start(
    struct scsi_address *ap,
    struct scsi_pkt *pkt)
{
    struct isp_cmd *sp;
    struct isp *isp;
    struct isp_request *req;
    u_long cur_lbolt;
    int xfercount;
    int i;

    sp = (struct isp_cmd *)pkt->pkt_ha_private;
    isp = (struct isp *)ap->a_hba_tran->tran_hba_private;

    sp->cmd_flags = (sp->cmd_flags & ~CFLAG_TRANFLAG) |
        CFLAG_IN_TRANSPORT;
    pkt->pkt_reason = CMD_CMPLT;
    /*
     * set up request in cmd_isp_request area so it is ready to
     * go once we have the request mutex
     */
    req = &sp->cmd_isp_request;

    req->req_header.cq_entry_type = CQ_TYPE_REQUEST;
    req->req_header.cq_entry_count = 1;
    req->req_header.cq_flags = 0;
    req->req_header.cq_seqno = 0;
    req->req_reserved = 0;
    req->req_token = (opaque_t)sp;
    req->req_target = TGT(sp);
    req->req_lun_trn = LUN(sp);
    ISP_SET_PKT_FLAGS(pkt->pkt_flags, req->req_flags);
    /*
     * Set up data segments for dma transfers.
     */
    if (sp->cmd_flags & CFLAG_DMAVALID) {
        if (sp->cmd_flags & CFLAG_CMDIOPB) {
            (void) ddi_dma_sync(sp->cmd_dmahandle,
                sp->cmd_dma_offset, sp->cmd_dma_len,
                DDI_DMA_SYNC_FORDEV);
        }
        ASSERT(sp->cmd_cookiecnt > 0 &&
            sp->cmd_cookiecnt <= ISP_NDATASEGS);
        xfercount = 0;
    }
}
```

---

**Entry Points for SCISA HBA Drivers**

Chapter 18 • SCSI Host Bus Adapter Drivers 363
EXAMPLE 18–8  HBA Driver tran_start(9E) Entry Point  (Continued)

```c
req->req_seg_count = sp->cmd_cookiecnt;
for (i = 0; i < sp->cmd_cookiecnt; i++) {
    req->req_dataseg[i].d_count =
        sp->cmd_dmacookies[i].dmac_size;
    req->req_dataseg[i].d_base =
        sp->cmd_dmacookies[i].dmac_address;
    xfercount +=
        sp->cmd_dmacookies[i].dmac_size;
}
for (; i < ISP_NDATASEGS; i++) {
    req->req_dataseg[i].d_count = 0;
    req->req_dataseg[i].d_base = 0;
}
 pkt->pkt_resid = xfercount;
if (sp->cmd_flags & CFLAG_DMASEND) {
    req->req_flags |= ISP_REQ_FLAG_DATA_WRITE;
} else {
    req->req_flags |= ISP_REQ_FLAG_DATA_READ;
}
} else {
    req->req_seg_count = 0;
    req->req_dataseg[0].d_count = 0;
}
/*
 * Set up cdb in the request
 */
 req->req_cdblen = sp->cmd_cdblen;
bcopy((caddr_t)pkt->pkt_cdbp, (caddr_t)req->req_cdb,
     sp->cmd_cdblen);
/*
 * Start the cmd. If NO_INTR, must poll for cmd completion.
 */
 if ((pkt->pkt_flags & FLAG_NOINTR) == 0) {
    mutex_enter(ISP_REQ_MUTEX(isp));
    rval = isp_i_start_cmd(isp, sp);
    mutex_exit(ISP_REQ_MUTEX(isp));
} else {
    rval = isp_i_polled_cmd_start(isp, sp);
}
return (rval);
}
```

**Interrupt Handler and Command Completion**

The interrupt handler must check the status of the device to be sure the device is generating the
interrupt in question. The interrupt handler must also check for any errors that have occurred
and service any interrupts generated by the device.

If data is transferred, the hardware should be checked to determine how much data was actually
transferred. The pkt_resid field in the `scsi_pkt(9S)` structure should be set to the residual of
the transfer.
Commands that are marked with the PKT_CONSISTENT flag when DMA resources are allocated through `tran_init_pkt(9E)` take special handling. The HBA driver must ensure that the data transfer for the command is correctly synchronized before the target driver’s command completion callback is performed.

Once a command has completed, you need to act on two requirements:

- If a new command is queued up, start the command on the hardware as quickly as possible.
- Call the command completion callback. The callback has been set up in the `scsi_pkt(9S)` structure by the target driver to notify the target driver when the command is complete.

Start a new command on the hardware, if possible, before calling the PKT_COMP command completion callback. The command completion handling can take considerable time. Typically, the target driver calls functions such as `biodone(9F)` and possibly `scsi_transport(9F)` to begin a new command.

The interrupt handler must return `DDI_INTRCLAIMED` if this interrupt is claimed by this driver. Otherwise, the handler returns `DDI_INTR_UNCLAIMED`.

The following example shows an interrupt handler for the SCSI HBA `isp` driver. The `caddr_t` parameter is set up when the interrupt handler is added in `attach(9E)`. This parameter is typically a pointer to the state structure, which is allocated on a per instance basis.

**EXAMPLE 18-9  HBA Driver Interrupt Handler**

```c
static u_int
isp_intr(caddr_t arg)
{
    struct isp_cmd     *sp;
    struct isp_cmd     *head, *tail;
    u_short            response_in;
    struct isp_response *resp;
    struct isp         *isp = (struct isp *)arg;
    struct isp_slot    *isp_slot;
    int                 n;

    if (ISP_INT_PENDING(isp) == 0) {
        return (DDI_INTR_UNCLAIMED);
    }

    do {
        again:
        /*
         * head list collects completed packets for callback later
         */
        head = tail = NULL;
        /*
         * Assume no mailbox events (e.g., mailbox cmds, asynch
         * events, and isp dma errors) as common case.
         */
        if (ISP_CHECK_SEMAPHORE_LOCK(isp) == 0) {
            mutex_enter(ISP_RESP_MUTEX(isp));
            /*
            */
            ...
* Loop through completion response queue and post completed pkts. Check response queue again afterwards in case there are more.

```c
isp->isp_response_in = response_in = ISP_GET_RESPONSE_IN(isp);
/*
* Calculate the number of requests in the queue
*/
int n = response_in - isp->isp_response_out;
if (n < 0) {
    n = ISP_MAX_REQUESTS - 
        isp->isp_response_out + response_in;
}
while (n-- > 0) {
    ISP_GET_NEXT_RESPONSE_OUT(isp, resp);
    struct isp_cmd *sp = (struct isp_cmd *)resp->resp_token;
    /*
    * Copy over response packet in sp
    */
    isp_i_get_response(isp, resp, sp);
}
if (head) {
    tail->cmd_forw = sp;
    tail = sp;
    tail->cmd_forw = NULL;
} else {
    tail = head = sp;
    sp->cmd_forw = NULL;
}
ISP_SET_RESPONSE_OUT(isp);
ISP_CLEAR_RISC_INT(isp);
mutex_exit(ISP_RESP_MUTEX(isp));
if (head) {
    isp_i_call_pkt_comp(isp, head);
} else {
    if (isp_i_handle_mbox_cmd(isp) != ISP_AEN_SUCCESS) {
        return (DDI_INTR_CLAIMED);
    }
    /*
    * if there was a reset then check the response
    * queue again
    */
    goto again;
}
```

} while (ISP_INT_PENDING(isp));

return (DDI_INTR_CLAIMED);

static void
isp_i_call_pkt_comp(
    struct isp        *isp,
Timeout Handler

The HBA driver is responsible for enforcing time outs. A command must be complete within a specified time unless a zero time out has been specified in the `scsi_pkt` structure.

When a command times out, the HBA driver should mark the `scsi_pkt` with `pkt_reason` set to `CMD_TIMEOUT` and `pkt_statistics` OR'd with `STAT_TIMEOUT`. The HBA driver should also
attempt to recover the target and bus. If this recovery can be performed successfully, the driver should mark the `scsi_pkt(9S)` using `pkt_statistics OR’d with either STAT_BUS_RESET or STAT_DEV_RESET`.

After the recovery attempt has completed, the HBA driver should call the command completion callback.

**Note** – If recovery was unsuccessful or not attempted, the target driver might attempt to recover from the timeout by calling `scsi_reset(9F)`.

The ISP hardware manages command timeout directly and returns timed-out commands with the necessary status. The timeout handler for the `isp` sample driver checks active commands for the time out state only once every 60 seconds.

The `isp` sample driver uses the `timeout(9F)` facility to arrange for the kernel to call the timeout handler every 60 seconds. The `caddr_t` argument is the parameter set up when the timeout is initialized at `attach(9E)` time. In this case, the `caddr_t` argument is a pointer to the state structure allocated per driver instance.

If timed-out commands have not been returned as timed-out by the ISP hardware, a problem has occurred. The hardware is not functioning correctly and needs to be reset.

## Capability Management

The following sections discuss capability management.

### `tran_getcap()` Entry Point

The `tran_getcap(9E)` entry point for a SCSI HBA driver is called by `scsi_ifgetcap(9F)`. The target driver calls `scsi_ifgetcap()` to determine the current value of one of a set of SCSI-defined capabilities.

The target driver can request the current setting of the capability for a particular target by setting the `whom` parameter to nonzero. A `whom` value of zero indicates a request for the current setting of the general capability for the SCSI bus or for adapter hardware.

The `tran_getcap()` entry point should return `-1` for undefined capabilities or the current value of the requested capability.

The HBA driver can use the function `scsi_hba_lookup_capstr(9F)` to compare the capability string against the canonical set of defined capabilities.

**EXAMPLE 18–10**  HBA Driver `tran_getcap(9E)` Entry Point

```c
static int
isp_scsi_getcap(
    struct scsi_address *ap,
```
EXAMPLE 18–10  HBA Driver tran_getcap(9E) Entry Point  (Continued)

```c
char *cap,
int whom)
{
struct isp *isp;
int rval = 0;
u_char tgt = ap->a_target;
/*
 * We don’t allow getting capabilities for other targets
 */
if (cap == NULL || whom == 0) {
    return (-1);
}
isp = (struct isp *)ap->a_hba_tran->tran_hba_private;
ISP_MUTEX_ENTER(isp);
switch (scsi_hba_lookup_capstr(cap)) {
    case SCSI_CAP_DMA_MAX:
        rval = 1 << 24; /* Limit to 16MB max transfer */
        break;
    case SCSI_CAP_MSG_OUT:
        rval = 1;
        break;
    case SCSI_CAP_DISCONNECT:
        if (((isp->isp_target_scsi_options[tgt] &
            SCSI_OPTIONS_DR) == 0) {
            break;
        } else if (
            (isp->isp_cap[tgt] & ISP_CAP_DISCONNECT) == 0) {
            break;
        }
        rval = 1;
        break;
    case SCSI_CAP_SYNCHRONOUS:
        if (((isp->isp_target_scsi_options[tgt] &
            SCSI_OPTIONS_SYNC) == 0) {
            break;
        } else if (
            (isp->isp_cap[tgt] & ISP_CAP_SYNC) == 0) {
            break;
        }
        rval = 1;
        break;
    case SCSI_CAP_WIDE_XFER:
        if (((isp->isp_target_scsi_options[tgt] &
            SCSI_OPTIONS_WIDE) == 0) {
            break;
        } else if (
            (isp->isp_cap[tgt] & ISP_CAP_WIDE) == 0) {
            break;
        }
        rval = 1;
        break;
    case SCSI_CAP_TAGGED_QING:
        if (((isp->isp_target_scsi_options[tgt] &
            SCSI_OPTIONS_DR) == 0 ||
            (isp->isp_target_scsi_options[tgt] &
```
The `tran_setcap()` entry point for a SCSI HBA driver is called by `scsi_ifsetcap()`. A target driver calls `scsi_ifsetcap()` to change the current one of a set of SCSA-defined capabilities.

The target driver might request that the new value be set for a particular target by setting the `whom` parameter to nonzero. A `whom` value of zero means the request is to set the new value for the SCSI bus or for adapter hardware in general.

`tran_setcap()` should return the following values as appropriate:

- -1 for undefined capabilities
if the HBA driver cannot set the capability to the requested value
1 if the HBA driver is able to set the capability to the requested value

The HBA driver can use the function `scsi_hba_lookup_capstr(9F)` to compare the capability string against the canonical set of defined capabilities.

**EXAMPLE 18-11** HBA Driver `tran_setcap(9E)` Entry Point

```c
static int
isp_scsi_setcap(
    struct scsi_address *ap,
    char *cap,
    int value,
    int whom)
{
    struct isp   *isp;
    int rval = 0;
    u_char      tgt = ap->a_target;
    int         update_isp = 0;
    /*
     * We don't allow setting capabilities for other targets
     */
    if (cap == NULL || whom == 0) {
        return (-1);
    }
    isp = (struct isp *)ap->a_hba_tran->tran_hba_private;
    ISP_MUTEX_ENTER(isp);
    switch (scsi_hba_lookup_capstr(cap)) {
    case SCSI_CAP_DMA_MAX:
    case SCSI_CAP_MSG_OUT:
    case SCSI_CAP_PARITY:
    case SCSI_CAP_UNTAGGED_QING:
    case SCSI_CAP_LINKED_CMDS:
    case SCSI_CAP_RESET_NOTIFICATION:
        /*
         * None of these are settable through
         * the capability interface.
         */
        break;
    case SCSI_CAP_DISCONNECT:
        if ((isp->isp_target_scsi_options[tgt] &
             SCSI_OPTIONS_DR) == 0) {
            break;
        } else {
            if (value) {
                isp->isp_cap[tgt] |= ISP_CAP_DISCONNECT;
            } else {
                isp->isp_cap[tgt] &= ~ISP_CAP_DISCONNECT;
            }
        }
        rval = 1;
        break;
    case SCSI_CAP_SYNCHRONOUS:
        if ((isp->isp_target_scsi_options[tgt] &
             SCSI_OPTIONS_SYNC) == 0) {
            break;
        } else {
            isp->isp_cap[tgt] |= ISP_CAP_SYNCHRONOUS;
            if ((isp->isp_target_scsi_options[tgt] &
                 SCSI_OPTIONS_SYNC) == 0) {
                break;
            } else {
                isp->isp_cap[tgt] &= ~ISP_CAP_SYNCHRONOUS;
            }
        }
        break;
    default:
        isp->isp_cap[tgt] = cap;
        rval = 1;
        break;
    }
    ISP_MUTEX_EXIT(isp);
    return rval;
}
```

Chapter 18 • SCSI Host Bus Adapter Drivers
break;
} else {
  if (value) {
    isp->isp_cap[tgt] |= ISP_CAP_SYNC;
  } else {
    isp->isp_cap[tgt] &= ~ISP_CAP_SYNC;
  }
}
rvval = 1;
break;
}
}
}
}
}
rvval = 1;
break;
}
}
rvval = 1;
break;
}
}
rvval = 1;
break;
}
}
rvval = 1;
break;
}
}
rvval = 1;
break;
}
}
rvval = 1;
break;
}
}
rvval = 1;
break;
}
}
rvval = 1;
break;
}
}
rvval = 1;
break;
}
is->isp_cap[tgt] |= ISP_CAP_AUTOSENSE;
} else {
  isp->isp_cap[tgt] &= ~ISP_CAP_AUTOSENSE;
}
  rval = 1;
  break;
default:
  rval = -1;
  break;
}
ISP_MUTEX_EXIT(isp);
return (rval);

Abort and Reset Management

The following sections discuss the abort and reset entry points for SCSI HBA.

**tran_abort() Entry Point**

The `tran_abort` entry point for a SCSI HBA driver is called to abort any commands that are currently in transport for a particular target. This entry point is called when a target driver calls `scsi_abort`.

The `tran_abort()` entry point should attempt to abort the command denoted by the `pkt` parameter. If the `pkt` parameter is `NULL`, `tran_abort()` should attempt to abort all outstanding commands in the transport layer for the particular target or logical unit.

Each command successfully aborted must be marked with `pkt_reason CMD_ABORTED` and `pkt_statistics OR'd with STAT_ABORTED`.

**tran_reset() Entry Point**

The `tran_reset` entry point for a SCSI HBA driver is called to reset either the SCSI bus or a particular SCSI target device. This entry point is called when a target driver calls `scsi_reset`.

The `tran_reset()` entry point must reset the SCSI bus if level is `RESET_ALL`. If level is `RESET_TARGET`, just the particular target or logical unit must be reset.

Active commands affected by the reset must be marked with `pkt_reason CMD_RESET`. The type of reset determines whether `STAT_BUS_RESET` or `STAT_DEV_RESET` should be used to OR `pkt_statistics`.

Commands in the transport layer, but not yet active on the target, must be marked with `pkt_reason CMD_RESET`, and `pkt_statistics OR'd with STAT_ABORTED`. 
**tran_bus_reset() Entry Point**

`tran_bus_reset(9E)` must reset the SCSI bus without resetting targets.

```
#include <sys/scsi/scsi.h>

int tran_bus_reset(dev_info_t *hba-dip, int level);
```

where:

* `hba-dip` Pointer associated with the SCSI HBA

* `level` Must be set toRESET_BUS so that only the SCSI bus is reset, not the targets

The `tran_bus_reset()` vector in the `scsi_hba_tran(9S)` structure should be initialized during the HBA driver's `attach(9E)`. The vector should point to an HBA entry point that is to be called when a user initiates a bus reset.

Implementation is hardware specific. If the HBA driver cannot reset the SCSI bus without affecting the targets, the driver should fail `RESET_BUS` or not initialize this vector.

**tran_reset_notify() Entry Point**

Use the `tran_reset_notify(9E)` entry point when a SCSI bus reset occurs. This function requests the SCSI HBA driver to notify the target driver by callback.

**EXAMPLE 18–12** HBA Driver `tran_reset_notify(9E)` Entry Point

```c
isp_scsi_reset_notify(
    struct scsi_address *ap,
    int flag,
    void (*callback)(caddr_t),
    caddr_t arg)
{
    struct isp *isp;
    struct isp_reset_notify_entry *p, *beforep;
    int rval = DDI_FAILURE;

    isp = (struct isp *)ap->a_hba_tran->tran_hba_private;
    mutex_enter(ISP_REQ_MUTEX(isp));
    /*
     * Try to find an existing entry for this target
     */
    p = isp->isp_reset_notify_listf;
    beforep = NULL;

    while (p) {
        if (p->ap == ap)
            break;
        beforep = p;
        p = p->next;
    }

    if ((flag & SCSI_RESET_CANCEL) && (p != NULL)) {
```
if (beforep == NULL) {
    isp->isp_reset_notify_listf = p->next;
} else {
    beforep->next = p->next;
}
kmem_free((caddr_t)p, sizeof (struct isp_reset_notify_entry));
rval = DDI_SUCCESS;
} else if ((flag & SCSI_RESET_NOTIFY) && (p == NULL)) {
    p = kmem_zalloc(sizeof (struct isp_reset_notify_entry),
        KM_SLEEP);
p->ap = ap;
p->callback = callback;
p->arg = arg;
p->next = isp->isp_reset_notify_listf;
isp->isp_reset_notify_listf = p;
rval = DDI_SUCCESS;
}
mutex_exit(ISP_REQ_MUTEX(isp));
return (rval);
}

Dynamic Reconfiguration

To support the minimal set of hot-plugging operations, drivers might need to implement support for bus quiesce, bus unquiesce, and bus reset. The scsi_hba_tran(9S) structure supports these operations. If quiesce, unquiesce, or reset are not required by hardware, no driver changes are needed.

The scsi_hba_tran structure includes the following fields:

int (*tran_quiesce)(dev_info_t *hba-dip);
int (*tran_unquiesce)(dev_info_t *hba-dip);
int (*tran_bus_reset)(dev_info_t *hba-dip, int level);

These interfaces quiesce and unquiesce a SCSI bus.

#include <sys/scsi/scsi.h>

int prefixtran_quiesce(dev_info_t *hba-dip);
int prefixtran_unquiesce(dev_info_t *hba-dip);

tran_quiesce(9E) and tran_unquiesce(9E) are used for SCSI devices that are not designed for hot-plugging. These functions must be implemented by an HBA driver to support dynamic reconfiguration (DR).

The tran_quiesce() and tran_unquiesce() vectors in the scsi_hba_tran(9S) structure should be initialized to point to HBA entry points during attach(9E). These functions are called when a user initiates quiesce and unquiesce operations.
The `tran_quiesce()` entry point stops all activity on a SCSI bus prior to and during the reconfiguration of devices that are attached to the SCSI bus. The `tran_unquiesce()` entry point is called by the SCSA framework to resume activity on the SCSI bus after the reconfiguration operation has been completed.

HBA drivers are required to handle `tran_quiesce()` by waiting for all outstanding commands to complete before returning success. After the driver has quiesced the bus, any new I/O requests must be queued until the SCSA framework calls the corresponding `tran_unquiesce()` entry point.

HBA drivers handle calls to `tran_unquiesce()` by starting any target driver I/O requests in the queue.

## SCSI HBA Driver Specific Issues

The section covers issues specific to SCSI HBA drivers.

### Installing HBA Drivers

A SCSI HBA driver is installed in similar fashion to a leaf driver. See Chapter 22, "Compiling, Loading, Packaging, and Testing Drivers." The difference is that the `add_drv(1M)` command must specify the driver class as SCSI, such as:

```
# add_drv -m "* 0666 root root" -i "pci1077,1020" -c scsi isp
```

### HBA Configuration Properties

When attaching an instance of an HBA device, `scsi_hba_attach_setup(9F)` creates a number of SCSI configuration properties for that HBA instance. A particular property is created only if no existing property of the same name is already attached to the HBA instance. This restriction avoids overriding any default property values in an HBA configuration file.

An HBA driver must use `ddi_prop_get_int(9F)` to retrieve each property. The HBA driver then modifies or accepts the default value of the properties to configure its specific operation.

**scsi-reset-delay Property**

The `scsi-reset-delay` property is an integer specifying the recovery time in milliseconds for a reset delay by either a SCSI bus or SCSI device.
**scsi-options Property**

The `scsi-options` property is an integer specifying a number of options through individually defined bits:

- **SCSI_OPTIONS_DR (0x008)** – If not set, the HBA should not grant disconnect privileges to a target device.
- **SCSI_OPTIONS_LINK (0x010)** – If not set, the HBA should not enable linked commands.
- **SCSI_OPTIONS_SYNC (0x020)** – If not set, the HBA driver must not negotiate synchronous data transfer. The driver should reject any attempt to negotiate synchronous data transfer initiated by a target.
- **SCSI_OPTIONS_PARITY (0x040)** – If not set, the HBA should run the SCSI bus without parity.
- **SCSI_OPTIONS_TAG (0x080)** – If not set, the HBA should not operate in Command Tagged Queuing mode.
- **SCSI_OPTIONS_FAST (0x100)** – If not set, the HBA should not operate the bus in FAST SCSI mode.
- **SCSI_OPTIONS_WIDE (0x200)** – If not set, the HBA should not operate the bus in WIDE SCSI mode.

**Per-Target scsi-options**

An HBA driver might support a per-target `scsi-options` feature in the following format:

```
target<n>-scsi-options=<hex value>
```

In this example, `<n>` is the target ID. If the per-target `scsi-options` property is defined, the HBA driver uses that value rather than the per-HBA driver instance `scsi-options` property. This approach can provide more precise control if, for example, synchronous data transfer needs to be disabled for just one particular target device. The per-target `scsi-options` property can be defined in the `driver.conf(4)` file.

The following example shows a per-target `scsi-options` property definition to disable synchronous data transfer for target device 3:

```
target3-scsi-options=0x2d8
```

**x86 Target Driver Configuration Properties**

Some x86 SCSI target drivers, such as the driver for `cmdk disk`, use the following configuration properties:

- `disk`
If you use the cmdk sample driver to write an HBA driver for an x86 platform, any appropriate properties must be defined in the `driver.conf` file.

Note – These property definitions should appear only in an HBA driver’s `driver.conf` file. The HBA driver itself should not inspect or attempt to interpret these properties in any way. These properties are advisory only and serve as an adjunct to the cmdk driver. The properties should not be relied upon in any way. The property definitions might not be used in future releases.

The `disk` property can be used to define the type of disk supported by cmdk. For a SCSI HBA, the only possible value for the `disk` property is:

- `disk="scdk"` – Disk type is a SCSI disk

The `queue` property defines how the disk driver sorts the queue of incoming requests during strategy (9E). Two values are possible:

- `queue="qsort"` – One-way elevator queuing model, provided by `disksort(9F)`
- `queue="qfifo"` – FIFO, that is, first in, first out queuing model

The `flow_control` property defines how commands are transported to the HBA driver. Three values are possible:

- `flow_control="dsngl"` – Single command per HBA driver
- `flow_control="dmult"` – Multiple commands per HBA driver. When the HBA queue is full, the driver returns `TRAN_BUSY`.
- `flow_control="duplx"` – The HBA can support separate read and write queues, with multiple commands per queue. FIFO ordering is used for the write queue. The queuing model that is used for the read queue is described by the `queue` property. When an HBA queue is full, the driver returns `TRAN_BUSY`.

The following example is a `driver.conf` file for use with an x86 HBA PCI device that has been designed for use with the cmdk sample driver:

```bash
# config file for ISP 1020 SCSI HBA driver
#
 flow_control="dsngl" queue="qsort" disk="scdk"
 scsi-initiator-id=7;
```
Support for Queuing

For a definition of tagged queuing, refer to the SCSI-2 specification. To support tagged queuing, first check the scsi_options flag SCSI_OPTIONS_TAG to see whether tagged queuing is enabled globally. Next, check to see whether the target is a SCSI-2 device and whether the target has tagged queuing enabled. If these conditions are all true, attempt to enable tagged queuing by using scsi_ifsetcap(9F).

If tagged queuing fails, you can attempt to set untagged queuing. In this mode, you submit as many commands as you think necessary or optimal to the host adapter driver. Then the host adapter queues the commands to the target one command at a time, in contrast to tagged queuing. In tagged queuing, the host adapter submits as many commands as possible until the target indicates that the queue is full.
Drivers for Network Devices

Over the years several, slightly different NIC architectures have emerged. The MAC layer is the common Oracle Solaris framework which interacts with the NIC hardware. The MAC layer needs to be able to take advantage of as many hardware features as possible, such as hardware classification, VLAN tagging, VLAN stripping, hardware checksum offload, large segment offload, load spreading and so on, while providing a common model which is applicable by different types of hardware.

To write a network driver for the Oracle Solaris OS, use the Solaris Generic LAN Driver (GLD) framework.

- For new Ethernet drivers, use the GLDv3 framework. See “GLDv3 Network Device Driver Framework” on page 381. The GLDv3 framework is a function calls-based interface.
- To maintain older Ethernet, Token Ring, or FDDI drivers, use the GLDv2 framework. See “GLDv2 Network Device Driver Framework” on page 403. The GLDv2 is a kernel module that provides common code for drivers to share.
- If you are writing a NIC driver, see also Chapter 21, “SR-IOV Drivers.”

**GLDv3 Network Device Driver Framework**

The GLDv3 framework is a function calls-based interface of MAC plugins and MAC driver service routines and structures. The GLDv3 framework implements the necessary STREAMS entry points on behalf of GLDv3 compliant drivers and handles DLPI compatibility.

This section discusses the following topics:

- “GLDv3 MAC Registration” on page 382
- “GLDv3 Capabilities” on page 386
- “GLDv3 Data Paths” on page 393
- “GLDv3 State Change Notifications” on page 397
- “GLDv3 Network Statistics” on page 397
GLDv3 MAC Registration

GLDv3 defines a driver API for drivers that register with a plugin type of MAC_PLUGIN_IDENT_ETHER.

GLDv3 MAC Registration Process

A GLDv3 device driver must perform the following steps to register with the MAC layer:

- Include the following two MAC header files: sys/mac_ether.h, and sys/mac_provider.h.
  Do not include any other MAC-related header file in your driver.
- Populate the mac_callbacks structure.
- Invoke the mac_init_ops() function in its _init() entry point.
- Invoke the mac_alloc() function in its attach() entry point to allocate a mac_register structure.
- Populate the mac_register structure and invoke the mac_register() function in its attach() entry point.
- Invoke the mac_unregister() function in its detach() entry point.
- Invoke the mac_fini_ops() function in its _fini() entry point.
- Link with a dependency on misc/mac:

  # ld -N"misc/mac" xx.o -o xx

GLDv3 MAC Registration Functions

The GLDv3 interface includes driver entry points that are advertised during registration with the MAC layer and MAC entry points that are invoked by drivers.

The mac_init_ops() and mac_fini_ops() Functions

void mac_init_ops(struct dev_ops *ops, const char *name);

A GLDv3 device driver must invoke the mac_init_ops() function in its _init() entry point before calling mod_install().

void mac_fini_ops(struct dev_ops *ops);

A GLDv3 device driver must invoke the mac_fini_ops() function in its _fini() entry point after calling mod_remove().
EXAMPLE 19–1  The mac_init_ops() and mac_fini_ops() Functions

```c
int init(void) {
    int rv;
    mac_init_ops(&xx_devops, "xx");
    if ((rv = mod_install(&xx_modlinkage)) != DDI_SUCCESS) {
        mac_fini_ops(&xx_devops);
    }
    return (rv);
}

int fini(void) {
    int rv;
    if ((rv = mod_remove(&xx_modlinkage)) == DDI_SUCCESS) {
        mac_fini_ops(&xx_devops);
    }
    return (rv);
}
```

The mac_alloc() and mac_free() Functions

```c
mac_register_t *mac_alloc(uint_t version);  
```

The `mac_alloc()` function allocates a new `mac_register` structure and returns a pointer to it. Initialize the structure members before you pass the new structure to `mac_register()`. MAC-private elements are initialized by the MAC layer before `mac_alloc()` returns. The value of `version` must be `MAC_VERSION_V1`.

```c
void mac_free(mac_register_t *mregp);
```

The `mac_free()` function frees a `mac_register` structure that was previously allocated by `mac_alloc()`.

The mac_register() and mac_unregister() Functions

```c
int mac_register(mac_register_t *mregp, mac_handle_t *mhp);
```

To register a new instance with the MAC layer, a GLDv3 driver must invoke the `mac_register()` function in its `attach()` entry point. The `mregp` argument is a pointer to a `mac_register` registration information structure. On success, the `mhp` argument is a pointer to a MAC handle for the new MAC instance. This handle is needed by other routines such as `mac_tx_update()`, `mac_link_update()`, and `mac_rx()`.

EXAMPLE 19–2  The mac_alloc(), mac_register(), and mac_free() Functions and mac_register Structure

```c
int xx_attach(dev_info_t *dip, ddi_attach_cmd_t cmd) {
    mac_register_t *macp;
```
EXAMPLE 19–2  The {mac_alloc()}, {mac_register()}, and {mac_free()} Functions and {mac_register} Structure (Continued)

/* ... */

if ((macp = mac_alloc(MAC_VERSION)) == NULL) {
    xx_error(dip, "mac_alloc failed");
    goto failed;
}

macp->m_type_ident = MAC_PLUGIN_IDENT_ETHER;
macp->m_driver = xxp;
macp->m_dip = dip;
macp->m_src_addr = xxp->xx_curraddr;
macp->m_callbacks = &xx_m_callbacks;
macp->m_min_sdu = 0;
macp->m_max_sdu = ETHERMTU;
macp->m_margin = VLAN_TAGSZ;

if (mac_register(macp, &xxp->xx_mh) == DDI_SUCCESS) {
    mac_free(macp);
    return (DDI_SUCCESS);
}

/* failed to register with MAC */
mac_free(macp);
failed:
    /* ... */
}

int mac_unregister(mac_handle_t mh);

The {mac_unregister} function unregisters a MAC instance that was previously registered with {mac_register}(). The {mh} argument is the MAC handle that was allocated by {mac_register}(). Invoke {mac_unregister}() from the {detach}() entry point.

EXAMPLE 19–3  The {mac_unregister}() Function

int
xx_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
    xx_t *xxp; /* driver soft state */

    /* ... */

    switch (cmd) {
    case DDI_DETACH:
        if (mac_unregister(xxp->xx_mh) != 0) {
            return (DDI_FAILURE);
        }
    /* ... */
    }
GLDv3 MAC Registration Data Structures

The structures described in this section are defined in the `sys/mac_provider.h` header file. Include the `sys/mac_ether.h` and `sys/mac_provider.h` MAC header files in your GLDv3 driver. Do not include any other MAC-related header file.

The `mac_register(9S)` data structure is the MAC registration information structure that is allocated by `mac_alloc()` and passed to `mac_register()`. Initialize the structure members before you pass the new structure to `mac_register()`. MAC-private elements are initialized by the MAC layer before `mac_alloc()` returns. The `m_version` structure member is the MAC version. Do not modify the MAC version. The `m_type_ident` structure member is the MAC type identifier. Set the MAC type identifier to `MAC_PLUGIN_IDENT_ETHER`. The `m_callbacks` member of the `mac_register` structure is a pointer to an instance of the `mac_callbacks` structure.

The `mac_callbacks(9S)` data structure is the structure that your device driver uses to expose its entry points to the MAC layer. These entry points are used by the MAC layer to control the driver. These entry points are used to do tasks such as start and stop the adapters, manage multicast addresses, set promiscuous mode, query the capabilities of the adapter, and get and set properties. See Table 19-1 for a complete list of required and optional GLDv3 entry points. Provide a pointer to your `mac_callbacks` structure in the `m_callbacks` field of the `mac_register` structure.

The `m_callbacks` member of the `mac_callbacks` structure is a bit mask that is a combination of the following flags that specify which of the optional entry points are implemented by the driver. Other members of the `mac_callbacks` structure are pointers to each of the entry points of the driver.

- **MC_IOCTL** The `mc_ioctl()` entry point is present.
- **MC_GETCAPAB** The `mc_getcapab()` entry point is present.
- **MC_SETPROP** The `mc_setprop()` entry point is present.
- **MC_GETPROP** The `mc_getprop()` entry point is present.
- **MC_PROPINFO** The `mc_propinfo()` entry point is present.
- **MC_PROPERTIES** All properties entry points are present. Setting `MC_PROPERTIES` is equivalent to setting all three flags: `MC_SETPROP`, `MC_GETPROP`, and `MC_PROPINFO`.

**EXAMPLE 19-4** The `mac_callbacks` Structure

```c
#define XX_M_CALLBACK_FLAGS \
  (MC_IOCTL | MC_GETCAPAB | MC_PROPERTIES)

static mac_callbacks_t xx_m_callbacks = {
  XX_M_CALLBACK_FLAGS,
  xx_m_getstat, /* mc_getstat() */
  xx_m_start, /* mc_start() */
};
```
The `mac_callbacks` Structure (Continued)

```c
const struct mac_callbacks x_seq = {
  xx_m_stop,    /* mc_stop() */
  xx_m_promisc, /* mc_setpromisc() */
  xx_m_multicast, /* mc_multicast() */
  xx_municst,  /* mc_unicst() */
  xx_m_tx,     /* mc_tx() */
  NULL,        /* Reserved, do not use */
  xx_m_ioctl,  /* mc_ioctl() */
  xx_m_getcapab, /* mc_getcapab() */
  NULL,        /* Reserved, do not use */
  NULL,        /* Reserved, do not use */
  xx_m_setprop, /* mc_setprop() */
  xx_m_getprop, /* mc_getprop() */
  xx_m_propinfo /* mc_propinfo() */
};
```

**GLDv3 Capabilities**

GLDv3 implements a capability mechanism that allows the framework to query and enable capabilities that are supported by the GLDv3 driver. Use the `mc_getcapab` entry point to report capabilities. If a capability is supported by the driver, pass information about that capability, such as capability-specific entry points or flags through `mc_getcapab()`. Pass a pointer to the `mc_getcapab()` entry point in the `mac_callbacks` structure. See “GLDv3 MAC Registration Data Structures” on page 385 for more information about the `mac_callbacks` structure.

```c
boolean_t mc_getcapab(void *driver_handle, mac_capab_t cap, void *cap_data);
```

The `cap` argument specifies the type of capability being queried. The value of `cap` can be `MAC_CAPAB_HCKSUM` (hardware checksum offload), `MAC_CAPAB_LSO` (large segment offload) or `MAC_CAPAB_RINGS`. Use the `cap` argument to return the capability data to the framework.

If the driver supports the `cap` capability, the `mc_getcapab()` entry point must return `B_TRUE`. If the driver does not support the `cap` capability, `mc_getcapab()` must return `B_FALSE`.

**Example 19–5 The `mc_getcapab()` Entry Point**

```c
static boolean_t xx_m_getcapab(void *arg, mac_capab_t cap, void *cap_data)
{
  switch (cap) {
    case MAC_CAPAB_HCKSUM:
      { uint32_t *txflags = cap_data;
        *txflags = HCKSUM_INET_FULL_V4 | HCKSUM_IPHDRCKSUM;
        break;
      }
    case MAC_CAPAB_LSO:
    {    /* ... */
        break;
  }
```
The following sections describe the supported capabilities and the corresponding capability data to return.

**Rings and Ring Groups Layer–2 Classification**

Both transmit and receive hardware rings are DMA channels and can be exposed by device drivers. Rings are associated with ring groups. Receive ring groups are associated with one or more MAC addresses, and all network traffic matching any of the MAC addresses associated with a receive group must be delivered by the NIC through one of the rings of that group. The steering of traffic to the receive ring groups is enabled in hardware through layer-2 classification.

The mapping of receive rings to ring groups can be either dynamic or static. With dynamic ring groups, rings can be moved between the groups, as requested by the framework, thereby dynamically shrinking or growing the size of the groups. However with static ring groups, the rings are statically assigned to the groups and this assignment cannot change.

If a receive group contains more than one ring, the NIC must spread traffic through these rings using a hashing mechanism such as RSS (Receive Side Scaling) allowing multiple connections to be assigned different ring.

Exactly one of the receive groups must be designated as the default group (usually the first group at index 0). The following properties are associated with this receive group:

- Should have at least one ring.
- Is assigned to the primary MAC client of the NIC. The primary MAC client is assigned the primary MAC address of the NIC, and is typically IP.
- Must be used to receive all multicast and broadcast traffic received from the network.
- If the NIC is placed in promiscuous mode, it must be used to receive all traffic which does not match the MAC addresses assigned to non-default receive groups.
The following points are noteworthy with regards to the hardware implementation of receive rings and receive ring groups:

- If multiple receive rings are implemented but layer-2 classification is not supported, the hardware should expose a single receive ring group with all the ring belonging to that group to the framework.
- If layer-2 hardware classification is implemented but RSS is not supported, the hardware should register multiple receive groups, with one ring per group.
- If both layer-2 hardware classification and RSS are implemented, the hardware should register multiple receive groups with one or more rings per group.
- If neither layer-2 hardware classification nor RSS is implemented, the hardware should either not advertise a ring capability, or advertise a ring capability with a single pseudo ring and ring group, which can be used to dynamically poll the adapter for traffic.

**Registering Rings and Groups Process Overview**

Registering rings with the framework involves a process consisting of various calls from the framework to the driver. The following steps describe the registration process:

1. The framework queries the `MAC_CAPAB_RINGS` capability of the driver by calling the driver. One call is made for the transmit rings and one call for the receive rings. See "MAC_CAPAB_RINGS Capability" on page 388 for more information.

2. The framework uses the `mr_rget(9E)` and `mr_gget(9E)` entry points which are obtained from the previous step, to retrieve information about a specific ring or ring group. See the `mr_rget(9E)` and `mr_gget(9E)` man pages for more information.

3. When the framework wants to use a ring, it starts the ring group with the `mgi_start(9E)` entry point, and then starts the ring using the `mri_start(9E)` entry point as advertised in the previous step.

   Traffic can now flow through the rings until they are stopped through the `mgi_stop(9E)` and `mri_stop(9E)` entry points.

**MAC_CAPAB_RINGS Capability**

To obtain information about support for hardware transmit and receive rings, the framework sends `MAC_CAPAB_RINGS` in the `cap` argument and expects the information back in the `cap_data` field, which points to the `mac_capab_rings` structure.

The framework allocates the `mac_capab_rings(9S)` structure and sets the `mr_type` member to `MAC_RING_TYPE_RX` for receive rings, or `MAC_RING_TYPE_TX` for transmit rings. The remaining members of the structure `mac_capab_rings` is then filled by the drivers.

The following fields are defined in the `mac_capab_rings` structure:
mr_version Must be set to MAC_RINGS_VERSION_1.
mr_rnum Number of rings.
mr_gnum Number of groups.

mr_group_type The following values are defined:
  - MAC_GROUP_TYPE_DYNAMIC – The group is dynamic.
  - MAC_GROUP_TYPE_STATIC – The group is static.

See “Rings and Ring Groups Layer–2 Classification” on page 387 for more information.

mr_gget() Driver entry point to get more information about ring groups. See “mr_gget() Entry Point” on page 389 for more information.

mr_rget() Driver entry point to get more information about ring. See “mr_rget() Entry Point” on page 390 for more information.

mr_gaddring() Driver entry point to add a ring to a group. See mr_gaddring(9E).

mr_gremring() Driver entry point to remove a ring from a group. See mr_gremring(9E).

**mr_gget() Entry Point**

The mr_gget(9E) entry point is invoked by the framework for each valid group indices corresponding to the number of groups which is indicated by mr_gnum parameter. See mr_gget(9E) for more information. After the call to mr_gget(), the group information is returned in the mac_group_info structure by the driver. The structure itself is pre-allocated by the framework and is filled in by the driver.

The following fields are defined in the mac_group_info structure:

mgi_driver An opaque driver group handle which is used by the framework in future calls to group entry points.

mgi_count Number of rings in the group.

mgi_flags Group flags MAC_GROUP_DEFAULT identifies the group to be a default group. See “Rings and Ring Groups Layer–2 Classification” on page 387 for more information.

mgi_start Group start entry point.

mgi_stop Group stop entry point.

mgi_addmac Add unicast MAC address entry point.

mgi_remmac Remove unicast MAC address entry point.
mgi_addvlan  Entry point to add hardware VLAN filtering, tagging, and stripping of VLAN tags.

mgi_remvlan  Entry point to remove hardware VLAN filtering, tagging, and stripping of VLAN tags.

mgi_setmtu  Set RX group MTU entry point

mgi_getsriov_info  Entry point to retrieve SR-IOV information for the group. See “Ring Groups and SR-IOV” on page 391 for more information.

See mac_group_info(9S) and mac_group_info(9E) for detailed information.

Note – mgi_addmac(9E) and mgi_remmac(9E) entry points are used only for the receive groups. The mc_unicst(9E) entry point must be set to NULL whenever device drivers support rings capability.

Note – The mgi_addvlan() entry point performs the following actions:

- It defines VLAN IDs that must be allowed, for transmission and reception, by the NIC. That is, any tagged packet that is not in the configured list will be dropped.
- If the MAC_GROUP_VLAN_TRANSPARENT_ENABLE flag is set then it also enables the hardware VLAN tagging and stripping for that particular VLAN ID.

mr_rget() Entry Point

The mr_rget(9E) entry point is invoked by the framework for each valid group and ring indices corresponding to the number of groups which is indicated by mr_gnum and the number of rings which is indicated by mr_rnum as advertised by the call to MAC_CAPAB_RINGS. See mr_rget(9E) for detailed information.

After the call to mr_rget() is completed, the ring information is returned in the mac_ring_info structure by the driver. The structure is pre-allocated by the framework and is filled in by the driver.

The following fields are defined in the mac_ring_info structure:

mri_driver  An opaque driver group handle which is used by the framework in future calls to ring entry points.

mri_start  Ring start entry point.

mri_stop  Ring stop entry point.
mri_stat

Ring statistics entry point. See “GLDv3 Network Statistics” on page 397 for more information.

mri_tx

Ring transmit entry point. See “Transmit Data Path” on page 394 for more information.

mri_poll

Ring poll entry point. “Receive Data Path” on page 395 for more information.

mri_intr_ddi_handle

The DDI interrupt handle associate with the interrupt for this ring.

mri_intr_enable(9E)

Enable interrupts on RX rings. See “Receive Data Path” on page 395 for more information.

mri_intr_disable(9E)

Disable interrupts on RX rings. “Receive Data Path” on page 395 for more information.

See mac_group_info(9S) and mac_ring_info(9S) man pages for detailed information.

Note – mri_tx() must be set for transmit rings only and mri_poll() must be set only for receive rings.

Note – If a driver implements rings capability, then the mc_tx() entry point in the mac_callbacks structure must be set to NULL.

Ring Groups and SR-IOV

The device drivers that are SR-IOV capable use the MAC_CAPAB_RINGS capability to inform the framework that they are SR-IOV capable by implementing the mgi_getsriov_info(9E) group entry point. The PF driver is responsible for implementing this entry point.

After the call to mgi_getsriov_info(9E), the SR-IOV information is returned in the mac_sriov_info structure by the driver. The structure is pre-allocated by the framework and is filled-in by the driver.

The PF (Physical Function) driver instance registers as many transmit and receive ring groups as the number of VFs (Virtual Functions). These ring groups advertised by the PF driver are special and are used to manage the VFs. The ring groups do not have any data flowing through them. They are used to configure unicast MAC address, set MTU, add VLAN filters, remove VLAN filters, remove VLAN hardware, and perform VLAN tagging and stripping for VFs.

Note – The VF driver programs the MAC multicast group that the driver wants to join. The PF driver does not control the programming of these addresses.
The msi_vf_index structure member, set by the PF driver, captures the VF index that corresponds to a ring group. This is the same index used by the device driver when the driver calls the pci_plist_getvf(9F) function.


**Hardware Checksum Offload**

To get data about support for hardware checksum offload, the framework sends MAC_CAPAB_HCKSUM in the cap argument. See “Hardware Checksum Offload Capability Information” on page 392.

To query checksum offload metadata and retrieve the per-packet hardware checksumming metadata when hardware checksumming is enabled, use mac_hcksum_get(9F). See “The mac_hcksum_get() Function Flags” on page 392.

To set checksum offload metadata, use mac_hcksum_set(9F). See “The mac_hcksum_set() Function Flags” on page 393.

See “Hardware Checksumming: Hardware” on page 395 and “Hardware Checksumming: MAC Layer” on page 396 for more information.

**Hardware Checksum Offload Capability Information**

To pass information about the MAC_CAPAB_HCKSUM capability to the framework, the driver must set a combination of the following flags in cap_data, which points to a uint32_t. These flags indicate the level of hardware checksum offload that the driver is capable of performing for outbound packets.

- HCKSUM_INET_PARTIAL: Partial 1’s complement checksum ability
- HCKSUM_INET_FULL_V4: Full 1’s complement checksum ability for IPv4 packets
- HCKSUM_INET_FULL_V6: Full 1’s complement checksum ability for IPv6 packets
- HCKSUM_IPHDRCKSUM: IPv4 Header checksum offload capability

**The mac_hcksum_get() Function Flags**

The flags argument of mac_hcksum_get() is a combination of the following values:

- HCK_FULLCKSUM: Compute the full checksum for this packet.
- HCK_FULLCKSUM_OK: The full checksum was verified in hardware and is correct.
- HCK_PARTIALCKSUM: Compute the partial 1’s complement checksum based on other parameters passed to mac_hcksum_get(). HCK_PARTIALCKSUM is mutually exclusive with HCK_FULLCKSUM.
HCK_IPV4_HDRCKSUM
Compute the IP header checksum.

HCK_IPV4_HDRCKSUM_OK
The IP header checksum was verified in hardware and is correct.

The `mac_hcksum_set()` Function Flags

The `flags` argument of `mac_hcksum_set()` is a combination of the following values:

- **HCK_FULLCKSUM**: The full checksum was computed and passed through the `value` argument.
- **HCK_FULLCKSUM_OK**: The full checksum was verified in hardware and is correct.
- **HCK_PARTIALCKSUM**: The partial checksum was computed and passed through the `value` argument. **HCK_PARTIALCKSUM** is mutually exclusive with **HCK_FULLCKSUM**.
- **HCK_IPV4_HDRCKSUM**: The IP header checksum was computed and passed through the `value` argument.
- **HCK_IPV4_HDRCKSUM_OK**: The IP header checksum was verified in hardware and is correct.

Large Segment (or Send) Offload

To query support for large segment (or send) offload, the framework sends `MAC_CAPAB_LSO` in the `cap` argument and expects the information back in `cap_data`, which points to a `mac_capab_lso(9S)` structure. The framework allocates the `mac_capab_lso` structure and passes a pointer to this structure in `cap_data`. The `mac_capab_lso` structure consists of an `lso_basic_tcp_ipv4(9S)` structure and an `lso_flags` member. If the driver instance supports LSO for TCP on IPv4, set the `LSO_TX_BASIC_TCP_IPV4` flag in `lso_flags` and set the `lso_max` member of the `lso_basic_tcp_ipv4` structure to the maximum payload size supported by the driver instance.

Use `mac_lso_get(9F)` to obtain per-packet LSO metadata. If LSO is enabled for this packet, the `HW_LSO` flag is set in the `mac_lso_get()` `flags` argument. The maximum segment size (MSS) to be used during segmentation of the large segment is returned through the location pointed to by the `mss` argument. See “Large Segment Offload” on page 395 for more information.

GLDv3 Data Paths

Data-path entry points are comprised of the following components:

- Callbacks exported by the driver and invoked by the GLDv3 framework for sending packets.
- GLDv3 framework entry points called by the driver for transmit flow control and for receiving packets.
Note – If a driver implements the rings capability then all data sent and received by the driver is passed through ring-specific entry points.

Transmit Data Path

The type of transmit entry point invoked by the GLDv3 framework to pass a message block to the driver is dependent on the underlying driver support for MAC_CAPAB_RINGS. If the driver supports MAC_CAPAB_RINGS capability then the framework invokes mri_tx(9E) ring entry point. Otherwise the framework invokes mc_tx(9E) entry point.

Accordingly, the device driver has to provide a pointer to the transmit entry point in either mc_tx() or mri_tx(). See “GLDv3 MAC Registration Data Structures” on page 385 and “mr_rget() Entry Point” on page 390 for more information.

EXAMPLE 19–6  The mc_tx() and mri_tx() Entry Point

```c
mblk_t *
xx_m_tx(void *arg, mblk_t *mp)
{
    xx_t *xxp = arg;
    mblk_t *nmp;
    mutex_enter(&xxp->xx_xmtlock);
    if (xxp->xx_flags & XX_SUSPENDED) {
        while ((nmp = mp) != NULL) {
            xxp->xx_carrier_errors++;
            mp = mp->b_next;
            freemsg(nmp);
        }
        mutex_exit(&xxp->xx_xmtlock);
        return (NULL);
    }
    while (mp != NULL) {
        nmp = mp->b_next;
        mp->b_next = NULL;
        if (!xx_send(xxp, mp)) {
            mp->b_next = nmp;
            break;
        }
        mp = nmp;
    }
    mutex_exit(&xxp->xx_xmtlock);
    return (mp);
}
```

The following sections discuss topics related to transmitting data to the hardware.
Flow Control

If the driver cannot send the packets because of insufficient hardware resources, the driver returns the sub-chain of packets that could not be sent. When more descriptors become available at a later time, the driver must invoke `mac_tx_update(9F)` or `mac_tx_ring(9F)` to notify the framework. The driver will invoke either function depending on whether the driver implements Rings Capability.

Hardware Checksumming: Hardware

If the driver specified hardware checksum support (see “Hardware Checksum Offload” on page 392), then the driver must do the following tasks:

- Use `mac_hcksum_get(9F)` to check every packet for hardware checksum metadata.
- Program the hardware to perform the required checksum calculation.

Large Segment Offload

If the driver specified LSO capabilities (see “Large Segment (or Send) Offload” on page 393), then the driver must use `mac_lso_get(9F)` to query whether LSO must be performed on the packet.

Virtual LAN: Hardware

When the administrator configures VLANs, the MAC layer inserts the needed VLAN headers on the outbound packets before they are passed to the driver through the `mc_tx()` entry point. However, if the hardware supports VLAN tagging then the tagging is offloaded to the hardware. See “mr_gget() Entry Point” on page 389 for more details.

Receive Data Path

The receive data-path can be interrupt-driven or poll-driven.

Receive Interrupt Data Path

Note: If the driver does not support the rings capability then call the `mac_rx(9F)` function in your driver’s interrupt handler to pass a chain of one or more packets up the stack to the MAC layer. Avoid holding mutex or other locks during the call to `mac_rx()` or `mac_rx_ring()`. In particular, do not hold locks that could be taken by a transmit thread during a call to `mac_rx()` or `mac_rx_ring()`.

In interrupt mode, packet chains are sent up from the driver to the framework whenever they are received by the NIC and available by the driver for pickup. Packet chains consists of one or
more mblk_t chained with each other through b_next and allow per-packet processing overhead to be reduced. Received packets are passed up to the framework in interrupt mode by calling the mac_rx_ring() entry point.

```c
void mac_rx_ring(mac_handle_t mh, mac_ring_handle_t mrh, mblk_t *mp_chain, int64_t mr_gen_num)
```

The `mh_handle` corresponds to the MAC handle obtained by the device driver when it registered with the kernel via the `mac_register()` function. The `mrh_handle` is the framework ring handle which was passed to the driver as part of the `mr_rget()` call. `mr_gen_num` must be set to the generation number specified by the framework when the receive ring was started via the `mri_start()` entry point. The ring generation number provided by the driver is matched with the ring generation number held in framework. If they do not match, received packets are considered stale packets coming from an older assignment of the ring and they will be dropped.

**Receive Polling Data-Path**

In addition to being able to receive packets through an interrupt-driven path, framework also supports a polling-based data path. In polling mode, a kernel thread running in the stack fetches packets from the driver through a polling entry point. This allows the stack to efficiently control when packets will be processed, with which priority, while reducing the numbers of interrupts coming into the system based on actual load. In addition, polling allows the stack to more effectively enforce bandwidth limits on received traffic, which is especially critical in virtualization scenarios. The host toggles between interrupt and polling mode on demand.

While a ring is in polling mode, the driver should not deliver packets received through the receive ring using `mac_rx_ring()` function. This is guaranteed as interrupts are disabled while in polling mode. Instead, the framework will call the `mri_poll()` entry point that was exposed by the driver as part of the `mac_ring_info` structure. See “`mr_rget()` Entry Point” on page 390 for more information.

**Switching Between Interrupt and Polling Mode**

By default, a ring should be in interrupt mode after it is started. As long as a ring is in interrupt mode, it should pass up received packets in the form of chains through the entry points. When the host switches a ring to polling mode, it disables its interrupt by invoking the entry point through the `mac_intr` structure, which was previously exposed through the `mac_ring_info` structure.

**Hardware Checksumming: MAC Layer**

If the driver specified hardware checksum support (see “Hardware Checksum Offload” on page 392), then the driver must use the `mac_hcksum_set(9F)` function to associate hardware checksumming metadata with the packet.
Virtual LAN: MAC Layer

VLAN packets must be passed with their tags to the MAC layer. Do not strip the VLAN headers from the packets. However if the hardware supports VLAN stripping and the framework has requested the hardware to strip VLAN tags then the hardware can strip VLAN tags to improve performance. See “mr_gget() Entry Point” on page 389 for more information.

GLDv3 State Change Notifications

A driver can call the following functions to notify the network stack that the driver’s state has changed.

```c
void mac_tx_update(mac_handle_t mh);
void mac_tx_ring_update(mac_handle_t mh, mac_ring_handle_t rh);
```

The `mac_tx_update()` or `mac_tx_ring()` functions notify the framework that more TX descriptors are available. If `mc_tx()` or `mri_tx()` return a non-empty chain of packets, then the driver must call `mac_tx_update()` or `mac_tx_ring_update()` as soon as possible after resources are available to inform the MAC layer to retry the packets that were returned as not sent. See “Transmit Data Path” on page 394 for more information about the `mc_tx()` and `mri_tx()` entry points.

```c
void mac_link_update(mac_handle_t mh, link_state_t new_state);
```

The `mac_link_update()` function notifies the MAC layer that the state of the media link has changed. The `new_state` argument must be one of the following values:

- `LINK_STATE_UP` The media link is up.
- `LINK_STATE_DOWN` The media link is down.
- `LINK_STATE_UNKNOWN` The media link is unknown.

GLDv3 Network Statistics

Device drivers maintain a set of statistics for the device instances they manage. The MAC layer queries these statistics through the `mc_getstat()` entry point of the driver.

```c
int mc_getstat(void *driver_handle, uint_t stat, uint64_t *stat_value);
```

The GLDv3 framework uses `stat` to specify the statistic being queried. The driver uses `stat_value` to return the value of the statistic specified by `stat`. If the value of the statistic is returned, `mc_getstat()` must return 0. If the `stat` statistic is not supported by the driver, `mc_getstat()` must return `ENOTSUP`. 
The GLDv3 statistics that are supported are the union of generic MAC statistics and Ethernet-specific statistics. See the `mc_getstat(9E)` man page for a complete list of supported statistics.

**EXAMPLE 19–7  The mc_getstat() Entry Point**

```c
int xx_m_getstat(void *arg, uint_t stat, uint64_t *val)
{
    xx_t *xxp = arg;

    mutex_enter(&xxp->xx_xmtlock);
    if (((xxp->xx_flags & (XX_RUNNING|XX_SUSPENDED)) == XX_RUNNING)
        xx_reclaim(xxp);
    mutex_exit(&xxp->xx_xmtlock);

    switch (stat) {
    case MAC_STAT_MULTIRCV:
        *val = xxp->xx_multircv;
        break;
    /* ... */
    case ETHER_STAT_MACRCV_ERRORS:
        *val = xxp->xx_macrcv_errors;
        break;
    /* ... */
    default:
        return (ENOTSUP);
    }
    return (0);
}
```

The `mri_stat()` ring entry point is a mandatory ring entry point that must be implemented by all the device drivers that support rings capability. This entry point will be used by the framework to query the statistics maintained for each of the hardware transmit and receive rings.

For the hardware transmit rings, the framework queries the following statistics:

- MAC_STAT_ERRORS
- MAC_STAT_OBYTES
- MAC_STAT_OPACKETS

For the hardware receive rings, the framework queries the following statistics:

- MAC_STAT_IERRORS
- MAC_STAT_RBYTES
- MAC_STAT_IPACKETS
GLDv3 Properties

Use the `mc_propinfo(9E)` entry point to return immutable attributes of a property. This information includes permissions, default values, and allowed value ranges. Use `mc_setprop(9E)` to set the value of a property for this particular driver instance. Use `mc_getprop(9E)` to return the current value of a property.

See the `mc_propinfo(9E)` man page for a complete list of properties and their types.

The `mc_propinfo()` entry point should invoke the `mac_prop_info_set_perm()`, `mac_prop_info_set_default()`, and `mac_prop_info_set_range()` functions to associate specific attributes of the property being queried, such as default values, permissions, or allowed value ranges.

The `mac_prop_info_set_default_uint8()`, `mac_prop_info_set_default_str()`, and `mac_prop_info_set_default_link_flowctrl()` functions associate a default value with a specific property. The `mac_prop_info_set_range_uint32()` function associates an allowed range of values for a specific property.

The `mac_prop_info_set_perm()` function specifies the permission of the property. The permission can be one of the following values:

- `MAC_PROP_PERM_READ` The property is read-only
- `MAC_PROP_PERM_WRITE` The property is write-only
- `MAC_PROP_PERM_RW` The property can be read and written

If the `mc_propinfo()` entry point does not call `mac_prop_info_set_perm()` for a particular property, the GLDv3 framework assumes that the property has read and write permissions, corresponding to `MAC_PROP_PERM_RW`.

In addition to the properties listed in the `mc_propinfo(9E)` man page, drivers can also expose driver-private properties. Use the `m_priv_props` field of the `mac_register` structure to specify driver-private properties supported by the driver. The framework passes the `MAC_PROP_PRIVATE` property ID in `mc_setprop()`, `mc_getprop()`, or `mc_propinfo()`. See the `mc_propinfo(9E)` man page for more information.

Summary of GLDv3 Interfaces

The following table lists entry points, other DDI functions, and data structures that are part of the GLDv3 network device driver framework.
### TABLE 19-1  GLDv3 Interfaces

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Required Entry Points</strong></td>
<td></td>
</tr>
<tr>
<td><code>mc_getstat(9E)</code></td>
<td>Retrieve network statistics from the driver. See “GLDv3 Network Statistics” on page 397.</td>
</tr>
<tr>
<td><code>mc_start(9E)</code></td>
<td>Start a driver instance. The GLDv3 framework invokes the start entry point before any operation is attempted.</td>
</tr>
<tr>
<td><code>mc_stop(9E)</code></td>
<td>Stop a driver instance. The MAC layer invokes the stop entry point before the device is detached.</td>
</tr>
<tr>
<td><code>mc_setpromisc(9E)</code></td>
<td>Change the promiscuous mode of the device driver instance.</td>
</tr>
<tr>
<td><code>mc_multicst(9E)</code></td>
<td>Add or remove a multicast address.</td>
</tr>
<tr>
<td><code>mc_unicst(9E)</code></td>
<td>Set the primary unicast address. The device must start passing back through <code>mac_rx()</code> the packets with a destination MAC address that matches the new unicast address. See “Receive Data Path” on page 395 for information about <code>mac_rx()</code>.</td>
</tr>
<tr>
<td><code>mc_tx(9E)</code></td>
<td>Send one or more packets. See “Transmit Data Path” on page 394.</td>
</tr>
<tr>
<td><code>mr_rget(9E)</code></td>
<td>Obtain transmit and receive ring information. See “mr_rget() Entry Point” on page 390 for more information.</td>
</tr>
<tr>
<td><code>mr_gget(9E)</code></td>
<td>Obtain transmit and receive ring information. See “mr_gget() Entry Point” on page 389 for more information.</td>
</tr>
<tr>
<td><code>mr_gaddring(9E)</code></td>
<td>Add ring to a receive group. This is required only if dynamic ring grouping is supported. See “MAC_CAPAB_RINGS Capability” on page 388.</td>
</tr>
<tr>
<td><code>mr_gremring(9E)</code></td>
<td>Remove ring from a receive group. This is required only if dynamic ring grouping is supported. See “MAC_CAPAB_RINGS Capability” on page 388.</td>
</tr>
<tr>
<td><code>mri_tx(9E)</code></td>
<td>Transmit packets for TX rings. See “mr_rget() Entry Point” on page 390 for more information.</td>
</tr>
<tr>
<td><code>mri_poll()</code></td>
<td>Poll RX ring for packets. See “mr_rget() Entry Point” on page 390 for more information.</td>
</tr>
<tr>
<td><code>mri_stat()</code></td>
<td>Ring statistics. See “mr_rget() Entry Point” on page 390 for more information.</td>
</tr>
</tbody>
</table>
### TABLE 19–1  GLDv3 Interfaces  
(Continued)

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mri_intr_enable(9E)</td>
<td>Enable interrupts on RX ring. See &quot;mr_rget() Entry Point&quot; on page 390 for more information.</td>
</tr>
<tr>
<td>mri_intr_enable(9E)</td>
<td>Disable interrupts on RX ring. See &quot;mr_rget() Entry Point&quot; on page 390 for more information.</td>
</tr>
<tr>
<td>mgi_addmac(9E)</td>
<td>Program a MAC address into the driver’s hardware for RX ring group. See &quot;mr_gget() Entry Point&quot; on page 389 for more information.</td>
</tr>
<tr>
<td>mgi_remmac(9E)</td>
<td>Remove a previously programmed MAC address from the driver’s hardware for RX ring group. See &quot;mr_gget() Entry Point&quot; on page 389 for more information.</td>
</tr>
</tbody>
</table>

#### Optional Entry Points

- **mc_ioctl(9E)**  
  Optional ioctl driver interface. This facility is intended to be used only for debugging purposes.

- **mc_getcapab(9E)**  
  Retrieve capabilities. See "GLDv3 Capabilities" on page 386.

- **mc_setprop(9E)**  
  Set a property value. See "GLDv3 Properties" on page 399.

- **mc_getprop(9E)**  
  Get a property value. See "GLDv3 Properties" on page 399.

- **mc_propinfo(9E)**  
  Get information about a property. See "GLDv3 Properties" on page 399.

- **mri_start()**  
  Start ring. See "mr_rget() Entry Point" on page 390 for more information.

- **mri_stop()**  
  Stop ring. See "mr_rget() Entry Point" on page 390 for more information.

- **mgi_start(9E)**  
  Ring Group start. See "mr_gget() Entry Point" on page 389 for more information.

- **mgi_stop(9E)**  
  Ring Group stop. See "mr_gget() Entry Point" on page 389 for more information.

- **mgi_addvlan()**  
  Enable VLAN filtering in hardware. See "mr_gget() Entry Point" on page 389 for more information.

- **mgi_remvlan()**  
  Remove previously programmed VLAN filter. See "mr_gget() Entry Point" on page 389 for more information.
### TABLE 19–1  GLDv3 Interfaces  (Continued)

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mgi_setmtu()</td>
<td>Set RX group MTU. See &quot;mgi_get() Entry Point&quot; on page 389 for more information.</td>
</tr>
<tr>
<td>mgi_get_sriov_info()</td>
<td>Obtain SR-IOV information. See &quot;Ring Groups and SR-IOV&quot; on page 391 for more information.</td>
</tr>
</tbody>
</table>

**Data Structures**

- **mac_register**<sup>(9S)</sup>: Registration information. See "GLDv3 MAC Registration Data Structures" on page 385.
- **mac_callbacks**<sup>(9S)</sup>: Driver callbacks. See "GLDv3 MAC Registration Data Structures" on page 385.
- **mac_capab_lso**<sup>(9S)</sup>: LSO metadata. See "Large Segment (or Send) Offload" on page 393.
- **lso_basic_tcp_ipv4**<sup>(9S)</sup>: LSO metadata for TCP/IPv4. See "Large Segment (or Send) Offload" on page 393.
- **mac_capab_rings**<sup>(9S)</sup>: See "MAC Rings Capability" on page 387 for more information.
- **mac_group_info**<sup>(9S)</sup>: See "mgi_get() Entry Point" on page 389 for more information.
- **mac_ring_info**<sup>(9S)</sup>: See "mgi_get() Entry Point" on page 390 for more information.
- **mac_intr_t**
- **mac_sriov_info**

**MAC Registration Functions**

- **mac_alloc**<sup>(9F)</sup>: Allocate a new mac_register structure. See "GLDv3 MAC Registration" on page 382.
- **mac_free**<sup>(9F)</sup>: Free a mac_register structure.
- **mac_register**<sup>(9F)</sup>: Register with the MAC layer.
- **mac_unregister**<sup>(9F)</sup>: Unregister from the MAC layer.
- **mac_init_ops**<sup>(9F)</sup>: Initialize the driver’s dev_ops<sup>(9S)</sup> structure.
- **mac_fini_ops**<sup>(9F)</sup>: Release the driver’s dev_ops structure.

**Data Transfer Functions**

- **mac_rx**<sup>(9F)</sup>: Pass up received packets. See "Receive Data Path" on page 395.
GLDv3 Interfaces (Continued)

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mac_rx_ring(9F)</td>
<td>Pass up received packets. See “Receive Data Path” on page 395.</td>
</tr>
<tr>
<td>mac_tx_update(9F)</td>
<td>TX resources are available. See “GLDv3 State Change Notifications” on page 397.</td>
</tr>
<tr>
<td>mac_tx_ring_update(9F)</td>
<td>TX resources are available. See “GLDv3 State Change Notifications” on page 397 for more information.</td>
</tr>
<tr>
<td>mac_link_update(9F)</td>
<td>Link state has changed.</td>
</tr>
<tr>
<td>mac_hcksum_get(9F)</td>
<td>Retrieve hardware checksum information. See “Hardware Checksum Offload” on page 392 and “Transmit Data Path” on page 394.</td>
</tr>
<tr>
<td>mac_hcksum_set(9F)</td>
<td>Attach hardware checksum information. See “Hardware Checksum Offload” on page 392 and “Receive Data Path” on page 395.</td>
</tr>
<tr>
<td>mac_lso_get(9F)</td>
<td>Retrieve LSO information. See “Large Segment (or Send) Offload” on page 393.</td>
</tr>
</tbody>
</table>

Properties Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mac_prop_info_set_perm(9F)</td>
<td>Set the permission of a property. See “GLDv3 Properties” on page 399.</td>
</tr>
<tr>
<td>mac_prop_info_set_default_uint8(9F), mac_prop_info_set_default_str(9F), mac_prop_info_set_default_link_flowctrl(9F)</td>
<td>Set a property value.</td>
</tr>
<tr>
<td>mac_prop_info_set_range_uint32(9F)</td>
<td>Set a property values range.</td>
</tr>
</tbody>
</table>

GLDv2 Network Device Driver Framework

GLDv2 is a multi-threaded, clonable, loadable kernel module that provides support to device drivers for local area networks. Local area network (LAN) device drivers in the Oracle Solaris OS are STREAMS-based drivers that use the Data Link Provider Interface (DLPI) to communicate with network protocol stacks. These protocol stacks use the network drivers to send and receive packets on a LAN. The GLDv2 implements much of the STREAMS and DLPI functionality for an Oracle Solaris LAN driver. The GLDv2 provides common code that many network drivers can share. Using the GLDv2 reduces duplicate code and simplifies your network driver.

For more information about GLDv2, see the gld(7D) man page.

STREAMS drivers are documented in Part II, “Kernel Interface,” in STREAMS Programming Guide. Specifically, see Chapter 9, “STREAMS Drivers,” in the STREAMS guide. The STREAMS
framework is a message-based framework. Interfaces that are unique to STREAMS drivers include STREAMS message queue processing entry points.

The DLPI specifies an interface to the Data Link Services (DLS) of the Data Link Layer of the OSI Reference Model. The DLPI enables a DLS user to access and use any of a variety of conforming DLS providers without special knowledge of the provider’s protocol. The DLPI specifies access to the DLS provider in the form of M_PROTO and M_PCPROTO type STREAMS messages. A DLPI module uses STREAMS ioctl calls to link to the MAC sub-layer. For more information about the DLPI protocol, including Oracle Solaris-specific DLPI extensions, see the dlpi(7P) man page. For general information about DLPI, see the DLPI standard at http://www.opengroup.org/pubs/catalog/c811.htm.

An Oracle Solaris network driver that is implemented using GLDv2 has two distinct parts:

- Generic component. Handles STREAMS and DLPI interfaces.
- Device-specific component. Handles the particular hardware device.
  - Indicates the driver’s dependence on the GLDv2 module by linking with a dependency on misc/gld. The GLDv2 module can be found at /kernel/misc/sparcv9/gld on SPARC systems, /kernel/misc/amd64/gld on 64-bit x86 systems, and /kernel/misc/gld on 32-bit x86 systems.
  - Registers with GLDv2: In its attach(9E) entry point, the driver provides GLDv2 with pointers to its other entry points. GLDv2 uses these pointers to call into the gld(9E) entry point.
  - Calls gld(9F) functions to process data or to use some other GLDv2 service. The gld_mac_info(9S) structure is the primary data interface between GLDv2 and the device-specific driver.

GLDv2 drivers must process fully formed MAC-layer packets and must not perform logical link control (LLC) handling.

This section discusses the following topics:

- “GLDv2 Device Support” on page 405
- “GLDv2 DLPI Providers” on page 406
- “GLDv2 DLPI Primitives” on page 407
- “GLDv2 I/O Control Functions” on page 408
- “GLDv2 Driver Requirements” on page 409
- “GLDv2 Network Statistics” on page 410
- “GLDv2 Declarations and Data Structures” on page 414
- “GLDv2 Function Arguments” on page 418
- “GLDv2 Entry Points” on page 419
- “GLDv2 Return Values” on page 423
- “GLDv2 Service Routines” on page 423
The GLDv2 framework supports the following types of devices:

- **DLEther**: ISO 8802-3, IEEE 802.3 protocol
- **DL_TPR**: IEEE 802.5, Token Passing Ring
- **DL_FDDI**: ISO 9314-2, Fibre Distributed Data Interface

### Ethernet V2 and ISO 8802-3 (IEEE 802.3)

For devices that are declared to be type **DL_Ether**, GLDv2 provides support for both Ethernet V2 and ISO 8802-3 (IEEE 802.3) packet processing. Ethernet V2 enables a user to access a conforming provider of data link services without special knowledge of the provider’s protocol. A service access point (SAP) is the point through which the user communicates with the service provider.

Streams bound to SAP values in the range [0-255] are treated as equivalent and denote that the user wants to use 8802-3 mode. If the SAP value of the **DL_BIND_REQ** is within this range, GLDv2 computes the length of each subsequent **DL_UNITDATA_REQ** message on that stream. The length does not include the 14-byte media access control (MAC) header. GLDv2 then transmits 8802-3 frames that have those lengths in the MAC frame header type fields. Such lengths do not exceed 1500.

Frames that have a type field in the range [0-1500] are assumed to be 8802-3 frames. These frames are routed up all open streams in 8802-3 mode. Those streams with SAP values in the [0-255] range are considered to be in 8802-3 mode. If more than one stream is in 8802-3 mode, the incoming frame is duplicated and routed up these streams.

Those streams that are bound to SAP values that are greater than 1500 are assumed to be in Ethernet V2 mode. These streams receive incoming packets whose Ethernet MAC header type value exactly matches the value of the SAP to which the stream is bound.

### TPR and FDDI: SNAP Processing

For media types **DL_TPR** and **DL_FDDI**, GLDv2 implements minimal SNAP (Sub-Net Access Protocol) processing. This processing is for any stream that is bound to a SAP value that is greater than 255. SAP values in the range [0-255] are LLC SAP values. Such values are carried naturally by the media packet format. SAP values that are greater than 255 require a SNAP header, subordinate to the LLC header, to carry the 16-bit Ethernet V2-style SAP value.

SNAP headers are carried under LLC headers with destination SAP 0xAA. Outbound packets with SAP values that are greater than 255 require an LLC+SNAP header take the following form:

```
AA AA 03 00 00 00 XX XX
```
XX XX represents the 16-bit SAP, corresponding to the Ethernet V2 style type. This header is unique in supporting non-zero organizational unique identifier fields. LLC control fields other than 03 are considered to be LLC packets with SAP 0xAA. Clients that want to use SNAP formats other than this format must use LLC and bind to SAP 0xAA.

Incoming packets are checked for conformance with the above format. Packets that conform are matched to any streams that have been bound to the packet’s 16-bit SNAP type. In addition, these packets are considered to match the LLC SNAP SAP 0xAA.

Packets received for any LLC SAP are passed up all streams that are bound to an LLC SAP, as described for media type DL_ETHER.

**TPR: Source Routing**

For type DL_TPR devices, GLDv2 implements minimal support for source routing.

Source routing support includes the following tasks:

- Specify routing information for a packet to be sent across a bridged medium. The routing information is stored in the MAC header. This information is used to determine the route.
- Learn routes.
- Solicit and respond to requests for information about possible multiple routes.
- Select among available routes.

Source routing adds routing information fields to the MAC headers of outgoing packets. In addition, this support recognizes such fields in incoming packets.

GLDv2 source routing support does not implement the full route determination entity (RDE) specified in Section 9 of ISO 8802-2 (IEEE 802.2). However, this support can interoperate with any RDE implementations that might exist in the same or a bridged network.

**GLDv2 DLPI Providers**

GLDv2 implements both Style 1 and Style 2 DLPI providers. A physical point of attachment (PPA) is the point at which a system attaches itself to a physical communication medium. All communication on that physical medium funnels through the PPA. The Style 1 provider attaches the streams to a particular PPA based on the major or minor device that has been opened. The Style 2 provider requires the DLS user to explicitly identify the desired PPA using DL_ATTACH_REQ. In this case, open(9E) creates a stream between the user and GLDv2, and DL_ATTACH_REQ subsequently associates a particular PPA with that stream. Style 2 is denoted by a minor number of zero. If a device node whose minor number is not zero is opened, Style 1 is indicated and the associated PPA is the minor number minus 1. In both Style 1 and Style 2 opens, the device is cloned.
GLDv2 implements several DLPI primitives. The DL_INFO_REQ primitive requests information about the DLPI streams. The message consists of one M_PROTO message block. GLDv2 returns device-dependent values in the DL_INFO_ACK response to this request. These values are based on information that the GLDv2-based driver specified in the gld_mac_info(9S) structure that was passed to the gld_register(9F) function.

GLDv2 returns the following values on behalf of all GLDv2-based drivers:

- Version is DL_VERSION_2.
- Service mode is DL_CLDLS. GLDv2 implements connectionless-mode service.
- Provider style is DL_STYLE1 or DL_STYLE2, depending on how the stream was opened.
- No optional Quality of Service (QOS) support is present. The QOS fields are zero.

Note – Contrary to the DLPI specification, GLDv2 returns the correct address length and broadcast address of the device in DL_INFO_ACK even before the stream has been attached to a PPA.

The DL_ATTACH_REQ primitive is used to associate a PPA with a stream. This request is needed for Style 2 DLS providers to identify the physical medium over which the communication is sent. Upon completion, the state changes from DL_UNATTACHED to DL_UNBOUND. The message consists of one M_PROTO message block. This request is not allowed when Style 1 mode is used. Streams that are opened using Style 1 are already attached to a PPA by the time the open completes.

The DL_DETACH_REQ primitive requests to detach the PPA from the stream. This detachment is allowed only if the stream was opened using Style 2.

The DL_BIND_REQ and DL_UNBIND_REQ primitives bind and unbind a DLSAP (data link service access point) to the stream. The PPA that is associated with a stream completes initialization before the completion of the processing of the DL_BIND_REQ on that stream. You can bind multiple streams to the same SAP. Each stream in this case receives a copy of any packets that were received for that SAP.

The DL_ENABMULTI_REQ and DL_DISABMULTI_REQ primitives enable and disable reception of individual multicast group addresses. Through iterative use of these primitives, an application or other DLS user can create or modify a set of multicast addresses. The streams must be attached to a PPA for these primitives to be accepted.

The DL_PROMISCON_REQ and DL_PROMISCOFF_REQ primitives turn promiscuous mode on or off on a per-stream basis. These controls operate at either at a physical level or at the SAP level. The DL Provider routes all received messages on the media to the DLS user. Routing continues until
a DL_DETACH_REQ is received, a DL_PROMISCOFF_REQ is received, or the stream is closed. You can specify physical level promiscuous reception of all packets on the medium or of multicast packets only.

**Note** – The streams must be attached to a PPA for these promiscuous mode primitives to be accepted.

The DL_UNITDATA_REQ primitive is used to send data in a connectionless transfer. Because this service is not acknowledged, delivery is not guaranteed. The message consists of one M_PROTO message block followed by one or more M_DATA blocks containing at least one byte of data.

The DL_UNITDATA_IND type is used when a packet is to be passed on upstream. The packet is put into an M_PROTO message with the primitive set to DL_UNITDATA_IND.

The DL_PHYS_ADDR_REQ primitive requests the MAC address currently associated with the PPA attached to the streams. The address is returned by the DL_PHYS_ADDR_ACK primitive. When using Style 2, this primitive is only valid following a successful DL_ATTACH_REQ.

The DL_SET_PHYS_ADDR_REQ primitive changes the MAC address currently associated with the PPA attached to the streams. This primitive affects all other current and future streams attached to this device. Once changed, all streams currently or subsequently opened and attached to this device obtain this new physical address. The new physical address remains in effect until this primitive changes the physical address again or the driver is reloaded.

**Note** – The superuser is allowed to change the physical address of a PPA while other streams are bound to the same PPA.

The DL_GET_STATISTICS_REQ primitive requests a DL_GET_STATISTICS_ACK response containing statistics information associated with the PPA attached to the stream. Style 2 Streams must be attached to a particular PPA using DL_ATTACH_REQ before this primitive can succeed.

**GLDv2 I/O Control Functions**

GLDv2 implements the ioctl ioc_cmd function described below. If GLDv2 receives an unrecognizable ioctl command, GLDv2 passes the command to the device-specific driver’s gldm_ioctl() routine, as described in gld(9E).

The DLIOCRAW ioctl function is used by some DLPI applications, most notably the snoop(1M) command. The DLIOCRAW command puts the stream into a raw mode. In raw mode, the driver passes full MAC-level incoming packets upstream in M_DATA messages instead of transforming the packets into the DL_UNITDATA_IND form. The DL_UNITDATA_IND form is normmally used for
reporting incoming packets. Packet SAP filtering is still performed on streams that are in raw mode. If a stream user wants to receive all incoming packets, the user must also select the appropriate promiscuous modes. After successfully selecting raw mode, the application is also allowed to send fully formatted packets to the driver as _M_DATA messages for transmission. DLIOCRAW takes no arguments. Once enabled, the stream remains in this mode until closed.

**GLDv2 Driver Requirements**

GLDv2-based drivers must include the header file <sys/gld.h>.

GLDv2-based drivers must be linked with the `-N"misc/gld"` option:

```
%ld -r -N"misc/gld" xx.o -o xx
```

GLDv2 implements the following functions on behalf of the device-specific driver:

- `open(9E)`
- `close(9E)`
- `put(9E)`, required for STREAMS
- `srv(9E)`, required for STREAMS
- `getinfo(9E)`

The `mi_idname` element of the `module_info(9S)` structure is a string that specifies the name of the driver. This string must exactly match the name of the driver module as defined in the file system.

The read-side `qinit(9S)` structure should specify the following elements:

```c
qi_putp NULL
qi_srvp gld_rsrv
qi_qopen gld_open
qi_qclose gld_close
```

The write-side `qinit(9S)` structure should specify these elements:

```c
qi_putp gld_wput
qi_srvp gld_wsrv
qi_qopen NULL
qi_qclose NULL
```

The `devo_getinfo` element of the `dev_ops(9S)` structure should specify `gld_getinfo` as the `getinfo(9E)` routine.
The driver's `attach(9E)` function associates the hardware-specific device driver with the GLDv2 facility. `attach()` then prepares the device and driver for use.

The `attach(9E)` function allocates a `gld_mac_info(9S)` structure using `gld_mac_alloc()`. The driver usually needs to save more information per device than is defined in the `macinfo` structure. The driver should allocate the additional required data structure and save a pointer to the structure in the `gldm_private` member of the `gld_mac_info(9S)` structure.

The `attach(9E)` routine must initialize the `macinfo` structure as described in the `gld_mac_info(9S)` man page. The `attach()` routine should then call `gld_register()` to link the driver with the GLDv2 module. The driver should map registers if necessary and be fully initialized and prepared to accept interrupts before calling `gld_register()`. The `attach(9E)` function should add interrupts but should not enable the device to generate these interrupts. The driver should reset the hardware before calling `gld_register()` to ensure the hardware is quiescent. A device must not be put into a state where the device might generate an interrupt before `gld_register()` is called. The device is started later when GLDv2 calls the driver's `gldm_start()` entry point, which is described in the `gld(9E)` man page. After `gld_register()` succeeds, the `gld(9E)` entry points might be called by GLDv2 at any time.

The `attach(9E)` routine should return `DDI_SUCCESS` if `gld_register()` succeeds. If `gld_register()` fails, `DDI_FAILURE` is returned. If a failure occurs, the `attach(9E)` routine should deallocate any resources that were allocated before `gld_register()` was called. The `attach` routine should then also return `DDI_FAILURE`. A failed `macinfo` structure should never be reused. Such a structure should be deallocated using `gld_mac_free()`.

The `detach(9E)` function should attempt to unregister the driver from GLDv2 by calling `gld_unregister()`. For more information about `gld_unregister()`, see the `gld(9F)` man page. The `detach(9E)` routine can get a pointer to the needed `gld_mac_info(9S)` structure from the device's private data using `ddi_get_driver_private(9F)`. `gld_unregister()` checks certain conditions that could require that the driver not be detached. If the checks fail, `gld_unregister()` returns `DDI_FAILURE`, in which case the driver's `detach(9E)` routine must leave the device operational and return `DDI_FAILURE`.

If the checks succeed, `gld_unregister()` ensures that the device interrupts are stopped. The driver's `gldm_stop()` routine is called if necessary. The driver is unlinked from the GLDv2 framework. `gld_unregister()` then returns `DDI_SUCCESS`. In this case, the `detach(9E)` routine should remove interrupts and use `gld_mac_free()` to deallocate any `macinfo` data structures that were allocated in the `attach(9E)` routine. The `detach()` routine should then return `DDI_SUCCESS`. The routine must remove the interrupt before calling `gld_mac_free()`.

**GLDv2 Network Statistics**

Oracle Solaris network drivers must implement statistics variables. GLDv2 tallies some network statistics, but other statistics must be counted by each GLDv2-based driver. GLDv2 provides
support for GLDv2-based drivers to report a standard set of network driver statistics. Statistics are reported by GLDv2 using the `kstat(7D)` and `kstat(9S)` mechanisms. The `DL_GET_STATISTICS_REQ` DLPI command can also be used to retrieve the current statistics counters. All statistics are maintained as unsigned. The statistics are 32 bits unless otherwise noted.

GLDv2 maintains and reports the following statistics.

- **rbytes64**: Total bytes successfully received on the interface. Stores 64-bit statistics.
- **rbytes**: Total bytes successfully received on the interface.
- **obytes64**: Total bytes that have requested transmission on the interface. Stores 64-bit statistics.
- **obytes**: Total bytes that have requested transmission on the interface.
- **ipackets64**: Total packets successfully received on the interface. Stores 64-bit statistics.
- **ipackets**: Total packets successfully received on the interface.
- **opackets64**: Total packets that have requested transmission on the interface. Stores 64-bit statistics.
- **opackets**: Total packets that have requested transmission on the interface.
- **multircv**: Multicast packets successfully received, including group and functional addresses (`long`).
- **multixmt**: Multicast packets requested to be transmitted, including group and functional addresses (`long`).
- **brdcstrcv**: Broadcast packets successfully received (`long`).
- **brdcsxmt**: Broadcast packets that have requested transmission (`long`).
- **unknowns**: Valid received packets not accepted by any stream (`long`).
- **noxmtbuf**: Packets discarded on output because transmit buffer was busy, or no buffer could be allocated for transmit (`long`).
- **blocked**: Number of times a received packet could not be put up a stream because the queue was flow-controlled (`long`).
- **xmtretry**: Times transmit was retried after having been delayed due to lack of resources (`long`).
- **promisc**: Current “promiscuous” state of the interface (string).

The device-dependent driver tracks the following statistics in a private per-instance structure. To report statistics, GLDv2 calls the driver’s `gldm_get_stats()` entry point. `gldm_get_stats()` then updates device-specific statistics in the `gld_stats(9S)` structure. See
the **gldm_get_stats**(9E) man page for more information. GLDv2 then reports the updated statistics using the named statistics variables that are shown below.

**ifspeed**  
Current estimated bandwidth of the interface in bits per second. Stores 64-bit statistics.

**media**  
Current media type in use by the device (string).

**intr**  
Number of times that the interrupt handler was called, causing an interrupt (long).

**norcvbuf**  
Number of times a valid incoming packet was known to have been discarded because no buffer could be allocated for receive (long).

**ierrors**  
Total number of packets that were received but could not be processed due to errors (long).

**oerrors**  
Total packets that were not successfully transmitted because of errors (long).

**missed**  
Packets known to have been dropped by the hardware on receive (long).

**uflo**  
Times FIFO underflowed on transmit (long).

**oflo**  
Times receiver overflowed during receive (long).

The following group of statistics applies to networks of type **DL_ETHER**. These statistics are maintained by device-specific drivers of that type, as shown previously.

**align_errors**  
Packets that were received with framing errors, that is, the packets did not contain an integral number of octets (long).

**fcs_errors**  
Packets received with CRC errors (long).

**duplex**  
Current duplex mode of the interface (string).

**carrier_errors**  
Number of times carrier was lost or never detected on a transmission attempt (long).

**collisions**  
Ethernet collisions during transmit (long).

**ex_collisions**  
Frames where excess collisions occurred on transmit, causing transmit failure (long).

**tx_late_collisions**  
Number of times a transmit collision occurred late, that is, after 512 bit times (long).

**defer_xmts**  
Packets without collisions where first transmit attempt was delayed because the medium was busy (long).

**first_collisions**  
Packets successfully transmitted with exactly one collision.

**multi_collisions**  
Packets successfully transmitted with multiple collisions.

**sqe_errors**  
Number of times that SQE test error was reported.
**macxmt_errors**  
Packets encountering transmit MAC failures, except carrier and collision failures.

**macrcv_errors**  
Packets received with MAC errors, except align_errors, fcs_errors, and too.long_errors.

**tooolong_errors**  
Packets received larger than the maximum allowed length.

**runt_errors**  
Packets received smaller than the minimum allowed length (long).

The following group of statistics applies to networks of type DL_TPR. These statistics are maintained by device-specific drivers of that type, as shown above.

**line_errors**  
Packets received with non-data bits or FCS errors.

**burst_errors**  
Number of times an absence of transitions for five half-bit timers was detected.

**signal_losses**  
Number of times loss of signal condition on the ring was detected.

**ace_errors**  
Number of times that an AMP or SMP frame, in which A is equal to C is equal to 0, is followed by another SMP frame without an intervening AMP frame.

**internal_errors**  
Number of times the station recognized an internal error.

**lost_frame_errors**  
Number of times the TRR timer expired during transmit.

**frame_copied_errors**  
Number of times a frame addressed to this station was received with the FS field 'A' bit set to 1.

**token_errors**  
Number of times the station acting as the active monitor recognized an error condition that needed a token transmitted.

**freq_errors**  
Number of times the frequency of the incoming signal differed from the expected frequency.

The following group of statistics applies to networks of type DL_FDDI. These statistics are maintained by device-specific drivers of that type, as shown above.

**mac_errors**  
Frames detected in error by this MAC that had not been detected in error by another MAC.

**mac_lost_errors**  
Frames received with format errors such that the frame was stripped.

**mac_tokens**  
Number of tokens that were received, that is, the total of non-restricted and restricted tokens.

**mac_tvx_expired**  
Number of times that TVX has expired.

**mac_late**  
Number of TRT expirations since either this MAC was reset or a token was received.
mac_ring_ops
Number of times the ring has entered the “Ring Operational” state from the “Ring Not Operational” state.

GLDv2 Declarations and Data Structures

This section describes the gld_mac_info(9S) and gld_stats structures.

gld_mac_info Structure

The GLDv2 MAC information (gld_mac_info) structure is the main data interface that links the device-specific driver with GLDv2. This structure contains data required by GLDv2 and a pointer to an optional additional driver-specific information structure.

Allocate the gld_mac_info structure using gld_mac_alloc(). Deallocate the structure using gld_mac_free(). Drivers must not make any assumptions about the length of this structure, which might vary in different releases of the Oracle Solaris OS, GLDv2, or both. Structure members private to GLDv2, not documented here, should neither be set nor be read by the device-specific driver.

The gld_mac_info(9S) structure contains the following fields.

caddr_t gldm_private; /* Driver private data */
int (*gldm_reset)(); /* Reset device */
int (*gldm_start)(); /* Start device */
int (*gldm_stop)(); /* Stop device */
int (*gldm_set_mac_addr)(); /* Set device phys addr */
int (*gldm_set_multicast)(); /* Set/delete multicast addr */
int (*gldm_set_promiscuous)(); /* Set/reset promiscuous mode */
int (*gldm_send)(); /* Transmit routine */
uint_t (*gldm_intr)(); /* Interrupt handler */
int (*gldm_get_stats)(); /* Get device statistics */
char *gldm_ident; /* Driver identity string */
uint32_t gldm_type; /* Device type */
uint32_t gldm_minpkt; /* Minimum packet size */
uint32_t gldm_maxpkt; /* Maximum packet size */
uint32_t gldm_addrlen; /* Physical address length */
int32_t gldm_saplen; /* SAP length for DL_INFO_ACK */
unsigned char *gldm_broadcast_addr; /* Physical broadcast addr */
unsigned char *gldm_vendor_addr; /* Factory MAC address */
t_uscalar_t gldm_ppa; /* Physical Point of */
dev_info_t *gldm_devinfo; /* Pointer to device’s */
ddi_iblock_cookie_t gldm_cookie; /* Device’s interrupt */
The `gldm_private` structure member is visible to the device driver. `gldm_private` is also private to the device-specific driver. `gldm_private` is not used or modified by GLDv2. Conventionally, `gldm_private` is used as a pointer to private data, pointing to a per-instance data structure that is both defined and allocated by the driver.

The following group of structure members must be set by the driver before calling `gld_register()`, and should not thereafter be modified by the driver. Because `gld_register()` might use or cache the values of structure members, changes made by the driver after calling `gld_register()` might cause unpredictable results. For more information on these structures, see the `gld(9E)` man page.

- `gldm_reset` Pointer to driver entry point.
- `gldm_start` Pointer to driver entry point.
- `gldm_stop` Pointer to driver entry point.
- `gldm_set_mac_addr` Pointer to driver entry point.
- `gldm_set_multicast` Pointer to driver entry point.
- `gldm_set_promiscuous` Pointer to driver entry point.
- `gldm_send` Pointer to driver entry point.
- `gldm_intr` Pointer to driver entry point.
- `gldm_get_stats` Pointer to driver entry point.
- `gldm_ioctl` Pointer to driver entry point. This pointer is allowed to be null.
- `gldm_ident` Pointer to a string that contains a short description of the device. This pointer is used to identify the device in system messages.
- `gldm_type` Type of device the driver handles. GLDv2 currently supports the following values:
  - DL_ETHER (ISO 8802-3 (IEEE 802.3) and Ethernet Bus)
  - DL_TPR (IEEE 802.5 Token Passing Ring)
  - DL_FDDI (ISO 9314-2 Fibre Distributed Data Interface)

  This structure member must be correctly set for GLDv2 to function properly.

- `gldm_minpkt` Minimum Service Data Unit size: the minimum packet size, not including the MAC header, that the device can transmit. This size is allowed to be zero if the device-specific driver handles any required padding.

- `gldm_maxpkt` Maximum Service Data Unit size: the maximum size of packet, not including the MAC header, that can be transmitted by the device. For Ethernet, this number is 1500.
gldm_addrlen
The length in bytes of physical addresses handled by the device. For Ethernet, Token Ring, and FDDI, the value of this structure member should be 6.

gldm_saplen
The length in bytes of the SAP address used by the driver. For GLDv2-based drivers, the length should always be set to -2. A length of -2 indicates that 2-byte SAP values are supported and that the SAP appears after the physical address in a DLSAP address. See Appendix A.2, “Message DL_INFO_ACK,” in the DLPI specification for more details.

gldm_broadcast_addr
Pointer to an array of bytes of length gldm_addrlen containing the broadcast address to be used for transmit. The driver must provide space to hold the broadcast address, fill the space with the appropriate value, and set gldm_broadcast_addr to point to the address. For Ethernet, Token Ring, and FDDI, the broadcast address is normally 0xFF-FF-FF-FF-FF-FF.

gldm_vendor_addr
Pointer to an array of bytes of length gldm_addrlen that contains the vendor-provided network physical address of the device. The driver must provide space to hold the address, fill the space with information from the device, and set gldm_vendor_addr to point to the address.

gldm_ppa
PPA number for this instance of the device. The PPA number should always be set to the instance number that is returned from ddi_get_instance(9F).

gldm_devinfo
Pointer to the dev_info node for this device.

gldm_cookie
Interrupt block cookie returned by one of the following routines:
- ddi_get_iblock_cookie(9F)
- ddi_add_intr(9F)
- ddi_get_soft_iblock_cookie(9F)
- ddi_add_softcintr(9F)

This cookie must correspond to the device’s receive-interrupt, from which gld_recv() is called.

**gld_stats Structure**

After calling gldm_get_stats(), a GLDv2-based driver uses the (gld_stats) structure to communicate statistics and state information to GLDv2. See the gld(9E) and gld(7D) man pages. The members of this structure, having been filled in by the GLDv2-based driver, are used when GLDv2 reports the statistics. In the tables below, the name of the statistics variable reported by GLDv2 is noted in the comments. See the gld(7D) man page for a more detailed description of the meaning of each statistic.
Drivers must not make any assumptions about the length of this structure. The structure length might vary in different releases of the Oracle Solaris OS, GLDv2, or both. Structure members private to GLDv2, which are not documented here, should not be set or be read by the device-specific driver.

The following structure members are defined for all media types:

- `uint64_t gls_speed; /* ifspeed */`
- `uint32_t gls_media; /* media */`
- `uint32_t gls_intr; /* intr */`
- `uint32_t gls_norcvbuf; /* norcvbuf */`
- `uint32_t gls_errrcv; /* ierrors */`
- `uint32_t gls_errnoxmt; /* oerrors */`
- `uint32_t gls_missed; /* missed */`
- `uint32_t gls_underflow; /* uflo */`
- `uint32_t gls_overflow; /* oflo */`

The following structure members are defined for media type DL_ETHER:

- `uint32_t gls_frame; /* align_errors */`
- `uint32_t gls_crc; /* fcs_errors */`
- `uint32_t gls_duplex; /* duplex */`
- `uint32_t gls_nocarrier; /* carrier_errors */`
- `uint32_t gls_collisions; /* collisions */`
- `uint32_t gls_excoll; /* ex_collisions */`
- `uint32_t gls_xmtlatecoll; /* tx_late_collisions */`
- `uint32_t gls_defer; /* defer_xmts */`
- `uint32_t gls_dot3_first_coll; /* first_collisions */`
- `uint32_t gls_dot3_multi_coll; /* multi_collisions */`
- `uint32_t gls_dot3_mac_lmt_error; /* macmt_errors */`
- `uint32_t gls_dot3_mac_rcv_error; /* macrcv_errors */`
- `uint32_t gls_dot3_frame_too_long; /* toolong_errors */`
- `uint32_t gls_short; /* runt_errors */`

The following structure members are defined for media type DL_TPR:

- `uint32_t gls_dot5_line_error /* line_errors */`
- `uint32_t gls_dot5_burst_error /* burst_errors */`
- `uint32_t gls_dot5_ace_error /* ace_errors */`
- `uint32_t gls_dot5_internal_error /* internal_errors */`
- `uint32_t gls_dot5_lost_frame_error /* lost_frame_errors */`
- `uint32_t gls_dot5_frame_copied_error /* frame_copied_errors */`
- `uint32_t gls_dot5_token_error /* token_errors */`
- `uint32_t gls_dot5_freq_error /* freq_errors */`

The following structure members are defined for media type DL_FDDI:

- `uint32_t gls_fddi_mac_error; /* mac_errors */`
- `uint32_t gls_fddi_mac_lost; /* mac_lost_errors */`
- `uint32_t gls_fddi_mac_token; /* mac_tokens */`
- `uint32_t gls_fddi_mac_tvx_exp; /* tvx_expired */`
- `uint32_t gls_fddi_mac_late; /* mac_late */`
Most of the above statistics variables are counters that denote the number of times that the particular event was observed. The following statistics do not represent the number of times:

- **glds_speed**: Estimate of the interface’s current bandwidth in bits per second. This object should contain the nominal bandwidth for those interfaces that do not vary in bandwidth or where an accurate estimate cannot be made.

- **glds_media**: Type of media (wiring) or connector used by the hardware. The following media names are supported:
  - GLDM_AUI
  - GLDM_BNC
  - GLDM_TP
  - GLDM_10BT
  - GLDM_100BT
  - GLDM_100BTX
  - GLDM_100BT4
  - GLDM_RING4
  - GLDM_RING16
  - GLDM_FIBER
  - GLDM_PHYMII
  - GLDM_UNKNOWN

- **glds_duplex**: Current duplex state of the interface. Supported values are GLD_DUPLEX_HALF and GLD_DUPLEX_FULL. GLD_DUPLEX_UNKNOWN is also allowed.

### GLDv2 Function Arguments

The following arguments are used by the GLDv2 routines.

- **macinfo**: Pointer to a `gldmac_info(9S)` structure.

- **macaddr**: Pointer to the beginning of a character array that contains a valid MAC address. The array is of the length specified by the driver in the `gldm_addrlen` element of the `gldmac_info(9S)` structure.

- **multicastaddr**: Pointer to the beginning of a character array that contains a multicast, group, or functional address. The array is of the length specified by the driver in the `gldm_addrlen` element of the `gldmac_info(9S)` structure.

- **multilflag**: Flag indicating whether to enable or disable reception of the multicast address. This argument is specified as GLD_MULTI_ENABLE or GLD_MULTI_DISABLE.
promiscflag  Flag indicating what type of promiscuous mode, if any, is to be enabled. This argument is specified as GLD_MAC_PROMISC_PHYS, GLD_MAC_PROMISC_MULTI, or GLD_MAC_PROMISC_NONE.

mp  gld_ioctl() uses mp as a pointer to a STREAMS message block containing the ioctl to be executed. gldm_send() uses mp as a pointer to a STREAMS message block containing the packet to be transmitted. gld_recv() uses mp as a pointer to a message block containing a received packet.

stats  Pointer to a gld_stats(9S) structure to be filled in with the current values of statistics counters.

q  Pointer to the queue(9S) structure to be used in the reply to the ioctl.

dip  Pointer to the device's dev_info structure.

name  Device interface name.

GLDv2 Entry Points

Entry points must be implemented by a device-specific network driver that has been designed to interface with GLDv2.

The gld_mac_info(9S) structure is the main structure for communication between the device-specific driver and the GLDv2 module. See the gld(7D) man page. Some elements in that structure are function pointers to the entry points that are described here. The device-specific driver must, in its attach(9E) routine, initialize these function pointers before calling gld_register().

gldm_reset() Entry Point

int prefix_reset(gld_mac_info_t *macinfo);

gldm_reset() resets the hardware to its initial state.

gldm_start() Entry Point

int prefix_start(gld_mac_info_t *macinfo);

gldm_start() enables the device to generate interrupts. gldm_start() also prepares the driver to call gld_recv() to deliver received data packets to GLDv2.

gldm_stop() Entry Point

int prefix_stop(gld_mac_info_t *macinfo);
gladm_stop() disables the device from generating any interrupts and stops the driver from calling gld_recv() for delivering data packets to GLDv2. GLDv2 depends on the gladm_stop() routine to ensure that the device will no longer interrupt. gladm_stop() must do so without fail. This function should always return GLD_SUCCESS.

**gladm_set_mac_addr() Entry Point**

```c
int prefix_set_mac_addr(gld_mac_info_t *macinfo, unsigned char *macaddr);
```

`gladm_set_mac_addr()` sets the physical address that the hardware is to use for receiving data. This function enables the device to be programmed through the passed MAC address `macaddr`. If sufficient resources are currently not available to carry out the request, `gladm_set_mac_addr()` should return GLD_NORESOURCES. If the requested function is not supported, `gladm_set_mac_addr()` should return GLD_NOTSUPPORTED.

**gladm_set_multicast() Entry Point**

```c
int prefix_set_multicast(gld_mac_info_t *macinfo,
                        unsigned char *multicastaddr, int multiflag);
```

`gladm_set_multicast()` enables and disables device-level reception of specific multicast addresses. If the third argument `multiflag` is set to GLD_MULTI_ENABLE, then `gladm_set_multicast()` sets the interface to receive packets with the multicast address. `gladm_set_multicast()` uses the multicast address that is pointed to by the second argument. If `multiflag` is set to GLD_MULTI_DISABLE, the driver is allowed to disable reception of the specified multicast address.

This function is called whenever GLDv2 wants to enable or disable reception of a multicast, group, or functional address. GLDv2 makes no assumptions about how the device does multicast support and calls this function to enable or disable a specific multicast address. Some devices might use a hash algorithm and a bitmask to enable collections of multicast addresses. This procedure is allowed, and GLDv2 filters out any superfluous packets. If disabling an address could result in disabling more than one address at the device level, the device driver should keep any necessary information. This approach avoids disabling an address that GLDv2 has enabled but not disabled.

`gladm_set_multicast()` is not called to enable a particular multicast address that is already enabled. Similarly, `gladm_set_multicast()` is not called to disable an address that is not currently enabled. GLDv2 keeps track of multiple requests for the same multicast address. GLDv2 only calls the driver’s entry point when the first request to enable, or the last request to disable, a particular multicast address is made. If sufficient resources are currently not available to carry out the request, the function should return GLD_NORESOURCES. The function should return GLD_NOTSUPPORTED if the requested function is not supported.
gladm_set_promiscuous() **Entry Point**

```c
int prefix_set_promiscuous(gld_mac_info_t *macinfo, int promiscflag);
```

The **gladm_set_promiscuous()** function enables and disables promiscuous mode. This function is called whenever GLDv2 wants to enable or disable the reception of all packets on the medium. The function can also be limited to multicast packets on the medium. If the second argument `promiscflag` is set to the value of `GLD_MAC_PROMISC_PHYS`, then the function enables physical-level promiscuous mode. Physical-level promiscuous mode causes the reception of all packets on the medium. If `promiscflag` is set to `GLD_MAC_PROMISC_MULTI`, then reception of all multicast packets are enabled. If `promiscflag` is set to `GLD_MAC_PROMISC_NONE`, then promiscuous mode is disabled.

In promiscuous multicast mode, drivers for devices without multicast-only promiscuous mode must set the device to physical promiscuous mode. This approach ensures that all multicast packets are received. In this case, the routine should return `GLD_SUCCESS`. The GLDv2 software filters out any superfluous packets. If sufficient resources are currently not available to carry out the request, the function should return `GLD_NORESOURCES`. The `gladm_set_promiscuous()` function should return `GLD_NOTSUPPORTED` if the requested function is not supported.

For forward compatibility, `gladm_set_promiscuous()` routines should treat any unrecognized values for `promiscflag` as though these values were `GLD_MAC_PROMISC_PHYS`.

**gladm_send()** **Entry Point**

```c
int prefix_send(gld_mac_info_t *macinfo, mblk_t *mp);
```

**gladm_send()** queues a packet to the device for transmission. This routine is passed a STREAMS message containing the packet to be sent. The message might include multiple message blocks. The `send()` routine must traverse all the message blocks in the message to access the entire packet to be sent. The driver should be prepared to handle and skip over any zero-length message continuation blocks in the chain. The driver should also check that the packet does not exceed the maximum allowable packet size. The driver must pad the packet, if necessary, to the minimum allowable packet size. If the send routine successfully transmits or queues the packet, `GLD_SUCCESS` should be returned.

The send routine should return `GLD_NORESOURCES` if the packet for transmission cannot be immediately accepted. In this case, GLDv2 retries later. If `gladm_send()` ever returns `GLD_NORESOURCES`, the driver must call `gld_sched()` at a later time when resources have become available. This call to `gld_sched()` informs GLDv2 to retry packets that the driver previously failed to queue for transmission. (If the driver’s `gladm_stop()` routine is called, the driver is absolved from this obligation until the driver returns `GLD_NORESOURCES` from the `gladm_send()` routine. However, extra calls to `gld_sched()` do not cause incorrect operation.)

If the driver’s send routine returns `GLD_SUCCESS`, then the driver is responsible for freeing the message when the message is no longer needed. If the hardware uses DMA to read the data...
directly, the driver must not free the message until the hardware has completely read the data. In this case, the driver can free the message in the interrupt routine. Alternatively, the driver can reclaim the buffer at the start of a future send operation. If the send routine returns anything other than GLD_SUCCESS, then the driver must not free the message. Return GLD_NOLINK if gldm_send() is called when there is no physical connection to the network or link partner.

**gldm_intr() Entry Point**

```
int prefix_intr(gld_mac_info_t *macinfo);
```

**gldm_intr()** is called when the device might have interrupted. Because interrupts can be shared with other devices, the driver must check the device status to determine whether that device actually caused the interrupt. If the device that the driver controls did not cause the interrupt, then this routine must return DDI_INTR_UNCLAIMED. Otherwise, the driver must service the interrupt and return DDI_INTRCLAIMED. If the interrupt was caused by successful receipt of a packet, this routine should put the received packet into a STREAMS message of type M_DATA and pass that message to gld_recv().

**gld_recv()** passes the inbound packet upstream to the appropriate next layer of the network protocol stack. The routine must correctly set the b_rptr and b_wptr members of the STREAMS message before calling gld_recv().

The driver should avoid holding mutex or other locks during the call to gld_recv(). In particular, locks that could be taken by a transmit thread must not be held during a call to gld_recv(). In some cases, the interrupt thread that calls gld_recv() sends an outgoing packet, which results in a call to the driver’s gldm_send() routine. If gldm_send() tries to acquire a mutex that is held by gldm_intr() when gld_recv() is called, a panic occurs due to recursive mutex entry. If other driver entry points attempt to acquire a mutex that the driver holds across a call to gld_recv(), deadlock can result.

The interrupt code should increment statistics counters for any errors. Errors include the failure to allocate a buffer that is needed for the received data and any hardware-specific errors, such as CRC errors or framing errors.

**gldm_get_stats() Entry Point**

```
int prefix_get_stats(gld_mac_info_t *macinfo, struct gld_stats *stats);
```

**gldm_get_stats()** gathers statistics from the hardware, driver private counters, or both, and updates the gld_stats(9S) structure pointed to by stats. This routine is called by GLDv2 for statistics requests. GLDv2 uses the gldm_get_stats() mechanism to acquire device-dependent statistics from the driver before GLDv2 composes the reply to the statistics request. See the gld_stats(9S), gld(7D), and qreply(9F) man pages for more information about defined statistics counters.
`gldm_ioctl()` Entry Point

```c
int prefix_ioctl(gld_mac_info_t *macinfo, queue_t *q, mblk_t *mp);
```

`gldm_ioctl()` implements any device-specific `ioctl` commands. This element is allowed to be null if the driver does not implement any `ioctl` functions. The driver is responsible for converting the message block into an `ioctl` reply message and calling the `qreply(9F)` function before returning `GLD_SUCCESS`. This function should always return `GLD_SUCCESS`. The driver should report any errors as needed in a message to be passed to `qreply(9F)`. If the `gldm_ioctl` element is specified as NULL, GLDv2 returns a message of type `M_IOCNAK` with an error of `EINVAL`.

GLDv2 Return Values

Some entry point functions in GLDv2 can return the following values, subject to the restrictions above:

- `GLD_BADARG` If the function detected an unsuitable argument, for example, a bad multicast address, a bad MAC address, or a bad packet
- `GLD_FAILURE` On hardware failure
- `GLD_SUCCESS` On success

GLDv2 Service Routines

This section provides the syntax and description for the GLDv2 service routines.

`gld_mac_alloc()` Function

```c
gld_mac_info_t *gld_mac_alloc(dev_info_t *dip);
```

`gld_mac_alloc()` allocates a new `gld_mac_info(9S)` structure and returns a pointer to the structure. Some of the GLDv2-private elements of the structure might be initialized before `gld_mac_alloc()` returns. All other elements are initialized to zero. The device driver must initialize some structure members, as described in the `gld_mac_info(9S)` man page, before passing the pointer to the `gld_mac_info` structure to `gld_register()`.

`gld_mac_free()` Function

```c
void gld_mac_free(gld_mac_info_t *macinfo);
```

`gld_mac_free()` frees a `gld_mac_info(9S)` structure previously allocated by `gld_mac_alloc()`.
### gld_register() Function

```c
int gld_register(dev_info_t *dip, char *name, gld_mac_info_t *macinfo);
```

`gld_register()` is called from the device driver's `attach()` routine. `gld_register()` links the GLDv2-based device driver with the GLDv2 framework. Before calling `gld_register()`, the device driver's `attach()` routine uses `gld_mac_alloc()` to allocate a `gld_mac_info` structure, and then initializes several structure elements. See `gld_mac_info` for more information. A successful call to `gld_register()` performs the following actions:

- Links the device-specific driver with the GLDv2 system
- Sets the device-specific driver's private data pointer, using `ddi_set_driver_private()` to point to the `macinfo` structure
- Creates the minor device node
- Returns DDI_SUCCESS

The device interface name passed to `gld_register()` must exactly match the name of the driver module as that name exists in the file system.

The driver's `attach()` routine should return DDI_SUCCESS if `gld_register()` succeeds. If `gld_register()` does not return DDI_SUCCESS, the `attach()` routine should deallocate any allocated resources before calling `gld_register()`, and then return DDI_FAILURE.

### gld_unregister() Function

```c
int gld_unregister(gld_mac_info_t *macinfo);
```

`gld_unregister()` is called by the device driver's `detach()` function, and if successful, performs the following tasks:

- Ensures that the device's interrupts are stopped, calling the driver's `gldm_stop()` routine if necessary
- Removes the minor device node
- Unlinks the device-specific driver from the GLDv2 system
- Returns DDI_SUCCESS

If `gld_unregister()` returns DDI_SUCCESS, the `detach()` routine should deallocate any data structures allocated in the `attach()` routine, using `gld_mac_free()` to deallocate the `macinfo` structure, and return DDI_SUCCESS. If `gld_unregister()` does not return DDI_SUCCESS, the driver's `detach()` routine must leave the device operational and return DDI_FAILURE.

### gld_recv() Function

```c
void gld_recv(gld_mac_info_t *macinfo, mblk_t *mp);
```

gld_recv() is called by the driver’s interrupt handler to pass a received packet upstream. The driver must construct and pass a STREAMS M_DATA message containing the raw packet. gld_recv() determines which STREAMS queues should receive a copy of the packet, duplicating the packet if necessary. gld_recv() then formats a DL_UNITDATA_IND message, if required, and passes the data up all appropriate streams.

The driver should avoid holding mutex or other locks during the call to gld_recv(). In particular, locks that could be taken by a transmit thread must not be held during a call to gld_recv(). The interrupt thread that calls gld_recv() in some cases carries out processing that includes sending an outgoing packet. Transmission of the packet results in a call to the driver’s gldm_send() routine. If gldm_send() tries to acquire a mutex that is held by gldm_intr() when gld_recv() is called, a panic occurs due to a recursive mutex entry. If other driver entry points attempt to acquire a mutex that the driver holds across a call to gld_recv(), deadlock can result.

**gld_sched() Function**

```c
void gld_sched(gld_mac_info_t *macinfo);
```

gld_sched() is called by the device driver to reschedule stalled outbound packets. Whenever the driver’s gldm_send() routine returns GLD_NORESOURCES, the driver must call gld_sched() to inform the GLDv2 framework to retry previously unsendable packets. gld_sched() should be called as soon as possible after resources become available so that GLDv2 resumes passing outbound packets to the driver’s gldm_send() routine. (If the driver’s gldm_stop() routine is called, the driver need not retry until GLD_NORESOURCES is returned from gldm_send(). However, extra calls to gld_sched() do not cause incorrect operation.)

**gld_intr() Function**

```c
uint_t gld_intr(caddr_t);
```

gld_intr() is GLDv2’s main interrupt handler. Normally, gld_intr() is specified as the interrupt routine in the device driver’s call to ddi_add_intr(9F). The argument to the interrupt handler is specified as int_handler_arg in the call to ddi_add_intr(9F). This argument must be a pointer to the gld_mac_info(9S) structure. gld_intr(), when appropriate, calls the device driver’s gldm_intr() function, passing that pointer to the gld_mac_info(9S) structure. However, to use a high-level interrupt, the driver must provide its own high-level interrupt handler and trigger a soft interrupt from within the handler. In this case, gld_intr() would normally be specified as the soft interrupt handler in the call to ddi_add_softintr(). gld_intr() returns a value that is appropriate for an interrupt handler.
This chapter describes how to write a client USB device driver using the USB 2.0 framework for the Oracle Solaris environment. This chapter discusses the following topics:

- "USB in the Oracle Solaris Environment" on page 427
- "Binding Client Drivers" on page 430
- "Basic Device Access" on page 435
- "Device Communication" on page 438
- "Device State Management" on page 448
- "Utility Functions" on page 456
- "Sample USB Device Drivers" on page 459

USB in the Oracle Solaris Environment

The Oracle Solaris USB architecture includes the USB 2.0 framework and USB client drivers.

USB 2.0 Framework

The USB 2.0 framework is a service layer that presents an abstract view of USB devices to USBA-compliant client drivers. The framework enables USBA-compliant client drivers to manage their USB devices. The USB 2.0 framework supports the USB 2.0 specification except for high-speed isochronous pipes. For information on the USB 2.0 specification, see http://www.usb.org/home.

The USBA 2.0 framework is platform-independent. The Oracle Solaris USB architecture is shown in the following figure. The USBA 2.0 framework is the USBA layer in the figure. This layer interfaces through a hardware-independent host controller driver interface to hardware-specific host controller drivers. The host controller drivers access the USB physical devices through the host controllers they manage.
The USB 2.0 framework is not a device driver itself. This chapter describes the client drivers shown in Figure 20–1 and Figure 20–2. The client drivers interact with various kinds of USB devices such as mass storage devices, printers, and human interface devices. The hub driver is a client driver that is also a nexus driver. The hub driver enumerates devices on its ports and creates devinfo nodes for those devices and then attaches the client drivers. This chapter does not describe how to write a hub driver.

USB drivers have the same structure as any other Oracle Solaris driver. USB drivers can be block drivers, character drivers, or STREAMS drivers. USB drivers follow the calling conventions and use the data structures and routines described in the Oracle Solaris OS section 9 man pages. See Intro(9E), Intro(9F), and Intro(9S).

The difference between USB drivers and other Oracle Solaris drivers is that USB drivers call USB 2.0 framework functions to access the device instead of directly accessing the device. The USB 2.0 framework supplements the standard Oracle Solaris DDI routines. See the following figure.
Figure 20–2 shows interfaces in more detail than Figure 20–1 does. Figure 20–2 shows that the USBA is a kernel subsystem into which a client driver can call, just as a client driver can call DDI functions.

Not all systems have all of the host controller interfaces shown in Figure 20–2. OHCI (Open Host Controller Interface) hardware is most prevalent on SPARC systems and third-party USB PCI cards. UHCI (Universal Host Controller Interface) hardware is most prevalent on x86 systems. However, both OHCI and UHCI hardware can be used on any system. When EHCI (Enhanced Host Controller Interface) hardware is present, the EHCI hardware is on the same card and shares the same ports with either OHCI or UHCI.

The host controllers, host controller drivers, and HCDI make up a transport layer that is commanded by the USBA. You cannot directly call into the OHCI, EHCI, or UHCI. You call into them indirectly through the platform-independent USBA interface.
This section discusses binding a driver to a device. It discusses compatible device names for devices with single interfaces and devices with multiple interfaces.

How USB Devices Appear to the System

A USB device can support multiple configurations. Only one configuration is active at any given time. The active configuration is called the current configuration.

A configuration can have more than one interface, possibly with intervening interface-associations that group two or more interfaces for a function. All interfaces of a configuration are active simultaneously. Different interfaces might be operated by different device drivers.

An interface can represent itself to the host system in different ways by using alternate settings. Only one alternate setting is active for any given interface.

Each alternate setting provides device access through endpoints. Each endpoint has a specific purpose. The host system communicates with the device by establishing a communication channel to an endpoint. This communication channel is called a pipe.

USB Devices and the Oracle Solaris Device Tree

If a USB device has one configuration, one interface, and device class zero, the device is represented as a single device node. If a USB device has multiple interfaces, the device is represented as a hierarchical device structure. In a hierarchical device structure, the device node for each interface is a child of the top-level device node. An example of a device with multiple interfaces is an audio device that presents simultaneously to the host computer both an audio control interface and an audio streaming interface. The audio control interface and the audio streaming interface each could be controlled by its own driver.

Compatible Device Names

The Oracle Solaris software builds an ordered list of compatible device names for USB binding based on identification information kept within each device. This information includes device class, subclass, vendor ID, product ID, revision, and protocol. See http://www.usb.org/home for a list of USB classes and subclasses.

This name hierarchy enables binding to a general driver if a more device-specific driver is not available. An example of a general driver is a class-specific driver. Device names that begin with usb1f designate single interface devices. See Example 20–1 for examples. The USBA 2.0 framework defines all compatible names for a device. Use the prtconf command to display these device names, as shown in Example 20–2.
The following example shows an example of compatible device names for a USB mouse device. This mouse device represents a combined node entirely operated by a single driver. The USBA 2.0 framework gives this device node the names shown in the example, in the order shown.

**EXAMPLE 20–1  USB Mouse Compatible Device Names**

1. `usb430,100.102` Vendor 430, product 100, revision 102
2. `usb430,100` Vendor 430, product 100
3. `usbif430, class3.1.2` Vendor 430, class 3, subclass 1, protocol 2
4. `usbif430, class3.1` Vendor 430, class 3, subclass 1
5. `usbif430, class3` Vendor 430, class 3
6. `usbif, class3.1.2` Class 3, subclass 1, protocol 2
7. `usbif, class3.1` Class 3, subclass 1
8. `usbif, class3` Class 3

Note that the names in the above example progress from the most specific to the most general. Entry 1 binds only to a particular revision of a specific product from a particular vendor. Entries 3, 4, and 5 are for class 3 devices manufactured by vendor 430. Entries 6, 7, and 8 are for class 3 devices from any vendor. The binding process looks for a match on the name from the top name down. To bind, drivers must be added to the system with an alias that matches one of these names. To get a list of compatible device names to which to bind when you add your driver, check the `compatible` property of the device in the output from the `prtconf -vp` command.

The following example shows compatible property lists for a keyboard and a mouse. Use the `prtconf -D` command to display the bound driver.

**EXAMPLE 20–2  Compatible Device Names Shown by the Print Configuration Command**

```bash
# prtconf -vD | grep compatible
compatible: 'usb430,5.200' + 'usb430,5' + 'usbif430, class3.1.1'
+ 'usbif430, class3.1' + 'usbif, class3' + 'usbif, class3.1.1' + 'usbif, class3.1' + 'usbif, class3'
   compatible: 'usb2222,2071.200' + 'usb2222,2071' +
   'usbif2222, class3.1.2' + 'usbif2222, class3.1' + 'usbif2222, class3' +
   'usbif, class3.1.2' + 'usbif, class3.1' + 'usbif, class3'
```

Use the most specific name you can to more accurately identify a driver for a device or group of devices. To bind drivers written for a specific revision of a specific product, use the most specific name match possible. For example, if you have a USB mouse driver written by vendor 430 for revision 102 of their product 100, use the following command to add that driver to the system:

```
add_drv -n -i "usb430,100.102" specific_mouse_driver
```

To add a driver written for any USB mouse (class 3, subclass 1, protocol 2) from vendor 430, use the following command:

```
add_drv -n -i "usbif430, class3.1.2" more_generic_mouse_driver
```
If you install both of these drivers and then connect a compatible device, the system binds the correct driver to the connected device. For example, if you install both of these drivers and then connect a vendor 430, model 100, revision 102 device, this device is bound to `specific_mouse_driver`. If you connect a vendor 430, model 98 device, this device is bound to `more_generic_mouse_driver`. If you connect a mouse from another vendor, this device also is bound to `more_generic_mouse_driver`. If multiple drivers are available for a specific device, the driver binding framework selects the driver with the first matching compatible name in the compatible names list.

**Devices With Multiple Interfaces**

Composite devices are devices that support multiple interfaces. Composite devices have a list of compatible names for each interface. This compatible names list ensures that the best available driver is bound to the interface. The most general multiple interface entry is `usb,device`.

For a USB audio composite device, the compatible names are as follows:

1. `‘usb471,101.100’` Vendor 471, product 101, revision 100
2. `‘usb471,101’` Vendor 471, product 101
3. `‘usb,device’` Generic USB device

The name `usb,device` is a compatible name that represents any whole USB device. The `usb_mid(7D)` driver (USB multiple-interface driver) binds to the `usb,device` device node if no other driver has claimed the whole device. The `usb_mid` driver creates a child device node for each interface of the physical device. The `usb_mid` driver also generates a set of compatible names for each interface. Each of these generated compatible names begins with `usbif`. The system then uses these generated compatible names to find the best driver for each interface. In this way, different interfaces of one physical device can be bound to different drivers.

For example, the `usb_mid` driver binds to a multiple-interface audio device through the `usb,device` node name of that audio device. The `usb_mid` driver then creates interface-specific device nodes. Each of these interface-specific device nodes has its own compatible name list. For an audio control interface node, the compatible name list might look like the list shown in the following example.

**EXAMPLE 20–3** USB Audio Compatible Device Names

1. `‘usbif471,101.100.config1.0’` Vend 471, prod 101, rev 100, cnfg 1, iface 0
2. `‘usbif471,101.config1.0’` Vend 471, product 101, config 1
3. `‘usbif471,class1.1.0’` Vend 471, class 1, subclass 1, protocol 0
4. `‘usbif471,class1.1’` Vend 471, class 1, subclass
5. `‘usbif471,class1’` Vend 471, class 1
6. `‘usbif,class1.1.0’` Class 1, subclass 1, protocol 0
7. `‘usbif,class1.1’` Class 1, subclass 1
8. `‘usbif,class1’` Class 1
Use the following command to bind a vendor-specific, device-specific client driver named
vendor_model_audio_usb to the vendor-specific, device-specific configuration 1, interface 0
interface compatible name shown in Example 20–3.

```
add_drv -n -i "usbif471,101.config1.0" vendor_model_audio_usb
```

Use the following command to bind a class driver named audio_class_usb_if_driver to the
more general class 1, subclass 1 interface compatible name shown in Example 20–3:

```
add_drv -n -i "usbif,class1.1" audio_class_usb_if_driver
```

Use the `prtconf -D` command to show a list of devices and their drivers. In the following
example, the `prtconf -D` command shows that the usb_mid driver manages the audio device.
The usb_mid driver is splitting the audio device into interfaces. Each interface is indented
under the audio device name. For each interface shown in the indented list, the `prtconf -D`
command shows which driver manages the interface.

```
audio, instance #0 (driver name: usb.mid)
    sound-control, instance #2 (driver name: usb.ac)
    sound, instance #2 (driver name: usb.as)
    input, instance #8 (driver name: hid)
```

**Devices With Interface-Association Descriptors**

If the device includes an interface-association descriptor, the device tree can be parsed at the
following three levels:

- The **usb_mid(7D)** USB multi-interface driver binds to device level nodes of a composite
device if no vendor or class-specific driver is available.
- A client driver is bound to the interface association nodes.
- The **usb_ia(7D)** USB interface association driver is bound by default if no client driver is
  found. Then client drivers can be bound to the interface level of this interface association.

The usb_mid driver creates an ia (interface association) node for each ia. The compatible
names of ia nodes generally begin with usbia. The name usb, ia is a compatible name that
represents any ia as the tail of the compatible names. The usb_ia driver is bound to an ia node
if no other driver has claimed this ia. The usb_ia driver creates a child node for each interface.
An interface node as the child node of an ia node has the same properties with an interface
node as the child of a device node.

**EXAMPLE 20–4  USB Video Interface Association Compatible Names**

1. `usbia46d,8c9.5.config1.0` vend 46d, prod 8c9, rev 5, cnfg 1, first_if_in_ia 0
2. `usbia46d,8c9.config1.0` vend 46d, prod 8c9, cnfg 1, first_if_in_ia 0
3. `usbia46d,classe.3.0` vend 46d, class e, subclass 3, protocol 0
4. `usbia46d,classe.3` vend 46d, class e
5. `usbia46d,classe` vend 46d, class e
Use the following command to bind a vendor-specific, device-specific client driver named vendor_model_video_usb to the vendor-specific, device-specific configuration 1, first_if_in_ia 0 compatible name shown in Example 20–4:

```
add_drv -n -i "usbia46d,8c9.config1.0" vendor_model_video_usb
```

Use the following command to bind a class driver named video_class_usb_ia_driver to the more general class e compatible names shown in Example 20–4:

```
add_drv -n -i "usbia,classee" video_class_usb_ia_driver
```

In the following example, the prtconf -D command shows a device tree of a webcam with ia of video and audio. The usb_mid driver manages the device and creates two ia respectively for video and audio. A video driver usbvc is bound to the video ia, and audio drivers are bound to the interface of the audio ia.

```
miscellaneous, instance #28 (driver name: usb mid)
  video, instance #24 (driver name: usbvc)
  audio, instance #30 (driver name: usb ia)
    sound-control, instance #38 (driver name: usb ac)
    sound, instance #47 (driver name: usb_as)
```

### Checking Device Driver Bindings

The file /etc/driver_aliases contains entries for the bindings that already exist on a system. Each line of the /etc/driver_aliases file shows a driver name, followed by a space, followed by a device name. Use this file to check existing device driver bindings.

**Note** – Do not edit the /etc/driver_aliases file manually. Use the `add_drv(1M)` command to establish a binding. Use the `update_drv(1M)` command to change a binding.
Basic Device Access

This section describes how to access a USB device and how to register a client driver. This section also discusses the descriptor tree.

Before the Client Driver Is Attached

The following events take place before the client driver is attached:

1. The PROM (OBP/BIOS) and USBA framework gain access to the device before any client driver is attached.
2. The hub driver probes devices on each of its hub's ports for identity and configuration.
3. The default control pipe to each device is opened, and each device is probed for its device descriptor.
4. Compatible names properties are constructed for each device, using the device and interface descriptors.

The compatible names properties define different parts of the device that can be individually bound to client drivers. Client drivers can bind either to the entire device or to just one interface. See "Binding Client Drivers" on page 430.

The Descriptor Tree

Parsing descriptors involves aligning structure members at natural boundaries and converting the structure members to the endianness of the host CPU. Parsed standard USB configuration descriptors, interface descriptors, and endpoint descriptors are available to the client driver in the form of a hierarchical tree for each configuration. Any raw class-specific or vendor-specific descriptor information also is available to the client driver in the same hierarchical tree.

Call the `usb_get_dev_data(9F)` function to retrieve the hierarchical descriptor tree. The “SEE ALSO” section of the `usb_get_dev_data(9F)` man page lists the man pages for each standard USB descriptor. Use the `usb_parse_data(9F)` function to parse raw descriptor information.

A descriptor tree for a device with two configurations might look like the tree shown in the following figure.
The `dev_cfg` array shown in the above figure contains nodes that correspond to configurations. Each node contains the following information:

- A parsed configuration descriptor
- A pointer to an array of descriptors that correspond to the interfaces of that configuration
- A pointer to an array of class-specific or vendor-specific raw data, if any exists

The node that represents the second interface of the second indexed configuration is at `dev_cfg[1].cfg_if[1]` in the diagram. That node contains an array of nodes that represent the alternate settings for that interface. The hierarchy of USB descriptors propagates through the tree. ASCII strings from string descriptor data are attached where the USB specification says these strings exist.

The array of configurations is non-sparse and is indexed by the configuration index. The first valid configuration (configuration 1) is `dev_cfg[0]`. Interfaces and alternate settings have indices that align with their numbers. Endpoints of each alternate setting are indexed consecutively. The first endpoint of each alternate setting is at index 0.

This numbering scheme makes the tree easy to traverse. For example, the raw descriptor data of endpoint index 0, alternate 0, interface 1, configuration index 1 is at the node defined by the following path:

```
dev_cfg[1].cfg_if[1].if_alt[0].altif_ep[0].ep_descr
```
An alternative to using the descriptor tree directly is using the `usb_lookup_ep_data(9F)` function. The `usb_lookup_ep_data(9F)` function takes as arguments the interface, alternate, which endpoint, endpoint type, and direction. You can use the `usb_lookup_ep_data(9F)` function to traverse the descriptor tree to get a particular endpoint. See the `usb_get_dev_data(9F)` man page for more information.

### Registering Drivers to Gain Device Access

Two of the first calls into the USBA 2.0 framework by a client driver are calls to the `usb_clientAttach(9F)` function and the `usb_get_dev_data(9F)` function. These two calls come from the client driver's `attach(9E)` entry point. You must call the `usb_clientAttach(9F)` function before you call the `usb_get_dev_data(9F)` function.

The `usb_clientAttach(9F)` function registers a client driver with the USBA 2.0 framework. The `usb_clientAttach(9F)` function enforces versioning. All client driver source files must start with the following lines:

```c
#define USBDRV_MAJOR_VER 2
#define USBDRV_MINOR_VER minor-version
#include <sys/usb/usba.h>
```

The value of `minor-version` must be less than or equal to `USBA_MINOR_VER`. The symbol `USBA_MINOR_VER` is defined in the `<sys/usb/usbai.h>` header file. The `<sys/usb/usbai.h>` header file is included by the `<sys/usb/usba.h>` header file.

`USBDRV_VERSION` is a macro that generates the version number from `USBDRV_MAJOR_VERSION` and `USBDRV_MINOR_VERSION`. The second argument to `usb_clientAttach()` must be `USBDRV_VERSION`. The `usb_clientAttach()` function fails if the second argument is not `USBDRV_VERSION` or if `USBDRV_VERSION` reflects an invalid version. This restriction ensures programming interface compatibility.

The `usb_get_dev_data()` function returns information that is required for proper USB device management. For example, the `usb_get_dev_data()` function returns the following information:

- The default control pipe
- The `iblock_cookie` to use in mutex initializations (see `mutex_init(9F)`)
- The parsed device descriptor
- ID strings
- The tree hierarchy as described in “The Descriptor Tree” on page 435

The call to the `usb_get_dev_data()` function is mandatory. Calling `usb_get_dev_data()` is the only way to retrieve the default control pipe and retrieve the `iblock_cookie` required for mutex initialization.
After calling `usb_get_dev_data()`, the client driver’s `attach()` routine typically copies the desired descriptors and data from the descriptor tree to the driver’s soft state. Endpoint descriptors copied to the soft state are used later to open pipes to those endpoints. The `attach()` routine usually calls `usb_free_descr_tree()` to free the descriptor tree after copying descriptors. Alternatively, you might choose to keep the descriptor tree and not copy the descriptors.

Specify one of the following three parse levels to the `usb_get_dev_data()` function to request the breadth of the descriptor tree you want returned. You need greater tree breadth if your driver needs to bind to more of the device.

- **USB_PARSE_LVL_IF**. If your client driver binds to a specific interface, the driver needs the descriptors for only that interface. Specify `USB_PARSE_LVL_IF` for the parse level in the `usb_get_dev_data()` call to retrieve only those descriptors.

- **USB_PARSE_LVL_CFG**. If your client driver binds to the whole device, specify `USB_PARSE_LVL_CFG` to retrieve all descriptors of the current configuration.

- **USB_PARSE_LVL_ALL**. Specify `USB_PARSE_LVL_ALL` to retrieve all descriptors of all configurations. For example, you need this greatest tree breadth to use `usb_print_descr_tree()` to print a descriptor dump of all configurations of a device.

The client driver’s `detach()` routine must call the `usb_free_dev_data()` function to release all resources allocated by the `usb_get_dev_data()` function. The `usb_free_dev_data()` function accepts handles where the descriptor tree has already been freed with the `usb_free_descr_tree()` function. The client driver’s `detach()` routine also must call the `usb_client_detach()` function to release all resources allocated by the `usb_client_attach()` function.

---

**Device Communication**

USB devices operate by passing requests through communication channels called *pipes*. Pipes must be open before you can submit requests. Pipes also can be flushed, queried, and closed. This section discusses pipes, data transfers and callbacks, and data requests.

**USB Endpoints**

The four kinds of pipes that communicate with the four kinds of USB endpoints are:

- **Control**. Control pipes are used primarily to send commands and retrieve status. Control pipes are intended for non-periodic, host-initiated request and response communication of small-sized structured data. Control pipes are bidirectional. The default pipe is a control pipe. See “The Default Pipe” on page 439.

- **Bulk**. Bulk pipes are used primarily for data transfer. Bulk pipes offer reliable transportation of large amounts of data. Bulk pipes do not necessarily deliver the data in a timely manner. Bulk pipes are unidirectional.
Interrupt. Interrupt pipes offer timely, reliable communication of small amounts of unstructured data. Periodic polling often is started on interrupt-IN pipes. Interrupt-IN pipes return data to the host when the data becomes present on the device. Some devices have interrupt-OUT pipes. Interrupt-OUT pipes transfer data to the device with the same timely, reliable “interrupt pipe” characteristics of interrupt-IN pipes. Interrupt pipes are unidirectional.

Isochronous. Isochronous pipes offer a channel for transferring constant-rate, time-relevant data, such as for audio devices. Data is not retried on error. Isochronous pipes are unidirectional.

See Chapter 5 of the USB 2.0 specification or see “Requests” on page 442 for more information on the transfer types that correspond to these endpoints.

The Default Pipe

Each USB device has a special control endpoint called the default endpoint. Its communication channel is called the default pipe. Most, if not all, device setup is done through this pipe. Many USB devices have this pipe as their only control pipe.

The `usb_get_dev_data(9F)` function provides the default control pipe to the client driver. This pipe is pre-opened to accommodate any special setup needed before opening other pipes. This default control pipe is special in the following ways:

- This pipe is shared. Drivers that are operating other interfaces of the same device use the same default control pipe. The USBA 2.0 framework arbitrates this pipe among the different drivers.
- This pipe cannot be opened, closed, or reset by the client driver. This restriction exists because the pipe is shared.
- The pipe is autocleared on an exception.

Other pipes, including other control pipes, must be opened explicitly and are exclusive-open only.

Pipe States

Pipes are in one of the following states:

- **USB_PIPE_STATE_IDLE**
  - All control and bulk pipes, interrupt-OUT pipes, and isochronous-OUT pipes: No request is in progress.
  - Interrupt-IN and isochronous-IN pipes: No polling is in progress.
- **USB_PIPE_STATE_ACTIVE**
- All control and bulk pipes, interrupt-OUT pipes, and isochronous-OUT pipes: The pipe is transferring data or an I/O request is active.
- Interrupt-IN and isochronous-IN pipes: Polling is active.
- **USB PIPE STATE_ERROR**: An error occurred. If this pipe is not the default pipe and if autoclearing is not enabled, then the client driver must call the `usb_pipe_reset(9F)` function.
- **USB PIPE STATE CLOSING**: The pipe is being closed.
- **USB PIPE STATE CLOSED**: The pipe is closed.

Call the `usb_pipe_get_state(9F)` function to retrieve the state of a pipe.

### Opening Pipes

To open a pipe, pass to the `usb_pipe_open(9F)` function the endpoint descriptor that corresponds to the pipe you want to open. Use the `usb_get_dev_data(9F)` and `usb_lookup_ep_data(9F)` functions to retrieve the endpoint descriptor from the descriptor tree. The `usb_pipe_open(9F)` function returns a handle to the pipe.

You must specify a pipe policy when you open a pipe. The pipe policy contains an estimate of the number of concurrent asynchronous operations that require separate threads that will be needed for this pipe. An estimate of the number of threads is the number of parallel operations that could occur during a callback. The value of this estimate must be at least 2. See the `usb_pipe_open(9F)` man page for more information on pipe policy.

### Closing Pipes

The driver must use the `usb_pipe_close(9F)` function to close pipes other than the default pipe. The `usb_pipe_close(9F)` function enables all remaining requests in the pipe to complete. The function then allows one second for all callbacks of those requests to complete.

### Data Transfer

For all pipe types, the programming model is as follows:

1. Allocate a request.
2. Submit the request using one of the pipe transfer functions. See the `usb_pipe_bulk_xfer(9F), usb_pipe_ctrl_xfer(9F), usb_pipe_intr_xfer(9F),` and `usb_pipe_isoc_xfer(9F)` man pages.
3. Wait for completion notification.
4. Free the request.
See “Requests” on page 442 for more information on requests. The following sections describe the features of different request types.

**Synchronous and Asynchronous Transfers and Callbacks**

Transfers are either synchronous or asynchronous. Synchronous transfers block until they complete. Asynchronous transfers callback into the client driver when they complete. Most transfer functions called with the `USB_FLAGS_SLEEP` flag set in the `flags` argument are synchronous.

Continuous transfers such as polling and isochronous transfers cannot be synchronous. Calls to transfer functions for continuous transfers made with the `USB_FLAGS_SLEEP` flag set block only to wait for resources before the transfer begins.

Synchronous transfers are the most simple transfers to set up because synchronous transfers do not require any callback functions. Synchronous transfer functions return a transfer start status, even though synchronous transfer functions block until the transfer is completed. Upon completion, you can find additional information about the transfer status in the completion reason field and callback flags field of the request. Completion reasons and callback flags fields are discussed below.

If the `USB_FLAGS_SLEEP` flag is not specified in the `flags` argument, that transfer operation is asynchronous. The exception to this rule are isochronous transfers. Asynchronous transfer operations set up and start the transfer, and then return before the transfer is complete. Asynchronous transfer operations return a transfer start status. The client driver receives transfer completion status through callback handlers.

Callback handlers are functions that are called when asynchronous transfers complete. Do not set up an asynchronous transfer without callbacks. The two types of callback handlers are normal completion handlers and exception handlers. You can specify one handler to be called in both of these cases.

- **Normal completion.** A normal completion callback handler is called to notify of a normally completed transfer.
- **Exception.** An exception callback handler is called to notify of an abnormally completed transfer and to process its errors.

Both completion handlers and exception handlers receive the transfer’s request as an argument. Exception handlers use the completion reason and callback status in the request to find out what happened. The completion reason (`usb_cr_t`) indicates how the original transaction completed. For example, a completion reason of `USB_CR_TIMEOUT` indicates that the transfer timed out. As another example, if a USB device is removed while in use, client drivers might receive `USB_CR_DEV_NOT_RESP` as the completion reason on their outstanding requests. The callback status (`usb_cb_flags_t`) indicates what the USBA framework did to remedy the situation. For example, a callback status of `USB_CB_STALL_CLEARED` indicates that the USBA...
framework cleared a functional stall condition. See the \texttt{usb_completion_reason(9S)} man page for more information on completion reasons. See the \texttt{usb_callback_flags(9S)} man page for more information on callback status flags.

The context of the callback and the policy of the pipe on which the requests are run limit what you can do in the callback.

- **Callback context.** Most callbacks execute in kernel context and usually can block. Some callbacks execute in interrupt context and cannot block. The \texttt{USB_CB_INTR_CONTEXT} flag is set in the callback flags to denote interrupt context. See the \texttt{usb_callback_flags(9S)} man page for more information on callback context and details on blocking.

- **Pipe policy.** The pipe policy's hint on concurrent asynchronous operations limits the number of operations that can be run in parallel, including those executed from a callback handler. Blocking on a synchronous operation counts as one operation. See the \texttt{usb_pipe_open(9F)} man page for more information on pipe policy.

### Requests

This section discusses request structures and allocating and deallocating different types of requests.

#### Request Allocation and Deallocation

Requests are implemented as initialized request structures. Each different endpoint type takes a different type of request. Each type of request has a different request structure type. The following table shows the structure type for each type of request. This table also lists the functions to use to allocate and free each type of structure.

**TABLE 20–1** Request Initialization

<table>
<thead>
<tr>
<th>Pipe or Endpoint Type</th>
<th>Request Structure</th>
<th>Request Structure Allocation Function</th>
<th>Request Structure Free Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control</td>
<td>\texttt{usb_ctrl_req_t} (see the \texttt{usb_ctrl_request(9S)} man page)</td>
<td>\texttt{usb_alloc_ctrl_req(9F)}</td>
<td>\texttt{usb_free_ctrl_req(9F)}</td>
</tr>
<tr>
<td>Bulk</td>
<td>\texttt{usb_bulk_req_t} (see the \texttt{usb_bulk_request(9S)} man page)</td>
<td>\texttt{usb_alloc_bulk_req(9F)}</td>
<td>\texttt{usb_free_bulk_req(9F)}</td>
</tr>
<tr>
<td>Interrupt</td>
<td>\texttt{usb_intr_req_t} (see the \texttt{usb_intr_request(9S)} man page)</td>
<td>\texttt{usb_alloc_intr_req(9F)}</td>
<td>\texttt{usb_free_intr_req(9F)}</td>
</tr>
<tr>
<td>Isochronous</td>
<td>\texttt{usb_isoc_req_t} (see the \texttt{usb_isoc_request(9S)} man page)</td>
<td>\texttt{usb_alloc_isoc_req(9F)}</td>
<td>\texttt{usb_free_isoc_req(9F)}</td>
</tr>
</tbody>
</table>
The following table lists the transfer functions that you can use for each type of request.

**TABLE 20–2 Request Transfer Setup**

<table>
<thead>
<tr>
<th>Pipe or Endpoint Type</th>
<th>Transfer Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control</td>
<td><code>usb_pipe_ctrl_xfer(9F), usb_pipe_ctrl_xfer_wait(9F)</code></td>
</tr>
<tr>
<td>Bulk</td>
<td><code>usb_pipe_bulk_xfer(9F)</code></td>
</tr>
<tr>
<td>Interrupt</td>
<td><code>usb_pipe_intr_xfer(9F), usb_pipe_stop_intr_polling(9F)</code></td>
</tr>
<tr>
<td>Isochronous</td>
<td><code>usb_pipe_isoc_xfer(9F), usb_pipe_stop_isoc_polling(9F)</code></td>
</tr>
</tbody>
</table>

Use the following procedure to allocate and deallocate a request:

1. Use the appropriate allocation function to allocate a request structure for the type of request you need. The man pages for the request structure allocation functions are listed in Table 20–1.
2. Initialize any fields you need in the structure. See "Request Features and Fields" on page 443 or the appropriate request structure man page for more information. The man pages for the request structures are listed in Table 20–1.
3. When the data transfer is complete, use the appropriate free function to free the request structure. The man pages for the request structure free functions are listed in Table 20–1.

**Request Features and Fields**

Data for all requests is passed in message blocks so that the data is handled uniformly whether the driver is a STREAMS, character, or block driver. The message block type, `mblk_t`, is described in the `mblk(9S)` man page. The DDI offers several routines for manipulating message blocks. Examples include `allocb(9F)` and `freemsg(9F)`. To learn about other routines for manipulating message blocks, see the "SEE ALSO" sections of the `allocb(9F)` and `freemsg(9F)` man pages. Also see the STREAMS Programming Guide.

The following request fields are included in all transfer types. In each field name, the possible values for `xxxx` are: `ctrl`, `bulk`, `intr`, or `isoc`.

- `xxxx_client_private` This field value is a pointer that is intended for internal data to be passed around the client driver along with the request. This pointer is not used to transfer data to the device.
- `xxxx_attributes` This field value is a set of transfer attributes. While this field is common to all request structures, the initialization of this field is somewhat different for each transfer type. See the appropriate request structure man page for more information. These man pages are listed in Table 20–1. See also the `usb_request_attributes(9S)` man page.
The following sections describe the request fields that are different for the four different transfer types. These sections describe how to initialize these structure fields. These sections also describe the restrictions on various combinations of attributes and parameters.

**Control Requests**

Use control requests to initiate message transfers down a control pipe. You can set up transfers manually, as described below. You can also set up and send synchronous transfers using the `usb_pipe_ctrl_xfer_wait(9F)` wrapper function.

The client driver must initialize the `ctrl_bmRequestType`, `ctrl_bRequest`, `ctrl_wValue`, `ctrl_wIndex`, and `ctrl_wLength` fields as described in the USB 2.0 specification.

The `ctrl_data` field of the request must be initialized to point to a data buffer. The `usb_alloc_ctrl_req(9F)` function initializes this field when you pass a positive value as the buffer `len`. The buffer, of course, be initialized for any outbound transfers. In all cases, the client driver must free the request when the transfer is complete.

Multiple control requests can be queued. Queued requests can be a combination of synchronous and asynchronous requests.

The `ctrl_timeout` field defines the maximum wait time for the request to be processed, excluding wait time on the queue. This field applies to both synchronous and asynchronous requests. The `ctrl_timeout` field is specified in seconds.
The `ctrl_exc_cb` field accepts the address of a function to call if an exception occurs. The arguments of this exception handler are specified in the `usb_ctrl_request(9S)` man page. The second argument of the exception handler is the `usb_ctrl_req_t` structure. Passing the request structure as an argument allows the exception handler to check the `ctrl_completion_reason` and `ctrl_cb_flags` fields of the request to determine the best recovery action.

The `USB_ATTRS_ONE_XFER` and `USB_ATTRS_ISOC_*` flags are invalid attributes for all control requests. The `USB_ATTRS_SHORT_XFER_OK` flag is valid only for host-bound requests.

**Bulk Requests**

Use bulk requests to send data that is not time-critical. Bulk requests can take several USB frames to complete, depending on overall bus load.

All requests must receive an initialized message block. See the `mb1k(9S)` man page for a description of the `mb1k_t` message block type. This message block either supplies the data or stores the data, depending on the transfer direction. Refer to the `usb_bulk_request(9S)` man page for more details.

The `USB_ATTRS_ONE_XFER` and `USB_ATTRS_ISOC_*` flags are invalid attributes for all bulk requests. The `USB_ATTRS_SHORT_XFER_OK` flag is valid only for host-bound requests.

The `usb_pipe_get_max_bulk_transfer_size(9F)` function specifies the maximum number of bytes per request. The value retrieved can be the maximum value used in the client driver’s `minphys(9F)` routine.

Multiple bulk requests can be queued.

**Interrupt Requests**

Interrupt requests typically are for periodic inbound data. Interrupt requests periodically poll the device for data. However, the USBA 2.0 framework supports one-time inbound interrupt data requests, as well as outbound interrupt data requests. All interrupt requests can take advantage of the USB interrupt transfer features of timeliness and retry.

The `USB_ATTRS_ISOC_*` flags are invalid attributes for all interrupt requests. The `USB_ATTRS_SHORT_XFER_OK` and `USB_ATTRS_ONE_XFER` flags are valid only for host-bound requests.

Only one-time polls can be done as synchronous interrupt transfers. Specifying the `USB_ATTRS_ONE_XFER` attribute in the request results in a one-time poll.

Periodic polling is started as an asynchronous interrupt transfer. An original interrupt request is passed to `usb_pipe_intr_xfer(9F)`. When polling finds new data to return, a new `usb_intr_req_t` structure is cloned from the original and is populated with an initialized data
block. When allocating the request, specify zero for the `len` argument to the `usb_alloc_intr_req(9F)` function. The `len` argument is zero because the USBA 2.0 framework allocates and fills in a new request with each callback. After you allocate the request structure, fill in the `intr_len` field to specify the number of bytes you want the framework to allocate with each poll. Data beyond `intr_len` bytes is not returned.

The client driver must free each request it receives. If the message block is sent upstream, decouple the message block from the request before you send the message block upstream. To decouple the message block from the request, set the data pointer of the request to `NULL`. Setting the data pointer of the request to `NULL` prevents the message block from being freed when the request is deallocated.

Call the `usb_pipe_stop_intr_polling(9F)` function to cancel periodic polling. When polling is stopped or the pipe is closed, the original request structure is returned through an exception callback. This returned request structure has its completion reason set to `USB_CR_STOPPED_POLLING`.

Do not start polling while polling is already in progress. Do not start polling while a call to `usb_pipe_stop_intr_polling(9F)` is in progress.

**Isochronous Requests**

Isochronous requests are for streaming, constant-rate, time-relevant data. Retries are not made on errors. Isochronous requests have the following request-specific fields:

- `isoc_frame_no` Specify this field when the overall transfer must start from a specific frame number. The value of this field must be greater than the current frame number. Use `usb_get_current_frame_number(9F)` to find the current frame number. Note that the current frame number is a moving target. For low-speed and full-speed buses, the current frame is new each millisecond. For high-speed buses, the current frame is new each 0.125 millisecond. Set the `USB_ATTR_ISOC_START_FRAME` attribute so that the `isoc_frame_no` field is recognized.

  To ignore this frame number field and start as soon as possible, set the `USB_ATTR_ISOC_XFER_ASAP` flag.

- `isoc_pkts_count` This field is the number of packets in the request. This value is bounded by the value returned by the `usb_get_max_pkts_per_isoc_request(9F)` function and by the size of the `isoc_pkt_descr` array (see below). The number of bytes transferable with this request is equal to the product of this `isoc_pkts_count` value and the `wMaxPacketSize` value of the endpoint.

- `isoc_pkts_length` This field is the sum of the lengths of all packets of the request. This value is set by the initiator. This value should be set to zero so that the sum of
isoc_pkts_length in the isoc_pkt_descr list will be used automatically and no check will be applied to this element.

isoc_error_count This field is the number of packets that completed with errors. This value is set by the USBA 2.0 framework.

isoc_pkt_descr This field points to an array of packet descriptors that define how much data to transfer per packet. For an outgoing request, this value defines a private queue of sub-requests to process. For an incoming request, this value describes how the data arrived in pieces. The client driver allocates these descriptors for outgoing requests. The framework allocates and initializes these descriptors for incoming requests. Descriptors in this array contain framework-initialized fields that hold the number of bytes actually transferred and the status of the transfer. See the usb_isoc_request(9S) man page for more details.

All requests must receive an initialized message block. This message block either supplies the data or stores the data. See the mblk(9S) man page for a description of the mblk_t message block type.

The USB_ATTR_ONE_XFER flag is an illegal attribute because the system decides how to vary the amounts of data through available packets. The USB_ATTR_SHORT_XFER_OK flag is valid only on host-bound data.

The usb_pipe_isoc_xfer(9F) function makes all isochronous transfers asynchronous, regardless of whether the USB_FLAGS_SLEEP flag is set. All isochronous input requests start polling.

Call the usb_pipe_stop_isoc_polling(9F) function to cancel periodic polling. When polling is stopped or the pipe is closed, the original request structure is returned through an exception callback. This returned request structure has its completion reason set to USB_CR_STOPPED_POLLING.

Polling continues until one of the following events occurs:

- A usb_pipe_stop_isoc_polling(9F) call is received.
- A device disconnect is reported through an exception callback.
- A usb_pipe_close(9F) call is received.

**Flushing Pipes**

You might need to clean up a pipe after errors, or you might want to wait for a pipe to clear. Use one of the following methods to flush or clear pipes:
The `usb_pipe_reset(9F)` function resets the pipe and flushes all of its requests. Do this for pipes that are in an error state if autoclearing is not enabled on those pipes. Use `usb_pipe_get_state(9F)` to determine the state of a pipe.

The `usb_pipe_drain_reqs(9F)` function blocks waiting for all pending requests to complete before continuing. This function can wait indefinitely, or it can time-out after a specified period of time. The `usb_pipe_drain_reqs(9F)` function neither closes nor flushes the pipe.

**Device State Management**

Managing a USB device includes accounting for hotplugging, system power management (checkpoint and resume), and device power management. All client drivers should implement the basic state machine shown in the following figure. For more information, see `/usr/include/sys/usb/usba1.h`.

![USB Device State Machine Diagram](image)

- **Device unplugged.**
- **Original device reconnected.**
- **Device idles for time T and transitions to low power state.**
- **Remote wakeup by the device or by an application sending I/O to the device.**
- **Notification to save state via DDI_SUSPEND.**
- **Notification to restore state via DDI_RESUME with correct device.**
- **Notification to restore state via DDI_RESUME with device disconnected or a wrong device.**

This state machine and its four states can be augmented with driver-specific states. Device states 0x80 to 0xff can be defined and used only by client drivers.
Hotplugging USB Devices

USB devices support hotplugging. A USB device can be inserted or removed at any time. The client driver must handle removal and reinsertion of an open device. Use hotplug callbacks to handle open devices. Insertion and removal of closed devices is handled by the `attach(9E)` and `detach(9E)` entry points.

Hotplug Callbacks

The USB 2.0 framework supports the following event notifications:

- The client driver receives a callback when the device is hot removed.
- The client driver receives a callback when the device is returned after hot removal. This event callback can occur when the user returns the device to its original port if the driver instance of the device is not offline. If the driver instance is held open, the driver instance cannot be offline.

Client drivers must call `usb_register_hotplug_cbs(9F)` in their `attach(9E)` routine to register for event callbacks. Drivers must call `usb_unregister_hotplug_cbs(9F)` in their `detach(9E)` routine before dismantling.

Hot Insertion

The sequence of events for hot insertion of a USB device is as follows:

1. The hub driver, `hubd(7D)`, waits for a port connect status change.
2. The `hubd` driver detects a port connect.
3. The `hubd` driver enumerates the device, creates child device nodes, and attaches client drivers. Refer to “Binding Client Drivers” on page 430 for compatible names definitions.
4. The client driver manages the device. The driver is in the `ONLINE` state.

Hot Removal

The sequence of events for hot removal of a USB device is as follows:

1. The hub driver, `hubd(7D)`, waits for a port connect status change.
2. The `hubd` driver detects a port disconnect.
3. The `hubd` driver sends a disconnect event to the child client driver. If the child client driver is the `hubd` driver or the `usb_mid(7D)` multi-interface driver, then the child client driver propagates the event to its children.
4. The client driver receives the disconnect event notification in kernel thread context. Kernel thread context enables the driver’s disconnect handler to block.
5. The client driver moves to the DISCONNECTED state. Outstanding I/O transfers fail with the completion reason of device not responding. All new I/O transfers and attempts to open the device node also fail. The client driver is not required to close pipes. The driver is required to save the device and driver context that needs to be restored if the device is reconnected.

6. The hubd driver attempts to offline the OS device node and its children in bottom-up order.

The following events take place if the device node is not open when the hubd driver attempts to offline the device node:

1. The client driver’s detach(9E) entry point is called.
2. The device node is destroyed.
3. The port becomes available for a new device.
4. The hotplug sequence of events starts over. The hubd driver waits for a port connect status change.

The following events take place if the device node is open when the hubd driver attempts to offline the device node:

1. The hubd driver puts the offline request in the periodic offline retry queue.
2. The port remains unavailable for a new device.

If the device node was open when the hubd driver attempted to offline the device node and the user later closes the device node, the hubd driver periodic offlineing of that device node succeeds and the following events take place:

1. The client driver’s detach(9E) entry point is called.
2. The device node is destroyed.
3. The port becomes available for a new device.
4. The hotplug sequence of events starts over. The hubd driver waits for a port connect status change.

If the user closes all applications that use the device, the port becomes available again. If the application does not terminate or does not close the device, the port remains unavailable.

**Hot Reinsertion**

The following events take place if a previously-removed device is reinserted into the same port while the device node of the device is still open:

1. The hub driver, hubd(7D), detects a port connect.
2. The hubd driver restores the bus address and the device configuration.
3. The hubd driver cancels the offline retry request.
4. The hubd driver sends a connect event to the client driver.
5. The client driver receives the connect event.
6. The client driver determines whether the new device is the same as the device that was previously connected. The client driver makes this determination first by comparing device descriptors. The client driver might also compare serial numbers and configuration descriptor clouds.

The following events might take place if the client driver determines that the current device is not the same as the device that was previously connected:
1. The client driver might issue a warning message to the console.
2. The user might remove the device again. If the user removes the device again, the hot remove sequence of events starts over. The hubd driver detects a port disconnect. If the user does not remove the device again, the following events take place:
   a. The client driver remains in the DISCONNECTED state, failing all requests and opens.
   b. The port remains unavailable. The user must close and disconnect the device to free the port.
   c. The hotplug sequence of events starts over when the port is freed. The hubd driver waits for a port connect status change.

The following events might take place if the client driver determines that the current device is the same as the device that was previously connected:
1. The client driver might restore its state and continue normal operation. This policy is up to the client driver. Audio speakers are a good example where the client driver should continue.
2. If it is safe to continue using the reconnected device, the hotplug sequence of events starts over. The hubd driver waits for a port connect status change. The device is in service once again.

**Power Management**

This section discusses device power management and system power management.

Device power management manages individual USB devices depending on their I/O activity or idleness.

System power management uses checkpoint and resume to checkpoint the state of the system into a file and shut down the system completely. (Checkpoint is sometimes called "system suspend.") The system is resumed to its pre-suspend state when the system is powered up again.
Device Power Management

The following summary lists what your driver needs to do to power manage a USB device. A more detailed description of power management follows this summary.

1. Create power management components during `attach(9E)`. See the `usb_create_pm_components(9F)` man page.
2. Implement the `power(9E)` entry point.
3. Call `pm_busy_component(9F)` and `pm_raise_power(9F)` before accessing the device.
4. Call `pm_idle_component(9F)` when finished accessing the device.

The USBA 2.0 framework supports four power levels as specified by the USB interface power management specification. See `/usr/include/sys/usb/usbai.h` for information on mapping USB power levels to operating system power levels.

The `hubd` driver suspends the port when the device goes to the `USB_DEV_OS_PWR_OFF` state. The `hubd` driver resumes the port when the device goes to the `USB_DEV_OS_PWR_1` state and above. Note that port suspend is different from system suspend. In port suspend, only the USB port is shut off. System suspend is defined in “System Power Management” on page 455.

The client driver might choose to enable remote wakeup on the device. See the `usb_handle_remote_wakeup(9F)` man page. When the `hubd` driver sees a remote wakeup on a port, the `hubd` driver completes the wakeup operation and calls `pm_raise_power(9F)` to notify the child.

The following figure shows the relationship between the different pieces of power management.
The driver can implement one of the two power management schemes described at the bottom of Figure 20–5. The passive scheme is simpler than the active scheme because the passive scheme does not do power management during device transfers.

**Active Power Management**

This section describes the functions you need to use to implement the active power management scheme.

Do the following work in the attach(9E) entry point for your driver:

1. Call `usb_create_pm_components(9F)`.
2. Optionally call `usb_handle_remote_wakeup(9F)` with USB_REMOTE_WAKEUP_ENABLE as the second argument to enable a remote wakeup on the device.
3. Call `pm_busy_component(9F)`.
4. Call `pm_raise_power(9F)` to take power to the USB_DEV_OS_FULL_PWR level.
5. Communicate with the device to initialize the device.
6. Call `pm_idle_component(9F)`.

Do the following work in the `detach(9E)` entry point for your driver:
1. Call `pm_busy_component(9F)`.
2. Call `pm_raise_power(9F)` to take power to the USB_DEV_OS_FULL_PWR level.
3. If you called the `usb_handle_remote_wakeup(9F)` function in your `attach(9E)` entry point, call `usb_handle_remote_wakeup(9F)` here with `USB_REMOTE_WAKEUP_DISABLE` as the second argument.
4. Communicate with the device to cleanly shut down the device.
5. Call `pm_lower_power(9F)` to take power to the USB_DEV_OS_PWR_OFF level.
   This is the only time a client driver calls `pm_lower_power(9F)`.
6. Call `pm_idle_component(9F)`.

When a driver thread wants to start I/O to the device, that thread does the following tasks:
1. Call `pm_busy_component(9F)`.
2. Call `pm_raise_power(9F)` to take power to the USB_DEV_OS_FULL_PWR level.
3. Begin the I/O transfer.

The driver calls `pm_idle_component(9F)` when the driver receives notice that an I/O transfer has completed.

In the `power(9E)` entry point for your driver, check whether the power level to which you are transitioning is valid. You might also need to account for different threads calling into `power(9E)` at the same time.

The `power(9E)` routine might be called to take the device to the USB_DEV_OS_PWR_OFF state if the device has been idle for some time or the system is shutting down. This state corresponds to the `PWRED_DWN` state shown in Figure 20–4. If the device is going to the USB_DEV_OS_PWR_OFF state, do the following work in your `power(9E)` routine:
1. Put all open pipes into the idle state. For example, stop polling on the interrupt pipe.
2. Save any device or driver context that needs to be saved.
   The port to which the device is connected is suspended after the call to `power(9E)` completes.

The `power(9E)` routine might be called to power on the device when either a device-initiated remote wakeup or a system-initiated wakeup is received. Wakeup notices occur after the device has been powered down due to extended idle time or system suspend. If the device is going to the USB_DEV_OS_PWR_1 state or above, do the following work in your `power(9E)` routine:
1. Restore any needed device and driver context.
2. Restart activity on the pipe that is appropriate to the specified power level. For example, start polling on the interrupt pipe.

If the port to which the device is connected was previously suspended, that port is resumed before power(9E) is called.

**Passive Power Management**

The passive power management scheme is simpler than the active power management scheme described above. In this passive scheme, no power management is done during transfers. To implement this passive scheme, call `pm_busy_component(9F)` and `pm_raise_power(9F)` when you open the device. Then call `pm_idle_component(9F)` when you close the device.

**System Power Management**

System power management consists of turning off the entire system after saving its state, and restoring the state after the system is turned back on. This process is called CPR (checkpoint and resume). USB client drivers operate the same way that other client drivers operate with respect to CPR. To suspend a device, the driver's `detach(9E)` entry point is called with a `cmd` argument of DDI_SUSPEND. To resume a device, the driver's `attach(9E)` entry point is called with a `cmd` argument of DDI_RESUME. When you handle the DDI_SUSPEND command in your `detach(9E)` routine, clean up device state and clean up driver state as much as necessary for a clean resume later. (Note that this corresponds to the SUSPENDED state in Figure 20–4.) When you handle the DDI_RESUME command in your `attach(9E)` routine, always take the device to full power to put the system in sync with the device.

For USB devices, suspend and resume are handled similarly to a hotplug disconnect and reconnect (see “Hotplugging USB Devices” on page 449). An important difference between CPR and hotplugging is that with CPR the driver can fail the checkpoint process if the device is not in a state from which it can be suspended. For example, the device cannot be suspended if the device has an error recovery in progress. The device also cannot be suspended if the device is busy and cannot be stopped safely.

**Serialization**

In general, a driver should not call USBA functions while the driver is holding a mutex. Therefore, race conditions in a client driver can be difficult to prevent.

Do not allow normal operational code to run simultaneously with the processing of asynchronous events such as a disconnect or CPR. These types of asynchronous events normally clean up and dismantle pipes and could disrupt the normal operational code.

One way to manage race conditions and protect normal operational code is to write a serialization facility that can acquire and release an exclusive-access synchronization object.
You can write the serialization facility in such a way that the synchronization object is safe to hold through calls to USBA functions. The `usbskel` sample driver demonstrates this technique.

### Utility Functions

This section describes several functions that are of general use.

### Device Configuration Facilities

This section describes functions related to device configuration.

#### Getting Interface Numbers

If you are using a multiple-interface device where the `usb_mid(7D)` driver is making only one of its interfaces available to the calling driver, you might need to know the number of the interface to which the calling driver is bound. Use the `usb_get_if_number(9F)` function to do any of the following tasks:

- Return the number of the interface to which the calling driver is bound. The `usb_get_if_number(9F)` function returns an interface number greater than zero in this case.
- Discover that the calling driver manages an entire multi-interface device. The driver is bound at the device level so that `usb_mid` has not split it. The `usb_get_if_number(9F)` function returns `USB_DEVICE_NODE` in this case.
- Discover that the calling driver manages an entire device by managing the only interface that device offers in its current configuration. The `usb_get_if_number(9F)` function returns `USB_COMBINED_NODE` in this case.

#### Managing Entire Devices

If a driver manages an entire composite device, that driver can bind to the entire device by using a compatible name that contains vendor ID, product ID, and revision ID. A driver that is bound to an entire composite device must manage all the interfaces of that device as a nexus driver would. In general, you should not bind your driver to an entire composite device. Instead, you should use the generic multiple-interface driver `usb_mid(7D)`.

Use the `usb_owns_device(9F)` function to determine whether a driver owns an entire device. The device might be a composite device. The `usb_owns_device(9F)` function returns TRUE if the driver owns the entire device.
Multiple-Configuration Devices

USB devices make only a single configuration available to the host at any particular time. Most devices support only a single configuration. However, a few USB devices support multiple configurations.

Any device that has multiple configurations is placed into the first configuration for which a driver is available. When seeking a match, device configurations are considered in numeric order. If no matching driver is found, the device is set to the first configuration. In this case, the usb_mid driver takes over the device and splits the device into interface nodes. Use the `usb_get_cfg(9F)` function to return the current configuration of a device.

You can use either of the following two methods to request a different configuration. Using either of these two methods to modify the device configuration ensures that the USBA module remains in sync with the device.

- Use the `cfgadm_usb(1M)` command.
- Call the `usb_set_cfg(9F)` function from the driver.

Because changing device configuration affects an entire device, the client driver must meet all of the following criteria to call the `usb_set_cfg(9F)` function successfully:

- The client driver must own the entire device.
- The device must have no child nodes, because other drivers could drive the device through them.
- All pipes except the default pipe must be closed.
- The device must have multiple configurations.

**Caution** – Do not change the device configuration by doing a SET_CONFIGURATION USB request manually. Using a SET_CONFIGURATION request to change the configuration is not supported.

Modifying or Getting the Alternate Setting

A client driver can call the `usb_set_alt_if(9F)` function to change the selected alternate setting of the currently selected interface. Be sure to close all pipes that were opened explicitly. When switching alternate settings, the `usb_set_alt_if(9F)` function verifies that only the default pipe is open. Be sure the device is settled before you call `usb_set_alt_if(9F).

Changing the alternate setting can affect which endpoints and which class-specific and vendor-specific descriptors are available to the driver. See “The Descriptor Tree” on page 435 for more information about endpoints and descriptors.

Call the `usb_get_alt_if(9F)` function to retrieve the number of the current alternate setting.
Note – When you request a new alternate setting, a new configuration, or a new interface, all pipes except the default pipe to the device must be closed. This is because changing an alternate setting, a configuration, or an interface changes the mode of operation of the device. Also, changing an alternate setting, a configuration, or an interface changes the device’s presentation to the system.

Other Utility Functions

This section describes other functions that are useful in USB device drivers.

Retrieving a String Descriptor

Call the `usb_get_string_descr(9F)` function to retrieve a string descriptor given its index. Some configuration, interface, or device descriptors have string IDs associated with them. Such descriptors contain string index fields with nonzero values. Pass a string index field value to the `usb_get_string_descr(9F)` to retrieve the corresponding string.

Pipe Private Data Facility

Each pipe has one pointer of space set aside for the client driver’s private use. Use the `usb_pipe_set_private(9F)` function to install a value. Use the `usb_pipe_get_private(9F)` function to retrieve the value. This facility is useful in callbacks, when pipes might need to bring their own client-defined state to the callback for specific processing.

Clearing a USB Condition

Use the `usb_clr_feature(9F)` function to do the following tasks:

- Issue a USB CLEAR_FEATURE request to clear a halt condition on an endpoint.
- Clear a remote wakeup condition on a device.
- Clear a device-specific condition at a device, interface, or endpoint level.

Getting Device, Interface, or Endpoint Status

Use the `usb_get_status(9F)` function to issue a USB GET_STATUS request to retrieve the status of a device, interface, or endpoint.

- **Device status.** Self-powered and remote-wakeup-enabled.
- **Interface status.** Returns zero, per USB 2.0 specification.
- **Endpoint status.** Endpoint halted. This status indicates a functional stall. A halt must be cleared before the device can operate again.
A protocol stall indicates that an unsupported control pipe request has been made. A protocol stall is cleared automatically at the beginning of the next control transfer.

**Getting the Bus Address of a Device**

Use the `usb_get_addr(9F)` function to get the USB bus address of a device for debugging purposes. This address maps to a particular USB port.

**Sample USB Device Drivers**

This section describes a template USB device driver that uses the USBA 2.0 framework for the Oracle Solaris environment. This driver demonstrates many of the features discussed in this chapter. This template or skeleton driver is named `usbskel`.

The `usbskel` driver is a template that you can use to start your own USB device driver. The `usbskel` driver demonstrates the following features:

- Reading the raw configuration data of a device. Every USB device needs to be able to report device raw configuration data.
- Managing pipes. The `usbskel` driver opens an interrupt pipe to show how to manage pipes.
- Polling. Comments in the `usbskel` driver discuss how to do polling.
- USB version management and registration.
- USB logging.
- Accommodations for USB hotplugging.
- Accommodations for Oracle Solaris suspend and resume.
- Accommodations for power management.
- USB serialization.
- Use of USB callbacks.
Introduction to SR-IOV

The SR-IOV technology is a hardware based virtualization solution that improves both performance and scalability. The SR-IOV standard enables efficient sharing of PCIe (Peripheral Component Interconnect) Express devices among virtual machines and is implemented in the hardware to achieve I/O performance which is comparable to native performance. The SR-IOV specification defines a new standard wherein the new devices that are created will enable the virtual machine to be directly connected to the I/O device.

The SR-IOV specification is defined and maintained by PCI-SIG at http://www.pcisig.com

A single I/O resource can be shared by many virtual machines. The shared devices will provide dedicated resources and also utilize shared common resources. In this way, each virtual machine will have access to unique resources. Therefore, a PCIe device, such as an Ethernet Port, that is SR-IOV enabled with appropriate hardware and OS support can appear as multiple, separate physical devices, each with its own configuration space.

The following figure illustrates the SR-IOV technology for PCIe hardware.
Two new function types in SR-IOV are:

**Physical Function (PF)** A PCI Function that supports the SR-IOV capabilities as defined in SR-IOV specification. A PF contains the SR-IOV capability structure and is used to manage the SR-IOV functionality. PFs are fully featured PCIe functions that can be discovered, managed, and manipulated like any other PCIe device. PFs have full configuration resources, and can be used to configure or control the PCIe device.

**Virtual Function (VF)** A function that is associated with a Physical Function. A VF is a lightweight PCIe function that shares one or more physical resources with the Physical Function and with other VFs that are associated with the same PF. VFs are only allowed to have configuration resources for its own behavior.

Each SR-IOV device can have a Physical Function (PF) and each PF can have up to 64K Virtual Functions (VFs) associated with the PF. The VFs can be created by the PF through registers designed with properties specifically for this purpose.
Once SR-IOV is enabled in the PF, the PCI configuration space of each VF can be accessed by the bus, device, and function number (Routing ID) of the PF. Each VF also has a PCI Memory Space, which is used to map its register set. The device drivers of the VFs operate on the register set to enable its functionality and appear as a real existing PCI device. Once VFs are created, they can be directly assigned to IO guest domains or individual applications such as Oracle Solaris Zones on bare-metal platforms. This capability enables them to share the physical device and perform I/O without CPU and Hypervisor software overhead.

Benefits of SR-IOV

The SR-IOV standard allows efficient sharing of PCIe devices among IO Guest Domains. An SR-IOV device can have hundreds of Virtual Functions (VFs) associating with a Physical Function (PF). The creation of VFs can be dynamically controlled by the PF through registers designed to turn on the SR-IOV capability. By default, the SR-IOV capability is turned off, and the PF behaves as traditional PCIe device.

Devices that have SR-IOV capability can take advantage of the following benefits:

- Performance – Direct access to hardware from virtual machines environment.
- Cost Reduction – Capital and operational expenditure savings include:
  - Power savings
  - Reduced adapter count
  - Less cabling
  - Fewer switch ports

Supported Platforms

- Sparc: All T3 and T4 series of systems support Oracle Solaris SR-IOV capable devices.
- x86: Intel Nehalem-EX based systems such as x4470 and x4800 support Oracle Solaris SR-IOV capable devices.

The following cards support SR-IOV capable devices:

- Intel based NICs such as Kawela (82576, 1G)
- Niantic (82599, 10G)

Note – To be able to run SR-IOV capable devices, the system firmware should support SR-IOV capability.
Overview of SR-IOV Device Driver

The SR-IOV functionality is comprised of the Physical Function (PF) driver and the Virtual Function (VF) driver. The following sections describe the PF and VF drivers and the details of the required device configuration.

Physical Function (PF) Driver

The PF driver of a SR-IOV device is used to manage the Physical Function (PF) of an SR-IOV capable device. A PCI Function that supports the SR-IOV capabilities is defined in the SR-IOV specification. A PF contains the SR-IOV capability structure and is used to manage the SR-IOV functionality. PFS are fully featured PCIe functions which can be discovered, managed, and manipulated like any other PCIe device. PFS have full configuration resources, and can be used to configure or control the PCIe device. A PF driver exhibits the following characteristics:

- Visible in Root Domain only
- May or may not possess data movement capabilities. The PF driver should function even in SR-IOV mode.
- Controls the enabling and disabling of the SR-IOV capability through APIs provided by the Oracle Solaris IOV Framework.
The number of VFs to be configured for a given PF is determined by the system administrator. This number defined either in Machine Descriptors (MD) on the Sparc OVM platforms or a configuration file in the bare metal environment.

The PF driver enables the VFs during the attachment phase by calling the Oracle Solaris IOV framework through the DDI interface. If the PF driver does not enable VFs during attachment, the Oracle Solaris IOV framework will attempt to configure VFs after the attachment as long as the driver callback flags indicates that the driver has IOV capability is supported.

PFs can enable and disable each associated VF individually through a device specific mechanism.

**Virtual Function (VF) Driver**

A function that is associated with a Physical Function. A VF is a lightweight PCIe function that shares one or more physical resources with the Physical Function and with other VFs that are associated with the same PF. A VF driver exhibits the following characteristics:

- Visible in both the root domain and I/O domain
- Can initiate communication with its PF either through the HW mailbox or an OS–provided interface
- Not visible in the root domain until the following conditions are satisfied:
  - The root domain has booted up
  - The PF driver attaches and invokes the configuration of the VF
  - The VFs are enabled by the root domain’s Oracle Solaris IOV framework
  - The system firmware allocates resources to the VFs
- Not visible in the I/O Domain until the following conditions are satisfied:
  - VFs are already enabled and are visible in the root domain
  - VFs are assigned to the I/O Domain
  - VFs are probed in the I/O Domain by the Oracle Solaris firmware (OBP)

*Note* – The SR-IOV Capable PF and VF drivers have to register Interrupt Resource Management (IRM) callbacks and provide support for this feature. See Chapter 8, “Interrupt Handlers,” for details and usage of IRM interfaces.
**Note** – If the VF is a network VF, the following parameters can be configured after the numVFs are enabled. The configuration should be completed before the VFs are enabled.

- **mac-addr**
- **vlan (Virtual LAN) ID**
- **port-vlan-id**
- **alt-mac-addrs**
- **mtu**

### Device Configuration Parameters

The PF driver must support the configuration parameters listed in the following table. These parameters may be exported to the Sparc OVM Manager. Configuration is complete only when all the parameters are configured.

<table>
<thead>
<tr>
<th>Configuration Parameters</th>
<th>Definition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standards related configuration parameters</td>
<td>Number of VFs it can support. <strong>Note</strong> – Changes to the number of VFs require the PF device to detach and attach again.</td>
<td><strong>max-config-vfs</strong> – Maximum number of VFs that can actually be configured. A PF driver can export this parameter when the maximum number of VFs that a PF driver supports is different from the capability indicated by the SR-IOV functionality.</td>
</tr>
<tr>
<td>Resource and device-specific parameters</td>
<td>Bandwidth, pools, and Q-pairs. Changes to these parameters can affect both PF and VF drivers. The framework may not be aware of device-specific parameters and they may be known only to the PF driver. These parameters should be known before enabling the VFs so that a PF driver initializes its hardware appropriately.</td>
<td><strong>pvid-exclusive</strong> – Indicates that a port-vlan-id and vlan-ids cannot be supported at the same time. <strong>max-vf-mtu</strong> – Maximum MTU allowed for a VF. <strong>max-vlans</strong> – Maximum number of vlan slots supported by the network class PF driver.</td>
</tr>
</tbody>
</table>

See `igb(7D)` and `ixgbe(7D)` to learn how to obtain the device specific parameters that are exported to the IOV framework.
### Table 21-1: Configuration Parameters Definition (Continued)

<table>
<thead>
<tr>
<th>Configuration Parameters</th>
<th>Definition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class-specific parameters</td>
<td>Common properties based on the class of the device. For example, Ethernet devices may have properties such as MAC addresses, VLAN-Ids, Port-VLAN-ID, bandwidth and so on. Class specific configuration is expected to be used and may define the behavior of each of the parameters.</td>
<td>None</td>
</tr>
</tbody>
</table>

**Note** – When device configuration parameters are changed, the devices should be reattached.

**Note** – Configure the parameters in the following order before enabling the VFs. Class specific parameters will be based on the class specific configurator.

1. Standards-related parameters
2. Resource and device-specific parameters
3. Class-specific parameters

**pci.conf File**

The PCI configuration information file, `/etc/pci.conf` enables the system to save PCI configurations such as the number of VFs of a particular PF. The `pci.conf` file provides the following:

- To persist the PCI configuration so that VFs can be created automatically upon booting of the system.
- Since the configuration file is part of the `boot_archive`, VFs can be used during the booting of the system.

**Note** – Add the `/etc/pci.conf` to the `/boot/solaris/filelist.ramdisk` file in order to include the `/etc/pci.conf` file into the Oracle Solaris boot process.

See Appendix E, “pci.conf File,” for more information.
### Setting Device Configuration Parameters

- **Sparc**: The parameters can be set through the `ldm` command. See the `ldm(1M)` man page for details.
- **x86**: The class-specific parameters can be specified by the `pci.conf` file. The following example shows the parameters set in a `pci.conf` file.

**EXAMPLE 21-1  Setting Device Configuration Parameters**

```plaintext
[[path=/pci@0,0/pci8086,3a40@1c/pci108e,4848@0,1]]
num-vf=2

[Device_Configuration]
[[path=/pci@0,0/pci8086,3a40@1c/pci108e,4848@0,1]]
VF[0] = {
    primary-mac-addr = 0xaabbccddeeff
    alt-mac-addrs = 0x102233445556, 0x102233445557
    vlan-id = 20, 30
}

VF[1] = {
    primary-mac-addr = 0xaabbccddeef1
    alt-mac-addrs = 0x102233445558
    vlan-id = 20, 30, 40, 50
}
```

### SR-IOV Configuration on Sparc OVM Platform

The Sparc OVM Manager is responsible for the SR-IOV configuration on all Sparc OVM platforms. The Sparc OVM Manager is responsible for the following operations:

- Obtain a list of PFs that have SR-IOV capable drivers
- Obtain the device specific parameters supported by the drivers
- Validate a specific device configuration
- Update the Machine Descriptor (MD) file with all valid configuration details along with the assignment and removal of VFs

The following figure shows a high-level view of the Sparc OVM configuration.
SR-IOV Configuration on Bare Metal Platforms

At the time of Oracle Solaris 11 release, no configuration tool is available to configure SR-IOV on bare metal platforms, including x86.

Boot Configuration Sequence

The SR-IOV capable PF driver performs the following actions during attachment:

1. Calls the `pciv_vf_config()` function to obtain the number of VFs.
2. Obtains the device specific parameters for both the PF and VF and validates them.
3. Initializes the hardware accordingly.
4. Calls the `pciv_vf_config()` interface to enable the VF.
5. If the PF driver is a network driver, the driver will register with the GLDv3 framework using the `mac_register()` interface during attachment. The PF driver also performs class-specific initialization. This results in the following set of actions:
   - The GLDv3 interface becomes aware of the existence of the PF device.
A new set of MAC provider interfaces are exported by the PF driver. This process enables the MAC layer to become aware that the driver is a PF driver. The MAC layer also obtains more information about the VF driver.

See Chapter 19, "Drivers for Network Devices," and the Oracle Solaris Administration: Network Interfaces and Network Virtualization chapter in for more information about network drivers and interfaces.

The VF instances are now initialized. A VF driver is attached only if the VF is assigned to the root domain.

**SR-IOV Interfaces Summary**

The following table lists the Oracle Solaris interfaces that are available to the SR-IOV drivers. The interfaces are called to obtain PF and VF parameters, to configure VFs, and to export the configuration parameters to any calling applications.

**Note** – All the interfaces listed in the table with the exception of `pciv_send()` are applicable only for PF drivers.

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pci_param_get(9F)</code></td>
<td>Used by PCI device drivers to get the name-value pair list of the currently configured parameters</td>
</tr>
<tr>
<td><code>pci_param_free(9F)</code></td>
<td>Must be called by the device driver after they have obtained the device parameters by using the <code>param</code> handle. This call frees up the resources allocated by the <code>pci_param_get()</code> and <code>pci_param_get_ioctl()</code> interfaces.</td>
</tr>
<tr>
<td><code>pci_plist_get(9F)</code></td>
<td>Used by the SR-IOV device drivers to get the name-value pair list for the PF devices.</td>
</tr>
<tr>
<td><code>pci_plist_getvf(9F)</code></td>
<td>Used by the SR-IOV device drivers to get the name-value pair list for the VF devices.</td>
</tr>
<tr>
<td><code>pci_param_get_ioctl(9F)</code></td>
<td>A helper function to SR-IOV device drivers to extract the parameters for the PF and VF devices from the <code>arg</code> argument if the drivers implement the ioctl IOV_VALIDATE_PARAM.</td>
</tr>
<tr>
<td><code>pciv_vf_config(9F)</code></td>
<td>Used by PF drivers to obtain VF configuration parameters and to configure their VFs.</td>
</tr>
<tr>
<td><code>pci_plist_lookup(9F)</code></td>
<td>Used to obtain the parameters for PF and VF from the list obtained from the <code>pci_plist_get()</code> and <code>pci_plist_getvf()</code> interfaces.</td>
</tr>
<tr>
<td><code>pciv_send(9F)</code></td>
<td>Used by the PF and VF drivers to transmit information.</td>
</tr>
<tr>
<td><code>ddi_cb_register(9F)</code></td>
<td>Callback registration mechanism interfaces.</td>
</tr>
</tbody>
</table>
Driver ioctl

The SR-IOV driver ioctl is used to identify the device–specific parameters that can be configured by the administrator and to validate a specific configuration before it is applied. The following ioctl and their data structures are defined.

- "iov_param_ver_info Structure" on page 479
- "iov_param_validate Structure" on page 480
- "iov_param_desc Structure" on page 480
- "IOV_GET_VER_INFO ioctl" on page 481
- "IOV_GET_PARAM_INFO ioctl" on page 481
- "IOV_VALIDATE_PARAM ioctl" on page 481

Interfaces for SR-IOV Drivers

The following sections describe the interfaces for SR-IOV drivers.

pci_param_get() Interface

SR-IOV drivers must use the `pci_param_get(9F)` interface to obtain the list of currently configured parameters. This interface is called during driver attachment or at any other appropriate time. The returned data, which is a pointer to the parameter list, contains the name–value information of both the PF and its corresponding VF devices.

```c
int pci_param_get(dev_info_t *dip, pci_param_t *php)
```

where:

dip A pointer to the `dev_info` structure
php A pointer to `param_handle`, `pci_param_t`

The device driver should perform the following steps to obtain the list of parameters of PF and VF after calling the `pci_param_get` interface:

1. Call the `pci_plist_get(9F)` interface to obtain the list of parameters of the PF device and the `pci_plist_getvf(9F)` interface to obtain the list of parameters of the configured VFs.
2. Call the `pci_plist_lookup(9F)` interface to obtain the device parameters.
3. Validate all the PF and VF parameters.
4. If the parameters do not match the current configuration, the driver should fail the device attachment.
5. Call the `pci_param_free(9F)` interface to free the pointer to the parameter handle obtaining the parameters for PF and the configured VF devices. See Example 21–2
**Note** – Validation of the parameters should be completed before the VFs are configured.

The name-value pairs are defined on a per-device basis. There is one set of name-value pairs for the PF and one set each for the configured VFs. The name-value pairs are optional and may be absent for any or all of the devices.

**EXAMPLE 21-2**  
SR-IOV `pci_param_get(9F)` Routine

```c
pci_param_t my_params;
pci_plist_t pf_plist;
pci_plist_t vf_plist[8];
labelp = NULL;
rval = pci_param_get(dip, &my_params);
if (rval || (my_params == NULL)) {
    cmn_err(CE_NOTE, "No params available\n");
    goto continue_with_attach;
}
rval = pci_plist_get(my_params, &pf_plist);
if (rval || (pf_plist == NULL)) {
    cmn_err(CE_NOTE, "No params for PF \n");
    goto continue_with_attach;
}
for (i = 0; i < 8; i++) {
    rval = pci_plist_getv(my_params, i, &vf_plist[i]);
    if (rval || (vf_plist[i] == NULL)) {
        cmn_err(CE_WARN, "No params for VF %d\n", i);
        continue;
    }
}
pci_param_free(my_params);
/*
 * Validate the PF and VF params lists.
 * Fail the attach if the params are incompatible or exceed the
 * resources available.
 */
continue_with_attach:
```

**pci_param_get_ioctl() Interface**

SR-IOV device drivers can use the `pci_param_get_ioctl(9F)` interface to extract the parameters for the PF and VF devices from the `arg` argument if they implement the IOV_VALIDATE_PARAM ioctl.

```c
int pci_param_get_ioctl(dev_info_t *dip, intptr_t arg, int mode, pci_param_t *php)
```

where:

- `dip`  A pointer to the `dev_info` structure
Argument obtained from the driver's ioctl call

mode

A pointer to the param handle, pci_param_t, which is a handle obtained from calling the pci_param_get() or pci_param_get_ioctl() interfaces

The driver should call the pci_param_free() interface to free the param handle returned in this call after retrieving the parameters.

pci_plist_get() Interface

The pci_plist_get(9F) interface is used to obtain the parameter list from the param handle obtained from either the pci_param_get(9F) call or the pci_param_get_ioctl(9F) call.

int pci_plist_get(pci_param_t param, pci_plist_t *plist_p)

where:

param A handle obtained from the pci_param_get() or pci_param_get_ioctl() interfaces.

plist_p A pointer to pci_plist_t where a non-NULL plist is returned on a successful return.

The plist that is returned from the pci_plist_get() call is only for the PF function. The structure pci_plist_t supports arrays of the following data types:

- int8_t
- uint8
- int16_t
- uint16_t
- int32_t
- uint32_t
- int64_t
- uint64_t
- char *

pci_plist_getvf() Interface

The pci_plist_getvf(9F) interface is used to obtain the name-value pair list for VF devices.

int pciv_plist_getvf (pci_param_t param, uint16_t vf_index, pci_plist_t *vfplist_p)
where:

**param**  A handle obtained from `pci_param_get()` or `pci_param_get_ioctl()` interfaces.

**vf_index**  A value between 0 through \#VFS - 1.

**vfplist_p**  A pointer to the `pci_plist_t` structure.

### `pciv_vf_config()` Interface

The `pciv_vf_config(9F)` interface is used by the SR-IOV drivers to obtain configuration info about the VFs and is also used to configure the VFs during the attachment of the driver.

```c
#include <sys/sunddi.h>
int pciv_vf_config(dev_info_t *dip, pciv_config_vf_t *vfcfg_p)
```

where:

**dip**  A pointer to the `dev_info` structure.

**vfcfg_p**  A pointer to the `pciv_config_vf` structure.

```c
typedef enum {
    PCIV_VFCFG_PARAM,
    PCIV_VF_ENABLE,
    PCIV_VF_DISABLE
} pciv_vf_config_cmd_t;
```

The `pciv_config_vf` structure contains the following fields:

```c
typedef struct pciv_config_vf {
    int version;
    pciv_vf_config_cmd_t cmd;
    uint16_t num_vf;
    uint16_t first_vf_offset;
    uint16_t vf_stride;
    boolean_t ari_cap;
    uint32_t page_size;
} pciv_config_vf_t;
```

where:

**version**  Version number.

**cmd**  Used to indicate whether this interface is called to obtain configuration information or to attach the VFs.

- **PCIV_VFCFG_PARAM** – Obtain configuration information
PCIV_VF_ENABLE – Enable the VFs
  PCIV_EVT_VFENABLE_PRE
  PCIV_EVT_VFDISABLE_PRE
  PCIV_EVT_VFENABLE_POST
  PCIV_EVT_VFDISABLE_POST

- num_vf – Number of VFs defined in the backend.
- vf_stride – Distance between the VFs.
- first_vf_offset – Offset between the first VF and PF.
- ari_cap – ARI-capable hierarchy.
- page_size – Specifies system page size.

The driver should first call the `pciv_vfconfig()` interface with the `cmd` field set to `PCIV_VFCONFIG_PARAM` to obtain the configuration information. The driver should then call this interface again with the `cmd` field set to `PCIV_VF_ENABLE` to configure the VFs.

The driver can return one of the following error codes:

- DDI_SUCCESS
- DDI_FAILURE
- PCIV_REQRESET
- PCIV_REQREATTACH

The driver does not have to register the callbacks with the `DDI_CB_FLAG_SRIOV` bit set before calling the `pciv_vf_config()` interface to enable the VFs. However, in order to receive notifications when the VFs are unconfigured by the SR-IOV framework, the drivers must register callbacks with `DDI_CB_FLAG_SRIOV` bit set after the VFs are enabled. See "Driver Callbacks" on page 482 for additional information.

All PF drivers that can support VFs should inform the PCIe framework of their capability by calling the `ddi_cb_register(9F)` with `DDI_CB_FLAG_SRIOV` flag set in the `flags` argument. The `ddi_cb_register()` function must be called in the driver’s attach routine. If the PF device driver calls the `pciv_vf_config()` function to enable VFs in its attach routine, then the PF driver should call the `ddi_cb_register()` function after enabling the VFs.

The `DDI_CB_FLAG_SRIOV` flag is required by the framework to perform the following actions:

- Indicate to the Sparc OVM agent that there is a device driver capable of supporting VFs. The Sparc OVM agent then will allow creation of VF devices. In the absence of this capability, the user will not be able to create VFs on Sparc platforms.
The framework will callback the PF driver before and after it disables the VFs. This will facilitate the PF driver to do its internal book keeping for VF support.

**pci_plist_lookup() Interface**

The `pci_plist_lookup(9F)` interface can be used by the drivers to look up name-value pairs of the various data types that are supported. The functions find the `nvpair` (name-value pair) that matches the name and type as indicated by the interface name. If found, `nelem` and `val` are modified to contain the number of elements in value and the starting address of data, respectively.

The following data types are supported by the `pci_plist_lookup()` interface:

- `int pci_plist_lookup_int8(pci_plist_t plist, const char *name, int8_t *val)`
- `int pci_plist_lookup_uint8(pci_plist_t plist, const char *name, uint8_t *val)`
- `int pci_plist_lookup_int16(pci_plist_t plist, const char *name, int16_t *val)`
- `int pci_plist_lookup_uint16(pci_plist_t plist, const char *name, uint16_t *val)`
- `int pci_plist_lookup_int32(pci_plist_t plist, const char *name, int32_t *val)`
- `int pci_plist_lookup_uint32(pci_plist_t plist, const char *name, uint32_t *val)`
- `int pci_plist_lookup_int64(pci_plist_t plist, const char *name, int64_t *val)`
- `int pci_plist_lookup_uint64(pci_plist_t plist, const char *name, uint64_t *val)`
- `int pci_plist_lookup_string(pci_plist_t plist, const char *name, char **val)`
- `int pci_plist_lookup_plist(pci_plist_t plist, const char *name, pci_plist_t **val)`
- `int pci_plist_lookup_int8_array(pci_plist_t plist, const char *name, int8_t *val, uint_t *nelem)`
- `int pci_plist_lookup_uint8_array(pci_plist_t plist, const char *name, int8_t *val, uint_t *nelem)`
- `int pci_plist_lookup_int16_array(pci_plist_t plist, const char *name, int16_t *val, uint_t *nelem)`
- `int pci_plist_lookup_uint16_array(pci_plist_t plist, const char *name, uint16_t *val, uint_t *nelem)`
- `int pci_plist_lookup_int32_array(pci_plist_t plist, const char *name, int32_t *val, uint_t *nelem)`
 vista pci_plist_lookup_uint32_array(pci_plist_t *plist, const char *name, uint32_t *val, uint_t *nelem)
- int pci_plist_lookup_int64_array(pci_plist_t *plist, const char *name, int64_t *val, uint_t *nelem)
- int pci_plist_lookup_uint64_array(pci_plist_t *plist, const char *name, uint64_t *val, uint_t *nelem)
- int pci_plist_lookup_string_array(pci_plist_t *plist, const char *name, char **val, uint_t *nelem)

where:
- plist   A pointer to the pci_plist_t structure to be processed.
- name    Name of the name-value pair to search.
- nelem   Address to store the number of elements in value.
- val     Starting address of data.

The pci_plist_lookup() function return 0 on success and an error value on failure. The following error values are supported:
- DDI_EINVAL    Invalid argument
- ENOENT        No matching name-value pair found
- ENOTSUP       An encode or decode method is not supported

**pci_param_free() Interface**

The pci_param_free(9F) interface must be called by the driver after obtaining the device parameters using the param handle. This call frees up the resources allocated by the pci_param_get() and pci_param_get_ioctl() interfaces.

```
int pci_param_free (pci_param_t param)
```

where `param` is a handle obtained from `pci_param_get()` or `pci_param_get_ioctl()` interfaces.

**pciv_send() Interface**

The pciv_send(9F) interface is used by SR-IOV capable PF and VF drivers to communicate with each other. A PF driver can communicate with any of its VF drivers although a VF driver can only communicate with its PF driver.
int pciv_send(dev_info_t *dip, pciv_pvp_req_t *req

where:

dip A pointer to the dev_info structure.

req A pointer to the pciv_pvp_req_t structure.

The structure of the pciv_pvp_req_t is:

typedef struct pciv_pvp_req {
  int pvp_dstfunc;
  caddr_t pvp_buf;
  size_t pvp_nbyte;
  buf_cb_t pvp_cb;
  caddr_t pvp_cb_arg;
  uint_t pvp_flag;
} pciv_pvp_req_t;

where:

pvp_dstfunc VF index ranges from 1 to num_vf if called by PF driver. If the caller is a VF driver it should always be PCIV_PF.

pvp_buf Buffer address of caller’s buffer to be sent.

pvp_nbyte Number of bytes to be transmitted, which must be less than 8k.

pvp_cb Call back function pointer if the pvp_flag is set as PCIV_NOWAIT.

If pvp_flag is set to PCIV_NOWAIT the call returns immediately and the callback routine in pvp_cb is called before data in pvp_buf has been transmitted to the destination. The caller is then allowed to free the buffer in its callback routine.

typedef void (*buf_cb_t)(int rc, caddr_t buf, size_t size, caddr_t cb_arg);

where:

rc DDI return code for the transmission.

buf Buffer address of caller’s buffer to be sent.

size Number of bytes to be transmitted.

cb_arg Input argument the caller has set when calling the routine.

pvp_cb_arg Call back input argument for pvp_cb if the pvp_flag is set as PCIV_NOWAIT.

pvp_flag

- PCIV_NOWAIT – Do not wait for receiver’s response.
- PCIV_WAIT – This is the default state. Wait until receiver acknowledges the transmission.
The `pciv_send()` interface returns one of the following return values:

- **DDI_SUCCESS**: Buffer has been sent successfully.
- **DDI_ENOTSUP**: Device driver does not support this operation. Caller may use other mechanisms, such as hardware mailbox.
- **DDI_EINVAL**: The `pvp_nbyte` or `pvp_dstfunc` is invalid.
- **DDI_ENOMEM**: Operation failed due to lack of resources.
- **DDI_ETRANSPORT**: The remote end did not register a call back to handle incoming transmission.
- **DDI_FAILURE**: Failed due to unspecified reasons.

### SR-IOV Driver Ioctls

The SR-IOV driver ioctls are used to identify the device specific parameters that can be configured by the administrator and to validate a specific configuration before it is applied. The following sections describe the ioctls data structures and the interfaces.

### Data Structures

The following sections list the data structures that the PF driver should define and initialize before implementing the ioctls:

#### `iov_param_ver_info` Structure

The `iov_param_ver_info` structure is defined as follows:

```c
typedef struct iov_param_ver_info {
    uint32_t version;
    uint32_t num_params;
} iov_param_ver_info_t;
```

where:

- **version**: Version information
num_params Number of parameters

**iov_param_validate Structure**

The iov_param_validate structure is defined as follows:

```c
#define IOV_IOCTL (('I' << 24) | ('O' << 16) | ('V' << 8))
#define IOV_GET_VER_INFO (IOV_IOCTL | 0)
#define IOV_GET_PARAM_INFO (IOV_IOCTL | 1)
#define IOV_VALIDATE_PARAM (IOV_IOCTL | 2)
#define IOV_PARAM_DESC_VERSION 1
```

The iov_param_validate contains the following fields:

```c
typedef struct iov_param_validate {
    char pv_reason[MAX_REASON_LEN + 1];
    int32_t pv_buflen;
    char pv_buf[1]; /* size of buf is pv_buflen */
} iov_param_validate_t;
```

where:

- **pv_reason** An ASCII string that explains the reason for failure if the ioctl call fails.
- **pv_buflen** Length of buffer `pv_buf`
- **pv_buf** Buffer containing the parameters

**iov_param_desc Structure**

The iov_param_desc structure is defined as follows:

```c
#define IOV_IOCTL (('I' << 24) | ('O' << 16) | ('V' << 8))
#define IOV_GET_VER_INFO (IOV_IOCTL | 0)
#define IOV_GET_PARAM_INFO (IOV_IOCTL | 1)
#define IOV_VALIDATE_PARAM (IOV_IOCTL | 2)
#define IOV_PARAM_DESC_VERSION 1
```

The structure iov_param_desc contains the following fields:

```c
typedef struct iov_param_desc {
    char pd_name[MAX_PARAM_NAME_SIZE];
    char pd_desc[MAX_PARAM_DESC_SIZE];
    int32_t pd_flag; /* applicable for PF or VF or both */
    int32_t pd_data_type; /* integer, string, plist */
    /* Following 3 are applicable for integer data types */
    uint64_t pd_default_value;
    uint64_t pd_min64;
    uint64_t pd_max64;
    char pd_default_string [MAX_PARAM_DEFAULT_STRING_SIZE];
} iov_param_desc_t;
```
where:

- **pd_name**: Used in the `ldm(1M)` command or the `pci.conf` file to assign a value to the parameter.
- **pd_desc**: A brief description of the parameter.
- **pd_flag**: Indicates if the parameter is applicable to only PF, only VF or applicable to both PF and VF.
- **pd_default_value**: Value assigned by the driver if parameter is not specified in the `ldm()` command or `pci.conf` file.
- **pd_min64**: Specifies the minimum range of values for the integer parameter.
- **pd_max64**: Specifies the maximum range of values for the integer parameter.
- **pd_default_string**: Specifies the default string that will be used, if parameter is a string.

### IOV_GET_VER_INFO ioctl

SR-IOV device drivers that implement the `IOV_GET_VER_INFO IOCTL()` ioctl should set the `version` and the `num_params` fields in the `iov_param_ver_info` structure and return the values to the calling function. The calling function then uses the `version` and `num_params` parameters to determine the size of buffer required to get parameter descriptions using the `IOV_GET_PARAM_INFO()` ioctl call.

### IOV_GET_PARAM_INFO ioctl

The general flow of control for a driver that calls the `IOV_GET_PARAM_INFO()` ioctl is as follows:

1. Keep an array of `iov_param_desc_t` structures which contain description for each of the configurable params that are supported by the structure. See the `iov_param_desc` structure for the structure description.
2. Copy out the array of `iov_param_desc_t` structures to the `arg` parameter. The fields in the `iov_param_desc_t` structure are static and can be defined at compile time.
   
   The number of elements in the array is the `num_params` value returned by the `IOV_GET_VER_INFO()` ioctl call. The size of the buffer is `sizeof (iov_param_desc_t) * num_params`

### IOV_VALIDATE_PARAM ioctl

The general flow of control for a driver that calls the `IOV_VALIDATE_PARAM()` is as follows:
1. Send the arg parameter to the pci_param_get_ioctl() interface and obtain a pointer to the pci_param_t structure.

2. Write an explanatory string to the pv_reason array if param validation fails.

3. Call the pci_get_plist() interface followed by the pci_plist_lookup() interface to obtain the device parameters.

4. Lookup the vfs name-value pair in the PF plist to obtain the number of VFs to be configured for the validation of this configuration. The drivers should use integer data type of at least 16 bits to look up vfs name-value pair. Use the pciv_plist_getvf() interface to get the plist parameter for the VF devices.

5. Validate the parameters without actually applying them to the device.

6. Return 0 when a valid configuration is found.

**Caution** – The parameters validated in the above procedure are not related to the current configuration of the device in any way. They need to be validated separately assuming they could be a future configuration. If not, the driver should return DDI_EINVAL to indicate improper configuration. The driver should also provide an explanatory string in the pv_reason field in the iov_param_validate structure when invalid configurations are found. This string informs the administrator why the configuration failed.

**Driver Callbacks**

The DDI interfaces ddi_cb_register() and ddi_cb_unregister() are used to register callbacks. Callbacks are used for event notifications and incoming data traffic. Callbacks serve as an event handler for each event during transmission.

The SR-IOV driver should implement additional callbacks to inform the PF driver before and after configuring or unconfiguring the VFs.

The driver can implement the callbacks using the following DDI callback registration mechanism interfaces:

- int ddi_cb_register(dev_info_t *dip, ddi_cb_flags_t flags, ddi_cb_func_t cbfunc, void *arg1, void *arg2, ddi_cb_handle_t *ret_hdlp)

  See the ddi_cb_register(9F) man page for details.

- typedef int(*ddi_cb_func_t)(dev_info_t *dip, ddi_cb_action_t action, void *cbarg, void *arg1, void *arg2)
where:

dip Is a pointer to the `dev_info` structure

cbarg Is a pointer to the structure `pciv_config_VF_t`.

action Is set as `DDI_CB_PCIV_CONFIG_VF` to receive notifications about changes to the VF configuration.

arg1 Private argument to send to itself during each execution of its `cbfunc()` routine.

arg2 Private argument to send to itself during each execution of its `cbfunc()` routine.

See “Register a Callback Handler Function” on page 135 for additional information.

---

**Note** – All PF drivers that are SR-IOV capable must use the `ddi_cb_flags_t DDI_CB_FLAG_SRIOV` to inform the Oracle Solaris IOV framework that the PF drivers are SR-IOV capable.

### Sample Code for Driver Ioctls

```c
enum ioc_reply igb_ioctl(igb_t *igb, struct iocblk *iocp, mblk_t *mp)
{
  int rval = 0;
  iov_param_ver_info_t *iov_param_ver;
  iov_param_validate_t pvalidate;
  pci_param_t my_params;
  char reason[81];

  switch (iocp->ioc_cmd) {
  case IOV_GET_PARAM_VER_INFO:
    if (iocp->ioc_count < sizeof (iov_param_ver_info_t))
      return (IOC_INVAL);
    iov_param_ver = (iov_param_ver_info_t *)(mp->b_cont->b_rptr);
    iov_param_ver->version = IOV_PARAM_DESC_VERSION;
    return (IOC_REPLY);
  case IOV_GET_PARAM_INFO:
    if (iocp->ioc_count <= 0)
      return (IOC_INVAL);
    memcpy((caddr_t)(mp->b_cont->b_rptr), &pci_list,sizeof (pci_list));
    return (IOC_REPLY);
  case IOV_VALIDATE_PARAM:
    if (iocp->ioc_count <= 0)
      return (IOC_INVAL);
    strncpy(reason, "Failed to read params sent\n", 81);
    rval = pci_param_get_ioctl(igb->dip,(uintptr_t)(mp->b_cont->b_rptr),...)
  }
}
```
if (rval == 0) {
    rval = validate_params(igb->dip, my_params, reason);
    pci_param_free(my_params);
}
if (rval) {
    memcpy(mp->b_cont->b_rptr, reason, sizeof(reason));
    iocp->ioc_count = sizeof(reason);
    return (IOC_REPLY);
} iocp->ioc_count = 0;
return (IOC_REPLY);
}
iov_param_ver_info_t iov_param_ver;
iov_param_validate_t pvalidate;
pci_param_t my_params;

switch (cmd) {
case IOV_GET_PARAM_VER_INFO:
    iov_param_ver.version = IOV_PARAM_DESC_VERSION;
    iov_param_ver.num_params = NUM_OF_PARAMS;
    if (ddi_copyout(&iov_param_ver, (caddr_t)arg, sizeof(iov_param_ver_info_t), mode) != DDI_SUCCESS)
        return (DEFAULT);
    return (0);
case IOV_GET_PARAM_INFO:
    if (ddi_copyout(&pci_list, (caddr_t)arg,param_list_size, mode) != DDI_SUCCESS)
        return (DEFAULT);
    return (0);
case IOV_VALIDATE_PARAM:
    strcpy(reason, "Failed to read params sent\n");
    rv = pci_param_get_ioctl(state->dip, arg, mode, &my_params);
    if (rv == 0)
        rv = validate_params(state->dip, my_params, reason);
    else
        return (rv);
    pci_param_free(my_params);
    if (rv) {
        if (ddi_copyout(reason,iov_param_validate_t *)arg)->pv_reason,
            sizeof (reason), mode) != DDI_SUCCESS)
            return (DEFAULT);
        return (rv);
    } return (0);
Building a Device Driver

The third part of this book provides advice on building device drivers for the Oracle Solaris OS:

- Chapter 22, “Compiling, Loading, Packaging, and Testing Drivers,” provides information on compiling, linking, and installing a driver.
- Chapter 23, “Debugging, Testing, and Tuning Device Drivers,” describes techniques for debugging, testing, and tuning drivers.
- Chapter 24, “Recommended Coding Practices,” describes the recommended coding practices for writing drivers.
Compiling, Loading, Packaging, and Testing Drivers

This chapter describes the procedure for driver development, including code layout, compilation, packaging, and testing.

This chapter provides information on the following subjects:

■ “Driver Code Layout” on page 488
■ “Preparing for Driver Installation” on page 490
■ “Installing, Updating, and Removing Drivers” on page 492
■ “Loading and Unloading Drivers” on page 495
■ “Driver Packaging” on page 495
■ “Criteria for Testing Drivers” on page 496

Driver Development Summary

This chapter and the following two chapters, Chapter 23, “Debugging, Testing, and Tuning Device Drivers,” and Chapter 24, “Recommended Coding Practices,” provide detailed information on developing a device driver.

Take the following steps to build a device driver:

1. Write, compile, and link the new code.
   
   See “Driver Code Layout” on page 488 for the conventions on naming files. Use a C compiler to compile the driver. Link the driver using \( \text{ld}(1) \). See “Compiling and Linking the Driver” on page 491 and “Module Dependencies” on page 492.

2. Create the necessary hardware configuration files.
   
   Create a hardware configuration file unique to the device called \( xx . \text{conf} \) where \( xx \) is the prefix for the device. This file is used to update the \( \text{driver.conf}(4) \) file. See “Writing a Hardware Configuration File” on page 492. For a pseudo device driver, create a \( \text{pseudo}(4) \) file.

3. Copy the driver to the appropriate module directory.
See "Copying the Driver to a Module Directory" on page 493.

4. Install the device driver using `add_drv(1M)`.
   See "Installing Drivers with `add_drv`" on page 494. Use the `update_drv(1M)` command to make any changes to the driver. See "Updating Driver Information" on page 494.

5. Load the driver.
   The driver can be loaded automatically by accessing the device. See “Loading and Unloading Drivers” on page 495. Drivers can also be loaded by using the `modload(1M)` command. The `modload` command does not call any routines in the module and therefore is useful for testing. See "Loading and Unloading Test Modules" on page 507.

6. Test the driver.
   Drivers should be rigorously tested in the following areas:
   - "Configuration Testing" on page 496
   - "Functionality Testing" on page 496
   - "Error Handling" on page 497
   - "Testing Loading and Unloading" on page 497
   - "Stress, Performance, and Interoperability Testing" on page 497
   - "DDI/DKI Compliance Testing" on page 498
   - "Installation and Packaging Testing" on page 498

   For additional driver-specific testing, see “Testing Specific Types of Drivers” on page 498.

7. Remove the driver if necessary.
   Use the `rem_drv(1M)` command to remove a device driver. See “Removing the Driver” on page 495.

---

**Driver Code Layout**

The code for a device driver is usually divided into the following files:

- Header files (.h files)
- Source files (.c files)
- Optional configuration file (`driver.conf` file)

**Header Files**

Header files provide the following definitions:

- Data structures specific to the device, such as a structure representing the device registers
- Data structures defined by the driver for maintaining state information
- Defined constants, such as those representing the bits of the device registers
Macros, such as those defining the static mapping between the minor device number and the instance number

Some of the header file definitions, such as the state structure, might be needed only by the device driver. This information should go in private header files that are only included by the device driver itself.

Any information that an application might require, such as the I/O control commands, should be in public header files. These files are included by the driver and by any applications that need information about the device.

While there is no standard for naming private and public files, one convention is to name the private header file xximpl.h and the public header file xxio.h.

**Source Files**

A C source file (a .c file) for a device driver has the following responsibilities:

- Contains the data declarations and the code for the entry points of the driver
- Contains the #include statements that are needed by the driver
- Declares extern references
- Declares local data
- Sets up the cb_ops and dev_ops structures
- Declares and initializes the module configuration section, that is, the modlinkage(9S) and modldrv(9S) structures
- Makes any other necessary declarations
- Defines the driver entry points

**Configuration Files**

In general, the configuration file for a driver defines all of the properties that the driver needs. Entries in the driver configuration file specify possible device instances that the driver can probe for existence. Driver global properties can be set in the driver's configuration file. See the driver.conf(4) man page for more information.

Driver configuration files are required for devices that are not self-identifying.

Driver configuration files are optional for self-identifying devices (SID). For self-identifying devices, the configuration file can be used to add properties into SID nodes.
The following properties are examples of properties that are not set in the driver configuration file:

- Drivers that use the SBus peripheral bus generally get property information from the SBus card. In cases where additional properties are needed, the driver configuration file can contain properties that are defined by `sbus(4)`.
- The properties of a PCI bus can generally be derived from the PCI configuration space. In cases where private driver properties are needed, the driver configuration file can contain properties that are defined by `pci(4)`.
- Drivers on the ISA bus can use additional properties that are defined by `isa(4)`.

Preparing for Driver Installation

The following steps precede installation of a driver:

1. Compile the driver.
2. Create a configuration file if necessary.
3. Identify the driver module to the system through either of the following alternatives:
   - Match the driver’s name to the name of the device node.
   - Use either `add_drv(1M)` or `update_drv(1M)` to inform the system of the module names.

The system maintains a one-to-one association between the name of the driver module and the name of the `dev_info` node. For example, consider a `dev_info` node for a device that is named `mydevice`. The device `mydevice` is handled by a driver module that is also named `mydevice`. The `mydevice` module resides in a subdirectory that is called `drv`, which is in the module path. The module is in `drv/mydevice` if you are using a 32-bit kernel. The module is in `drv/sparcv9/mydevice` if you are using a 64-bit SPARC kernel. The module is in `drv/amd64/mydevice` if you are using a 64-bit x86 kernel.

If the driver is a STREAMS network driver, then the driver name must meet the following constraints:

- Only alphanumeric characters (a-z, A-Z, 0-9), plus the underscore (`_`), are allowed.
- Neither the first nor the last character of the name can be a digit.
- The name cannot exceed 16 characters in length. Names in the range of 3-8 characters in length are preferable.

If the driver must manage `dev_info` nodes with different names, the `add_drv(1M)` utility can create aliases. The `-i` flag specifies the names of other `dev_info` nodes that the driver handles. The `update_drv` command can also modify aliases for an installed device driver.
Compiling and Linking the Driver

You need to compile each driver source file and link the resulting object files into a driver module. The Oracle Solaris OS is compatible with both the Oracle Solaris Studio C compiler and the GNU C compiler from the Free Software Foundation, Inc. The examples in this section use the Oracle Solaris Studio C compiler unless otherwise noted. For information on the Oracle Solaris Studio C compiler, see the Oracle Solaris Studio 12.3: C User’s Guide and the Oracle Solaris Studio Documentation. For more information on compile and link options, see the Oracle Solaris Studio 12.3 Command-Line Reference documentation. The GNU C compiler is supplied in the /usr/sfw directory. For information on the GNU C compiler, see http://gcc.gnu.org/ or check the man pages in /usr/sfw/man.

The example below shows a driver that is called xx with two C source files. A driver module that is called xx is generated. The driver that is created in this example is for a 32-bit kernel. You must use ld -r even if your driver has only one object module.

```bash
% cc -D_KERNEL -c xx1.c
% cc -D_KERNEL -c xx2.c
% ld -r -o xx xx1.o xx2.o
```

The _KERNEL symbol must be defined to indicate that this code defines a kernel module. No other symbols should be defined, except for driver private symbols. The DEBUG symbol can be defined to enable any calls to ASSERT(9F).

<table>
<thead>
<tr>
<th>TABLE 22-1</th>
<th>Compiler Options for SPARC and x86 64–bit Architectures</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Compiler</strong></td>
<td><strong>SPARC 64</strong></td>
</tr>
<tr>
<td>Studio 9</td>
<td>cc -D_KERNEL -xarch=v9 -c xx.c</td>
</tr>
<tr>
<td>Studio 10</td>
<td>cc -D_KERNEL -xarch=v9 -c xx.c cc -D_KERNEL -xarch=amd64 -xmodel=kernel -c xx.c</td>
</tr>
<tr>
<td>Studio 11</td>
<td>cc -D_KERNEL -xarch=v9 -c xx.c cc -D_KERNEL -xarch=amd64 -xmodel=kernel -c xx.c</td>
</tr>
<tr>
<td>Oracle Solaris Studio 12</td>
<td>cc -D_KERNEL -m64 -c xx.c cc -D_KERNEL -m64 -xmodel=kernel -c xx.c</td>
</tr>
</tbody>
</table>

**Note** – If you are compiling for a 32–bit or 64–bit x86 architecture using Oracle Solaris Studio 12.2 or older compilers, do not add -xarch=sse2a since the default value is SSE2.

**Caution** – MMX instructions are not supported in the x86 kernel. Use of MMX instructions triggers a kernel panic and therefore should not be used.
After the driver is stable, you might want to add optimization flags to build a production quality driver. See the cc(1) man page in Oracle Solaris Studio 12.3 Command-Line Reference for specific information on optimizations in the Oracle Solaris Studio C compiler.

Global variables should be treated as volatile in device drivers. The volatile tag is discussed in greater detail in “Declaring a Variable Volatile” on page 534. Use of the flag depends on the platform. See the man pages.

### Module Dependencies

If the driver module depends on symbols exported by another kernel module, the dependency can be specified by the -dy and -N options of the loader, ld(1). If the driver depends on a symbol exported by misc/mySymbol, the example below should be used to create the driver binary.

```
% ld -dy -r -o xx xx1.o xx2.o -N misc/mySymbol
```

### Writing a Hardware Configuration File

If a device is non-self-identifying, the kernel might require a hardware configuration file for that device. If the driver is called xx, the hardware configuration file for the driver should be called xx.conf.

On the x86 platform, device information is now supplied by the booting system. Hardware configuration files should no longer be needed, even for non-self-identifying devices.

See the driver.conf(4), pseudo(4), sbus(4), scsi_free_consistent_buf(9F), and update_drv(1M) man pages for more information on hardware configuration files.

Arbitrary properties can be defined in hardware configuration files. Entries in the configuration file are in the form property=value, where property is the property name and value is its initial value. The configuration file approach enables devices to be configured by changing the property values.

### Installing, Updating, and Removing Drivers

Before a driver can be used, the system must be informed that the driver exists. The add_drv(1M) utility must be used to correctly install the device driver. After a driver is installed, that driver can be loaded and unloaded from memory without using the add_drv command.
Copying the Driver to a Module Directory

Three conditions determine a device driver module's path:

- The platform that the driver runs on
- The architecture for which the driver is compiled
- Whether the path is needed at boot time

Device drivers reside in the following locations:

/platform/`uname -i`/kernel/drv
Contains 32-bit drivers that run only on a specific platform.

/platform/`uname -i`/kernel/drv/sparcv9
Contains 64-bit drivers that run only on a specific SPARC-based platform.

/platform/`uname -i`/kernel/drv/amd64
Contains 64-bit drivers that run only on a specific x86-based platform.

/platform/`uname -m`/kernel/drv
Contains 32-bit drivers that run only on a specific family of platforms.

/platform/`uname -m`/kernel/drv/sparcv9
Contains 64-bit drivers that run only on a specific family of SPARC-based platforms.

/platform/`uname -m`/kernel/drv/amd64
Contains 64-bit drivers that run only on a specific family of x86-based platforms.

/usr/kernel/drv
Contains 32-bit drivers that are independent of platforms.

/usr/kernel/drv/sparcv9
Contains 64-bit drivers on SPARC-based systems that are independent of platforms.

/usr/kernel/drv/amd64
Contains 64-bit drivers on x86-based systems that are independent of platforms.

To install a 32-bit driver, the driver and its configuration file must be copied to a `drv` directory in the module path. For example, to copy a driver to `/usr/kernel/drv`, type:

```bash
$ su
# cp xx /usr/kernel/drv
# cp xx.conf /usr/kernel/drv
```

To install a SPARC driver, copy the driver to a `drv/sparcv9` directory in the module path. Copy the driver configuration file to the `drv` directory in the module path. For example, to copy a driver to `/usr/kernel/drv`, you would type:

```bash
$ su
# cp xx /usr/kernel/drv/sparcv9
# cp xx.conf /usr/kernel/drv
```
To install a 64-bit x86 driver, copy the driver to a `drv/amd64` directory in the module path. Copy the driver configuration file to the `drv` directory in the module path. For example, to copy a driver to `/usr/kernel/drv`, you would type:

```bash
$ su
# cp xx /usr/kernel/drv/amd64
# cp xx.conf /usr/kernel/drv
```

**Note** – All driver configuration files (`.conf` files) must go in the `drv` directory in the module path. The `.conf` files cannot go into any subdirectory of the `drv` directory.

### Installing Drivers with `add_drv`

Use the `add_drv(1M)` command to install the driver in the system. If the driver installs successfully, `add_drv` runs `devfsadm(1M)` to create the logical names in the `/dev` directory.

```bash
# add_drv xx
```

In this case, the device identifies itself as `xx`. The device special files have default ownership and permissions (`0600 root sys`). The `add_drv` command also allows additional names for the device (aliases) to be specified. See the `add_drv(1M)` man page for information on adding aliases and setting file permissions explicitly.

**Note** – Do not use the `add_drv` command to install a STREAMS module. See the *STREAMS Programming Guide* for details.

If the driver creates minor nodes that do not represent terminal devices such as disks, tapes, or ports, you can modify `/etc/devlink.tab` to cause `devfsadm` to create logical device names in `/dev`. Alternatively, logical names can be created by a program that is run at driver installation time.

### Updating Driver Information

Use the `update_drv(1M)` command to notify the system of any changes to an installed device driver. By default, the system re-reads the driver configuration file and reloads the driver binary module.
Removing the Driver

To remove a driver from the system, use the `rem_drv(1M)` command, and then delete the driver module and configuration file from the module path. A driver cannot be used again until that driver is reinstalled with `add_drv(1M)`. The removal of a SCSI HBA driver requires a reboot to take effect.

Loading and Unloading Drivers

Opening a special file (accessing the device) that is associated with a device driver causes that driver to be loaded. You can use the `modload(1M)` command to load the driver into memory, but `modload` does not call any routines in the module. The preferred method is to open the device.

Normally, the system automatically unloads device drivers when the device is no longer in use. During development, you might want to use `modunload(1M)` to unload the driver explicitly. In order for `modunload` to be successful, the device driver must be inactive. No outstanding references to the device should exist, such as through `open(2)` or `mmap(2)`.

The `modunload` command takes a runtime-dependent `module_id` as an argument. To find the `module_id`, use `grep` to search the output of `modinfo(1M)` for the driver name in question. Check in the first column.

```
# modunload -i module-id
```

To unload all currently unloadable modules, specify module ID zero:

```
# modunload -i 0
```

In addition to being inactive, the driver must have working `detach(9E)` and `_fini(9E)` routines for `modunload(1M)` to succeed.

Driver Packaging

The normal delivery vehicle for software is to create a package that contains all of the software components. A package provides a controlled mechanism for installation and removal of all the components of a software product. Developers can utilize the driver actions in the Image Packaging System (IPS) to perform tasks while packaging the driver. Refer to the following documents for more information:

- Chapter 2, "Packaging Software With IPS," in *Packaging and Delivering Software With the Image Packaging System in Oracle Solaris 11.1*
- *Adding and Updating Oracle Solaris 11.1 Software Packages*
In prior versions of Oracle Solaris, the SVR4 packaging system was used, which required including postinstall and preremove scripts in the package to run `add_drv(1M)` and `rem_drv(1M)` commands. Existing driver packages using these commands may still be installed on Oracle Solaris 11.

**Criteria for Testing Drivers**

Once a device driver is functional, that driver should be thoroughly tested prior to distribution. Besides testing the features in traditional UNIX device drivers, Oracle Solaris drivers require testing power management features, such as dynamic loading and unloading of drivers.

**Configuration Testing**

A driver’s ability to handle multiple device configurations is an important part of the test process. Once the driver is working on a simple, or default, configuration, additional configurations should be tested. Depending on the device, configuration testing can be accomplished by changing jumpers or DIP switches. If the number of possible configurations is small, all configurations should be tried. If the number is large, various classes of possible configurations should be defined, and a sampling of configurations from each class should be tested. Defining these classes depends on the potential interactions among the different configuration parameters. These interactions are a function of the type of the device and the way in which the driver was written.

For each device configuration, the basic functions must be tested, which include loading, opening, reading, writing, closing, and unloading the driver. Any function that depends upon the configuration deserves special attention. For example, changing the base memory address of device registers is not likely to affect the behavior of most driver functions. If a driver works well with one address, that driver is likely to work as well with a different address. On the other hand, a special I/O control call might have different effects depending on the particular device configuration.

Loading the driver with varying configurations ensures that the `probe(9E)` and `attach(9E)` entry points can find the device at different addresses. For basic functional testing, using regular UNIX commands such as `cat(1)` or `dd(1M)` is usually sufficient for character devices. Mounting or booting might be required for block devices.

**Functionality Testing**

After a driver has been completely tested for configuration, all of the driver’s functionality should be thoroughly tested. These tests require exercising the operation of all of the driver’s entry points.
Many drivers require custom applications to test functionality. However, basic drivers for
devices such as disks, tapes, or asynchronous boards can be tested using standard system
utilities. All entry points should be tested in this process, including \texttt{devmap(9E)}, \texttt{chpoll(9E)},
and \texttt{ioctl(9E)}, if applicable. The \texttt{ioctl()} tests might be quite different for each driver. For
nonstandard devices, a custom testing application is generally required.

\section*{Error Handling}

A driver might perform correctly in an ideal environment but fail in cases of errors, such as
erroneous operations or bad data. Therefore, an important part of driver testing is the testing of
the driver’s error handling.

All possible error conditions of a driver should be exercised, including error conditions for
actual hardware malfunctions. Some hardware error conditions might be difficult to induce, but
an effort should be made to force or to simulate such errors if possible. All of these conditions
could be encountered in the field. Cables should be removed or be loosened, boards should be
removed, and erroneous user application code should be written to test those error paths. See
also Chapter 13, “Hardening Oracle Solaris Drivers.”

\section*{Caution – Be sure to take proper electrical precautions when testing.}

\section*{Testing Loading and Unloading}

Because a driver that does not load or unload can force unscheduled downtime, loading and
unloading must be thoroughly tested.

A script like the following example should suffice:

\begin{verbatim}
#!/bin/sh
cd <location_of_driver>
while \[1 \] -=
do
    modunload -i `modinfo | grep " <driver_name>" | cut -cl-3` &
    modload <driver_name> &
done
\end{verbatim}

\section*{Stress, Performance, and Interoperability Testing}

To help ensure that a driver performs well, that driver should be subjected to vigorous stress
testing. For example, running single threads through a driver does not test locking logic or
conditional variables that have to wait. Device operations should be performed by multiple
processes at once to cause several threads to execute the same code simultaneously.
Techniques for performing simultaneous tests depend upon the driver. Some drivers require special testing applications, while starting several UNIX commands in the background is suitable for others. Appropriate testing depends upon where the particular driver uses locks and condition variables. Testing a driver on a multiprocessor machine is more likely to expose problems than testing on a single-processor machine.

Interoperability between drivers must also be tested, particularly because different devices can share interrupt levels. If possible, configure another device at the same interrupt level as the one being tested. A stress test can determine whether the driver correctly claims its own interrupts and operates according to expectations. Stress tests should be run on both devices at once. Even if the devices do not share an interrupt level, this test can still be valuable. For example, consider a case in which serial communication devices experience errors when a network driver is tested. The same problem might be causing the rest of the system to encounter interrupt latency problems as well.

Driver performance under these stress tests should be measured using UNIX performance-measuring tools. This type of testing can be as simple as using the `time(1)` command along with commands to be used in the stress tests.

**DDI/DKI Compliance Testing**

To ensure compatibility with later releases and reliable support for the current release, every driver should be DDI/DKI compliant. Check that only kernel routines in `man pages section 9: DDI and DKI Kernel Functions` and `man pages section 9: DDI and DKI Driver Entry Points` and data structures in `man pages section 9: DDI and DKI Properties and Data Structures` are used.

**Installation and Packaging Testing**

Drivers are delivered to customers in packages. A package can be added or be removed from the system using a standard mechanism. The ability of a user to add or remove the package from a system should be tested. In testing, the package should be both installed and uninstalled directly through the `pkg install` command and the `pkg uninstall` commands. See the `Adding and Updating Oracle Solaris 11.1 Software Packages` for more information.

**Testing Specific Types of Drivers**

This section provides some suggestions about how to test certain types of standard devices.

**Testing Tape Drivers**

Tape drivers should be tested by performing several archive and restore operations. The `cpio(1)` and `tar(1)` commands can be used for this purpose. Use the `dd(1M)` command to write an entire disk partition to tape. Next, read back the data, and write the data to another partition...
of the same size. Then compare the two copies. The `mt(1)` command can exercise most of the I/O controls that are specific to tape drivers. See the `mtio(7I)` man page. Try to use all the options. These three techniques can test the error-handling capabilities of tape drivers:

- Remove the tape and try various operations
- Write-protect the tape and try a write
- Turn off power in the middle of different operations

Tape drivers typically implement exclusive-access `open(9E)` calls. These `open()` calls can be tested by opening a device and then having a second process try to open the same device.

**Testing Disk Drivers**

Disk drivers should be tested in both the raw and block device modes. For block device tests, create a new file system on the device. Then try to mount the new file system. Then try to perform multiple file operations.

Note – The file system uses a page cache, so reading the same file over and over again does not really exercise the driver. The page cache can be forced to retrieve data from the device by memory-mapping the file with `mmap(2)`. Then use `msync(3C)` to invalidate the in-memory copies.

Copy another (unmounted) partition of the same size to the raw device. Then use a command such as `fsck(1M)` to verify the correctness of the copy. The new partition can also be mounted and then later compared to the old partition on a file-by-file basis.

**Asynchronous Communication Drivers**

Asynchronous drivers can be tested at the basic level by setting up a `login` line to the serial ports. A good test is to see whether a user can log in on this line. To sufficiently test an asynchronous driver, however, all the I/O control functions must be tested, with many interrupts at high speed. A test involving a loopback serial cable and high data transfer rates can help determine the reliability of the driver. You can run `uucp(1C)` over the line to provide some exercise. However, because `uucp` performs its own error handling, verify that the driver is not reporting excessive numbers of errors to the `uucp` process.

These types of devices are usually STREAMS-based. See the *STREAMS Programming Guide* for more information.

**Testing Network Drivers**

Network drivers can be tested using standard network utilities. The `ftp(1)` and `rcp(1)` commands are useful because the files can be compared on each end of the network. The driver should be tested under heavy network loading, so that various commands can be run by multiple processes.
Heavy network loading includes the following conditions:

- Traffic to the test machine is heavy.
- Traffic among all machines on the network is heavy.

Network cables should be unplugged while the tests are executing to ensure that the driver recovers gracefully from the resulting error conditions. Another important test is for the driver to receive multiple packets in rapid succession, that is, back-to-back packets. In this case, a relatively fast host on a lightly loaded network should send multiple packets in quick succession to the test machine. Verify that the receiving driver does not drop the second and subsequent packets.

These types of devices are usually STREAMS-based. See the STREAMS Programming Guide for more information.

**Testing SR-IOV Drivers**

Drivers supporting SR-IOV require additional testing. Standard bare-metal testing is also required, and the utilities used for bare-metal testing such as ftp and rcp for network devices can be used.


Use the following commands to test the status of the Virtual Functions (VFs):

- VF enabled – `hotplug install`
- VF disabled – `hotplug uninstall`
- VF assigned – `hotplug list`

Use the `lmd(1M)` command in addition to the `hotplug` commands when testing the status of VFs on a SPARC system.

It is also important to test SR-IOV devices in a variety of virtualized configurations. Try the following options when testing SR-IOV drivers on both SPARC and x86 platforms:

- Do not configure any Virtual Functions (VFs)
- Configure only one VF
- Increase configured VFs by powers of 2 until the maximum number of VFs is reached

On SPARC platforms, test the functionality with a varying number of IO Domains and varying distribution of the VFs within those domains. Try the following configurations:

- Assign a single VF to a single IO Domain
- Assign powers of 2 VFs (up to the maximum) to a single IO Domain
- Create 2, 4, or 8 IO Domains and assign varying numbers of VF to each of the domains
- Assign some VFs to the root domain and assign some VFs to an IO Domain

Test the following features if the device or the platform support it:
Criteria for Testing Drivers

- Boot the IO Domain from a VF
- Physically hotplug or unplug the SR-IOV card
- Perform dynamic reconfiguration operations on the SR-IOV card

If your device is supported on multiple versions of Oracle Solaris, a final testing can be performed by mixing the OS versions across the Root and IO Domains for some of the tests.
Debugging, Testing, and Tuning Device Drivers

This chapter presents an overview of the various tools that are provided to assist with testing, debugging, and tuning device drivers. This chapter provides information on the following subjects:

- "Testing Drivers" on page 503 – Testing a driver can potentially impair a system’s ability to function. Use of both serial connections and alternate kernels helps facilitate recovery from crashes.
- “Debugging Tools” on page 511 – Integral debugging facilities enable you to exercise and observe driver features conveniently without having to run a separate debugger.
- “Tuning Drivers” on page 524 – The Oracle Solaris OS provides facilities for measuring the performance of device drivers. Writing kernel statistics structures for your device exports continuous statistics as the device is running. If an area for performance improvement is determined, then the DTrace dynamic instrumentation tool can help determine any problems more precisely.

Testing Drivers

To avoid data loss and other problems, you should take special care when testing a new device driver. This section discusses various testing strategies. For example, setting up a separate system that you control through a serial connection is the safest way to test a new driver. You can load test modules with various kernel variable settings to test performance under different kernel conditions. Should your system crash, you should be prepared to restore backup data, analyze any crash dumps, and rebuild the device directory.

Enable the Deadman Feature to Avoid a Hard Hang

If your system is in a hard hang, then you cannot break into the debugger. If you enable the deadman feature, the system panics instead of hanging indefinitely. You can then use the `kmdb(1)` kernel debugger to analyze your problem.
The deadman feature checks every second whether the system clock is updating. If the system clock is not updating, then you are in an indefinite hang. If the system clock has not been updated for 50 seconds, the deadman feature induces a panic and puts you in the debugger.

Take the following steps to enable the deadman feature:

1. Make sure you are capturing crash images with `dumpadm(1M)`.
2. Set the snooping variable in the `/etc/system` file. See the `system(4)` man page for information on the `/etc/system` file.
   ```
   set snooping=1
   ```
3. Reboot the system so that the `/etc/system` file is read again and the snooping setting takes effect.

Note that any zones on your system inherit the deadman setting as well.

If your system hangs while the deadman feature is enabled, you should see output similar to the following example on your console:

```
panic[cpu1]/thread=30018dd6cc0: deadman: timed out after 9 seconds of clock inactivity
panic: entering debugger (continue to save dump)
```

Inside the debugger, use the `::cpuinfo` command to investigate why the clock interrupt was not able to fire and advance the system time.

### Testing With a Serial Connection

Using a serial connection is a good way to test drivers. Use the `tip(1)` command to make a serial connection between a host system and a test system. With this approach, the `tip window` on the host console is used as the console of the test machine. See the `tip(1)` man page for additional information.

A tip window has the following advantages:

- Interactions with the test system and kernel debuggers can be monitored. For example, the window can keep a log of the session for use if the driver crashes the test system.
- The test machine can be accessed remotely by logging into a `tip host` machine and using `tip(1)` to connect to the test machine.

**Note** – Although using a tip connection and a second machine are not required to debug an Oracle Solaris device driver, this technique is still recommended.
To Set Up the Host System for a tip Connection

1. Connect the host system to the test machine using serial port A on both machines.
   This connection must be made with a null modem cable.

2. On the host system, make sure there is an entry in /etc/remote for the connection. See the
   remote(4) man page for details.
   The terminal entry must match the serial port that is used. The operating system comes with
   the correct entry for serial port B, but a terminal entry must be added for serial port A:
   ```
   debug:
   :dv=/dev/term/a:br#9600:el="C^S"Q"U":ie=%$:oe="D:
   ```
   Note – The baud rate must be set to 9600.

3. In a shell window on the host, run `tip(1)` and specify the name of the entry:
   ```
   % tip debug
   connected
   ```
   The shell window is now a tip window with a connection to the console of the test machine.

   Caution – Do not use STOP-A for SPARC machines or F1-A for x86 architecture machines on
   the host machine to stop the test machine. This action actually stops the host machine. To send a
   break to the test machine, type -# in the tip window. Commands such as -# are recognized only
   if these characters on first on the line. If the command has no effect, press either the Return key
   or Control-U.

Setting Up a Target System on the SPARC Platform

A quick way to set up the test machine on the SPARC platform is to unplug the keyboard before
turning on the machine. The machine then automatically uses serial port A as the console.

Another way to set up the test machine is to use boot PROM commands to make serial port A
the console. On the test machine, at the boot PROM ok prompt, direct console I/O to the serial
line. To make the test machine always come up with serial port A as the console, set the
environment variables: input-device and output-device.

```bash
EXAMPLE 23-1 Setting input-device and output-device With Boot PROM Commands
ok setenv input-device ttya
ok setenv output-device ttya
```

The eeprom command can also be used to make serial port A the console. As superuser, execute
the following commands to make the input-device and output-device parameters point to
serial port A. The following example demonstrates the eeprom command.
EXAMPLE 23–2 Setting input-device and output-device With the eeprom Command

```bash
# eeprom input-device=ttya
# eeprom output-device=ttya
```

The eeprom commands cause the console to be redirected to serial port A at each subsequent system boot.

**Setting Up a Target System on the x86 Platform**

On x86 platforms, use the eeprom command to make serial port A the console. This procedure is the same as the SPARC platform procedure. See “Setting Up a Target System on the SPARC Platform” on page 505. The eeprom command causes the console to switch to serial port A (COM1) during reboot.

---

**Note** – x86 machines do not transfer console control to the tip connection until an early stage in the boot process unless the BIOS supports console redirection to a serial port. In SPARC machines, the tip connection maintains console control throughout the boot process.

---

**Setting Up Test Modules**

The `system(4)` file in the `/etc` directory enables you to set the value of kernel variables at boot time. With kernel variables, you can toggle different behaviors in a driver and take advantage of debugging features that are provided by the kernel. The kernel variables `moddebug` and `kmem_flags`, which can be very useful in debugging, are discussed later in this section. See also “Enable the Deadman Feature to Avoid a Hard Hang” on page 503.

Changes to kernel variables after boot are unreliable, because `/etc/system` is read only once when the kernel boots. After this file is modified, the system must be rebooted for the changes to take effect. If a change in the file causes the system not to work, boot with the `ask (-a)` option. Then specify `/dev/null` as the system file.

---

**Note** – Kernel variables cannot be relied on to be present in subsequent releases.

---

**Setting Kernel Variables**

The `set` command changes the value of module or kernel variables. To set module variables, specify the module name and the variable:

```
set module_name:variable=value
```

For example, to set the variable `test_debug` in a driver that is named `myTest`, use `set` as follows:

```
% set myTest:test_debug=1
```
To set a variable that is exported by the kernel itself, omit the module name.

You can also use a bitwise OR operation to set a value, for example:

```
% set moddebug | 0x80000000
```

## Loading and Unloading Test Modules

The commands `modload(1M)`, `modunload(1M)`, and `modinfo(1M)` can be used to add test modules, which is a useful technique for debugging and stress-testing drivers. These commands are generally not needed in normal operation, because the kernel automatically loads needed modules and unloads unused modules. The `moddebug` kernel variable works with these commands to provide information and set controls.

### Using the modload() Function

Use `modload(1M)` to force a module into memory. The `modload` command verifies that the driver has no unresolved references when that driver is loaded. Loading a driver does not necessarily mean that the driver can attach. When a driver loads successfully, the driver's `_info(9E)` entry point is called. The `attach()` entry point is not necessarily called.

### Using the modinfo() Function

Use `modinfo(1M)` to confirm that the driver is loaded.

**EXAMPLE 23-3  Using modinfo to Confirm a Loaded Driver**

```
$ modinfo
          Id Loadaddr  Size Info Rev Module Name
6  101b6000  732  -  1 obsym (OBP symbol callbacks)
7  101b65bd 1acd0 226  1 rpcmod (RPC syscall)
7  101b65bd 1acd0 226  1 rpcmod (32-bit RPC syscall)
7  101b65bd 1acd0  1  1 rpcmod (rpc interface str mod)
8  101ce8dd 74600  0  1 ip (IP STREAMS module)
8  101ce8dd 74600  3  1 ip (IP STREAMS device)
...
$ modinfo | grep mydriver
169 781a8d78 13fb  0  1 mydriver (Test Driver 1.5)
```

The number in the `info` field is the major number that has been chosen for the driver. The `modunload(1M)` command can be used to unload a module if the module ID is provided. The module ID is found in the left column of `modinfo` output.

Sometimes a driver does not unload as expected after a `modunload` is issued, because the driver is determined to be busy. This situation occurs when the driver fails `detach(9E)`, either because the driver really is busy, or because the `detach` entry point is implemented incorrectly.
**Using modunload()**

To remove all of the currently unused modules from memory, run `modunload(1M)` with a module ID of 0:

```bash
# modunload -i 0
```

**Setting the moddebug Kernel Variable**

The `moddebug` kernel variable controls the module loading process. The possible values of `moddebug` are:

- **0x80000000**: Prints messages to the console when loading or unloading modules.
- **0x40000000**: Gives more detailed error messages.
- **0x20000000**: Prints more detail when loading or unloading, such as including the address and size.
- **0x00001000**: No auto-unloading drivers. The system does not attempt to unload the device driver when the system resources become low.
- **0x00000080**: No auto-unloading streams. The system does not attempt to unload the STREAMS module when the system resources become low.
- **0x00000010**: No auto-unloading of kernel modules of any type.
- **0x00000001**: If running with `kmdb`, `moddebug` causes a breakpoint to be executed and a return to `kmdb` immediately before each module’s `_init()` routine is called. This setting also generates additional debug messages when the module’s `_info()` and `_fini()` routines are executed.

**Setting kmem_flags Debugging Flags**

The `kmem_flags` kernel variable enables debugging features in the kernel’s memory allocator. Set `kmem_flags` to `0xf` to enable the allocator’s debugging features. These features include runtime checks to find the following code conditions:

- Writing to a buffer after the buffer is freed
- Using memory before the memory is initialized
- Writing past the end of a buffer

The *Oracle Solaris Modular Debugger Guide* describes how to use the kernel memory allocator to analyze such problems.
Note - Testing and developing with kmem_flags set to 0xf can help detect latent memory corruption bugs. Because setting kmem_flags to 0xf changes the internal behavior of the kernel memory allocator, you should thoroughly test without kmem_flags as well.

Avoiding Data Loss on a Test System

A driver bug can sometimes render a system incapable of booting. By taking precautions, you can avoid system reinstallation in this event, as described in this section.

Using an Alternate Boot Environment

A number of driver-related system files are difficult, if not impossible, to reconstruct. Files such as /etc/name_to_major, /etc/driver_aliases, /etc/driver_classes, and /etc/minor_perm can be corrupted if the driver crashes the system during installation. See the add_drv(1M) man page.

To be safe, use the beadm(1M) command to make a backup copy of the root file system after the test machine is in the proper configuration. If you plan to modify the /etc/system file, make a backup copy of the file before making modifications.

Booting With an Alternate Kernel

See the Chapter 4, "Administering Boot Environments," in Creating and Administering Oracle Solaris 11.1 Boot Environments and "Booting From a ZFS Boot Environment (Task Map)" in Oracle Solaris Administration: Common Tasks for detailed information.

Consider Alternative Back-Up Plans

If the system is attached to a network, the test machine can be added as a client of a server. If a problem occurs, the system can be booted from the network. The local disks can then be mounted, and any fixes can be made. Alternatively, the system can be booted directly from the Oracle Solaris system CD-ROM.

Another way to recover from disaster is to have another bootable root file system. Use format(1M) to make a partition that is the exact size of the original. Then use dd(1M) to copy the bootable root file system. After making a copy, run fsck(1M) on the new file system to ensure its integrity.

Subsequently, if the system cannot boot from the original root partition, boot the backup partition. Use dd(1M) to copy the backup partition onto the original partition. You might have a situation where the system cannot boot even though the root file system is undamaged. For
example, the damage might be limited to the boot block or the boot program. In such a case, you can boot from the backup partition with the ask (-a) option. You can then specify the original file system as the root file system.

**Capture System Crash Dumps**

When a system panics, the system writes an image of kernel memory to the dump device. The dump device is by default the most suitable swap device. The dump is a system crash dump, similar to core dumps generated by applications. On rebooting after a panic, `savecore(1M)` checks the dump device for a crash dump. If a dump is found, `savecore` makes a copy of the kernel’s symbol table, which is called `unix. n`. The `savecore` utility then dumps a core file that is called `vncore. n` in the core image directory. By default, the core image directory is `/var/crash/machine_name`. If `/var/crash` has insufficient space for a core dump, the system displays the needed space but does not actually save the dump. The `mdb(1)` debugger can then be used on the core dump and the saved kernel.

In the Oracle Solaris operating system, crash dump is enabled by default. The `dumpadm(1M)` command is used to configure system crash dumps. Use the `dumpadm` command to verify that crash dumps are enabled and to determine the location of core files that have been saved.

**Note** – You can prevent the `savecore` utility from filling the file system. Add a file that is named `minfree` to the directory in which the dumps are to be saved. In this file, specify the number of kilobytes to remain free after `savecore` has run. If insufficient space is available, the core file is not saved.

**Recovering the Device Directory**

Damage to the `/devices` and `/dev` directories can occur if the driver crashes during `attach(9E)`. If either directory is damaged, you can rebuild the directory by booting the system and running `fsck(1M)` to repair the damaged root file system. The root file system can then be mounted. Recreate the `/devices` and `/dev` directories by running `devfsadm(1M)` and specifying the `/devices` directory on the mounted disk.

The following example shows how to repair a damaged root file system on a SPARC system. In this example, the damaged disk is `/dev/dsk/c0t3d0s0`, and an alternate boot disk is `/dev/dsk/c0t1d0s0`.

**EXAMPLE 23-4  Recovering a Damaged Device Directory**

```
ok boot disk1
...
Rebooting with command: boot kernel.test/sparcv9/unix
Boot device: /sbus@1f,0/espdma@e,8400000/esp@e,8000000/sd@31,0:a File and \
  args:
```
EXAMPLE 23–4  Recovering a Damaged Device Directory  (Continued)

kernel.test/sparcv9/unix
...
# fsck /dev/dsk/c0t3d0s0** /dev/dsk/c0t3d0s0
** Last Mounted on /
** Phase 1 - Check Blocks and Sizes
** Phase 2 - Check Pathnames
** Phase 3 - Check Connectivity
** Phase 4 - Check Reference Counts
** Phase 5 - Check Cyl groups
1478 files, 9922 used, 29261 free
   (141 frags, 3640 blocks, 0.4% fragmentation)
# mount /dev/dsk/c0t3d0s0 /mnt
# devfsadm -r /mnt

Note – A fix to the /devices and /dev directories can allow the system to boot while other parts of the system are still corrupted. Such repairs are only a temporary fix to save information, such as system crash dumps, before reinstalling the system.

---

**Debugging Tools**

This section describes two debuggers that can be applied to device drivers. Both debuggers are described in detail in the *Oracle Solaris Modular Debugger Guide*.

- The `kmdb(1)` **kernel debugger** provides typical runtime debugger facilities, such as breakpoints, watch points, and single-stepping. The `kmdb` debugger supersedes `kadb`, which was available in previous releases. The commands that were previously available from `kadb` are used in `kmdb`, in addition to new functionality. Where `kadb` could only be loaded at boot time, `kmdb` can be loaded at any time. The `kmdb` debugger is preferred for live, interactive debugging due to its execution controls.

- The `mdb(1)` **modular debugger** is more limited than `kmdb` as a real-time debugger, but `mdb` has rich facilities for postmortem debugging.

The `kmdb` and `mdb` debuggers mostly share the same user interface. Many debugging techniques therefore can be applied with the same commands in both tools. Both debuggers support macros, dcmds, and dmods. A `dcmd` (pronounced dee-command) is a routine in the debugger that can access any of the properties of the current target program. A `dcmd` can be dynamically loaded at runtime. A `dmod`, which is short for debugger module, is a package of dcmds that can be loaded to provide non-standard behavior.
Both mdb and kmdb are backward-compatible with legacy debuggers such as adb and kadb. The mdb debugger can execute all of the macros that are available to kmdb as well as any legacy user-defined macros for adb. See the Oracle Solaris Modular Debugger Guide for information about where to find standard macro sets.

**Postmortem Debugging**

Postmortem analysis offers numerous advantages to driver developers. More than one developer can examine a problem in parallel. Multiple instances of the debugger can be used simultaneously on a single crash dump. The analysis can be performed offline so that the crashed system can be returned to service, if possible. Postmortem analysis enables the use of user-developed debugger functionality in the form of dmods. Dmods can bundle functionality that would be too memory-intensive for real-time debuggers, such as kmdb.

When a system panics while kmdb is loaded, control is passed to the debugger for immediate investigation. If kmdb does not seem appropriate for analyzing the current problem, a good strategy is to use :c to continue execution and save the crash dump. When the system reboots, you can perform postmortem analysis with mdb on the saved crash dump. This process is analogous to debugging an application crash from a process core file.

---

**Note** – In earlier versions of the Oracle Solaris operating system, adb(1) was the recommended tool for postmortem analysis. In the current Oracle Solaris operating system, mdb(1) is the recommended tool for postmortem analysis. The mdb() feature set surpasses the set of commands from the legacy crash(1M) utility. The crash utility is no longer available in the Oracle Solaris operating system.

**Using the kmdb Kernel Debugger**

The kmdb debugger is an interactive kernel debugger that provides the following capabilities:

- Control of kernel execution
- Inspection of the kernel state
- Live modifications to the code

This section assumes that you are already familiar with the kmdb debugger. The focus in this section is on kmdb capabilities that are useful in device driver design. To learn how to use kmdb in detail, refer to the kmdb(1) man page and to the Oracle Solaris Modular Debugger Guide. If you are familiar with kadb, refer to the kadb(1M) man page for the major differences between kadb and kmdb.

The kmdb debugger can be loaded and unloaded at will. Instructions for loading and unloading kmdb are in the Oracle Solaris Modular Debugger Guide. For safety and convenience, booting
with an alternate kernel is highly encouraged. The boot process is slightly different between the SPARC platform and the x86 platform, as described in this section.

**Note** – By default, `kmdb` uses the CPU ID as the prompt when `kmdb` is running. In the examples in this chapter `[0]` is used as the prompt unless otherwise noted.

### Booting `kmdb` With an Alternate Kernel on the SPARC Platform

Use either of the following commands to boot a SPARC system with both `kmdb` and an alternate kernel:

```
boot kmdb -D kernel.test/sparcv9/unix
boot kernel.test/sparcv9/unix -k
```

### Booting `kmdb` With an Alternate Kernel on the x86 Platform

Use either of the following commands to boot an x86 system with both `kmdb` and an alternate kernel:

```
b kmdb -D kernel.test/unix
b kernel.test/unix -k
```

### Setting Breakpoints in `kmdb`

Use the `bp` command to set a breakpoint, as shown in the following example.

**EXAMPLE 23–5** Setting Standard Breakpoints in `kmdb`

```
[0]> myModule'"myBreakpointLocation":bp
```

If the target module has not been loaded, then an error message that indicates this condition is displayed, and the breakpoint is not created. In this case you can use a *deferred breakpoint*. A deferred breakpoint activates automatically when the specified module is loaded. Set a deferred breakpoint by specifying the target location after the `bp` command. The following example demonstrates a deferred breakpoint.

**EXAMPLE 23–6** Setting Deferred Breakpoints in `kmdb`

```
[0]>::bp myModule'"myBreakpointLocation"
```

For more information on using breakpoints, see the *Oracle Solaris Modular Debugger Guide*. You can also get help by typing either of the following two lines:
The \texttt{kmdb}(1M) debugger supports macros that can be used to display kernel data structures. Use \texttt{$M$} to display \texttt{kmdb} macros. Macros are used in the form:

\begin{verbatim}
[ address ] $<\text{macroname}
\end{verbatim}

\textbf{Note} – Neither the information displayed by these macros nor the format in which the information is displayed, constitutes an interface. Therefore, the information and format can change at any time.

The \texttt{kmdb} macros in the following table are particularly useful to developers of device drivers. For convenience, legacy macro names are shown where applicable.

\begin{table}[h]
\centering
\caption{\texttt{kmdb} Macros}
\begin{tabular}{lll}
\hline
\textbf{Dcmd} & \textbf{Legacy Macro} & \textbf{Description} \\
\hline
::devinfo & devinfo & Print a summary of a device node \\
 & devinfo\_brief & \\
 & devinfo\_prop & \\
::walk devinfo\_parents & devinfo\_parent & Walk the ancestors of a device node \\
::walk devinfo\_sibling & devinfo\_sibling & Walk the siblings of a device node \\
::minornodes & devinfo\_minor & Print the minor nodes that correspond to the given device node \\
::major2name & & Print the name of a device that is bound to a given device node. \\
::devbindings & & Print the device nodes that are bound to a given device node or major number. \\
\hline
\end{tabular}
\end{table}

The \texttt{::devinfo dcmd} displays a node state that can have one of the following values:

\begin{tabular}{ll}
\hline
\texttt{DS\_ATTACHED} & The driver’s \texttt{attach(9E)} routine returned successfully. \\
\texttt{DS\_BOUND} & The node is bound to a driver, but the driver’s \texttt{probe(9E)} routine has not yet been called. \\
\hline
\end{tabular}
The parent nexus has assigned a bus address for the driver. The implementation-specific initializations have been completed. The driver’s probe(9E) routine has not yet been called at this point.

The device node has been linked into the kernel’s device tree, but the system has not yet found a driver for this node.

The driver’s probe(9E) routine returned successfully.

The device is fully configured.

**Using the `mdb` Modular Debugger**

The `mdb`(1) modular debugger can be applied to the following types of files:

- Live operating system components
- Operating system crash dumps
- User processes
- User process core dumps
- Object files

The `mdb` debugger provides sophisticated debugging support for analyzing kernel problems. This section provides an overview of `mdb` features. For a complete discussion of `mdb`, refer to the *Oracle Solaris Modular Debugger Guide*.

Although `mdb` can be used to alter live kernel state, `mdb` lacks the kernel execution control that is provided by `kmdb`. As a result `kmdb` is preferred for runtime debugging. The `mdb` debugger is used more for static situations.

**Note** – The prompt for `mdb` is `>`,

**Getting Started With the Modular Debugger**

The `mdb` debugger provides an extensive programming API for implementing debugger modules so that driver developers can implement custom debugging support. The `mdb` debugger also provides many usability features, such as command-line editing, command history, an output pager, and online help.

**Note** – The `adb` macros should no longer be used. That functionality has largely been superseded by the `dcmds` in `mdb`. 
The `mdb` debugger provides a rich set of modules and dcms. With these tools, you can debug the Oracle Solaris kernel, any associated modules, and device drivers. These facilities enable you to perform tasks such as:

- Formulate complex debugging queries
- Locate all the memory allocated by a particular thread
- Print a visual picture of a kernel STREAM
- Determine what type of structure a particular address refers to
- Locate leaked memory blocks in the kernel
- Analyze memory to locate stack traces
- Assemble dcms into modules called `dmods` for creating customized operations

To get started, switch to the crash directory and type `mdb`, specifying a system crash dump, as illustrated in the following example.

**EXAMPLE 23–7** Invoking `mdb` on a Crash Dump

```
% cd /var/crash/testsystem
% ls
bounds unix.0 vmcore.0
% mdb unix.0 vmcore.0
Loading modules: [ unix krtld genunix ufs_log ip usba s1394 cpc nfs ]
> ::status
debugging crash dump vmcore.0 (64-bit) from testsystem
operating system: 5.10 Generic (sun4u)
panic message: zero
dump content: kernel pages only
```

When `mdb` responds with the > prompt, you can run commands.

To examine the running kernel on a live system, run `mdb` from the system prompt as follows.

**EXAMPLE 23–8** Invoking `mdb` on a Running Kernel

```
# mdb -k
Loading modules: [ unix krtld genunix ufs_log ip usba s1394 ptm cpc ipc nfs ]
> ::status
debugging live kernel (64-bit) on testsystem
operating system: 5.10 Generic (sun4u)
```

**Useful Debugging Tasks With kmdb and mdb**

This section provides examples of useful debugging tasks. The tasks in this section can be performed with either `mdb` or `kmdb` unless specifically noted. This section assumes a basic knowledge of the use of `kmdb` and `mdb`. Note that the information presented here is dependent on the type of system used. A Sun Blade 100 workstation running the 64-bit kernel was used to produce these examples.
Caution – Because irreversible destruction of data can result from modifying data in kernel structures, you should exercise extreme caution. Do not modify or rely on data in structures that are not part of the Oracle Solaris DDI. See the Intro(9S) man page for information on structures that are part of the Oracle Solaris DDI.

Exploring System Registers With kmdb

The kmdb debugger can display machine registers as a group or individually. To display all registers as a group, use $r as shown in the following example.

EXAMPLE 23–9  Reading All Registers on a SPARC Processor With kmdb

[0]$: $r

g0  0  10  0
g1  100130a4 debug_enter  11  edd00028
g2  10411c00 tsmisst_area+0xe00  12  10449c90
g3  10442000 ti_statetbl+0x1ba  13  1b
g4  3000061a004  14  10474400 ecc syndrome_tab+0x80
g5  0  15  3b9aca00
g6  0  16  0
g7  2a10001fd40  17  0
08  0  10  0
09  c  11  10449e50
02  20  12  0
03  300006b2d08  13  10
04  0  14  0
05  0  15  b0
sp  2a10001b451  fp  2a10001b521
o7  1001311c debug enter+0x78  17  1034bb24 zsa xsint+0x2c4
y  0
tstate: 1604 (ccr=0x0, asi=0x0, pstate=0x16, cwp=0x4)
pstate: ag:0 ie:1 priv:1 am:0 pef:1 mm:0 tle:0 cle:0 mg:0 ig:0
winreg: cur:4 other:0 clean:7 cansave:1 canrest:5 wstate:14
tba  0x10000000
pc  edd000d8 edd000d8: ta %icc,%g0 + 125
npc  edd000dc edd000dc: nop

The debugger exports each register value to a variable with the same name as the register. If you read the variable, the current value of the register is returned. If you write to the variable, the value of the associated machine register is changed. The following example changes the value of the %o0 register from 0 to 1 on an x86 machine.

EXAMPLE 23–10  Reading and Writing Registers on an x86 Machine With kmdb

[0]> &eax=K
c1e6e0f0
[0]> 0>eax
[0]> &eax=K
0
[0]> c1e6e0f0>eax

Chapter 23  •  Debugging, Testing, and Tuning Device Drivers 517
If you need to inspect the registers of a different processor, you can use the `::cpuregs` dcmd. The ID of the processor to be examined can be supplied as either the address to the dcmd or as the value of the `-c` option, as shown in the following example.

**EXAMPLE 23–11** Inspecting the Registers of a Different Processor

```
[0]> 0::cpuregs
%cs = 0x0158 %eax = 0xc1e6e0f0 kmdbmod'kaif_dvec
%ds = 0x0160 %ebx = 0x00000000
```

The following example switches from processor 0 to processor 3 on a SPARC machine. The `%g3` register is inspected and then cleared. To confirm the new value, `%g3` is read again.

**EXAMPLE 23–12** Retrieving the Value of an Individual Register From a Specified Processor

```
[0]> 3::switch
[3]> <g3=K
24
[3]> 0-g3
[3]> <g3
```

### Detecting Kernel Memory Leaks

The `::findleaks` dcmd provides powerful, efficient detection of memory leaks in kernel crash dumps. The full set of kernel-memory debugging features must be enabled for `::findleaks` to be effective. For more information, see “Setting `kmem_flags` Debugging Flags” on page 508. Run `::findleaks` during driver development and testing to detect code that leaks memory, thus wasting kernel resources. See Chapter 9, “Debugging With the Kernel Memory Allocator,” in Oracle Solaris Modular Debugger Guide for a complete discussion of `::findleaks`.

**Note** — Code that leaks kernel memory can render the system vulnerable to denial-of-service attacks.

### Writing Debugger Commands With `mdb`

The `mdb` debugger provides a powerful API for implementing debugger facilities that you customize to debug your driver. The Oracle Solaris Modular Debugger Guide explains the programming API in detail.

The SUNWmdbdm package installs sample `mdb` source code in the directory `/usr/demo/mdb`. You can use `mdb` to automate lengthy debugging chores or help to validate that your driver is behaving properly. You can also package your `mdb` debugging modules with your driver product. With packaging, these facilities are available to service personnel at a customer site.
Obtaining Kernel Data Structure Information

The Oracle Solaris kernel provides data type information in structures that can be inspected with either kmdb or mdb.

\[\text{Note – The kmdb and mdb dcmds can be used only with objects that contain compressed symbolic debugging information that has been designed for use with mdb. This information is currently available only for certain Oracle Solaris kernel modules. The SUNWzlib package must be installed to process the symbolic debugging information.}\]

The following example demonstrates how to display the data in the scsi_pkt structure.

EXAMPLE 23–13 Displaying Kernel Data Structures With a Debugger

\[
> \text{7079ceb0::print -t 'struct scsi_pkt'}
\]

\[
\{
\quad \text{opaque_t pkt_ha_private = 0x7079ce20}
\quad \text{struct scsi_address pkt_address = {}
\quad \quad \text{struct scsi_hba_tran *a_hba_tran = 0x70175e68}
\quad \quad \text{ushort_t a_target = 0x6}
\quad \quad \text{uchar_t a_lun = 0}
\quad \quad \text{uchar_t a_sublun = 0}
\quad \}
\quad \text{opaque_t pkt_private = 0x708db4d0}
\quad \text{int (*)() *pkt_comp = sd_intr}
\quad \text{uint_t pkt_flags = 0}
\quad \text{int pkt_time = 0x78}
\quad \text{uchar_t *pkt_scbp = 0x7079ce74}
\quad \text{uchar_t *pkt_cdbp = 0x7079ce64}
\quad \text{ssize_t pkt_resid = 0}
\quad \text{uint_t pkt_state = 0x37}
\quad \text{uint_t pkt_statistics = 0}
\quad \text{uchar_t pkt_reason = 0}
\}
\]

The size of a data structure can be useful in debugging. Use the ::sizeof dcmd to obtain the size of a structure, as shown in the following example.

EXAMPLE 23–14 Displaying the Size of a Kernel Data Structure

\[
> \text{::sizeof struct scsi_pkt}
\]

\[
\text{sizeof (struct scsi_pkt) = 0x58}
\]

The address of a specific member within a structure is also useful in debugging. Several methods are available for determining a member’s address. Use the ::offsetof dcmd to obtain the offset for a given member of a structure, as in the following example.
EXAMPLE 23–15  Displaying the Offset to a Kernel Data Structure

```c
> ::offsetof struct scsi_pkt pkt_state
offsetof (struct pkt_state) = 0x48
```

Use the `::print` dcmd with the `-a` option to display the addresses of all members of a structure, as in the following example.

EXAMPLE 23–16  Displaying the Relative Addresses of a Kernel Data Structure

```c
> ::print -a struct scsi_pkt
{ 
  0 pkt_ha_private 
  8 pkt_address { 
    ... 
  } 
  18 pkt_private 
  ... 
}
```

If an address is specified with `::print` in conjunction with the `-a` option, the absolute address for each member is displayed.

EXAMPLE 23–17  Displaying the Absolute Addresses of a Kernel Data Structure

```c
> 10000000::print -a struct scsi_pkt
{ 
  10000000 pkt_ha_private 
  10000008 pkt_address { 
    ... 
  } 
  10000018 pkt_private 
  ... 
}
```

The `::print`, `::sizeof` and `::offsetof` dcmds enable you to debug problems when your driver interacts with the Oracle Solaris kernel.

**Caution** – This facility provides access to raw kernel data structures. You can examine any structure whether or not that structure appears as part of the DDI. Therefore, you should refrain from relying on any data structure that is not explicitly part of the DDI.
Note – These dcmds should be used only with objects that contain compressed symbolic debugging information that has been designed for use with mdb. Symbolic debugging information is currently available for certain Oracle Solaris kernel modules only. The SUNWzlib (32-bit) or SUNWzlibx (64-bit) decompression software must be installed to process the symbolic debugging information. The kmdb debugger can process symbolic type data with or without the SUNWzlib or SUNWzlibx packages.

Obtaining Device Tree Information

The mdb debugger provides the ::prtconf dcmd for displaying the kernel device tree. The output of the ::prtconf dcmd is similar to the output of the prtconf(1M) command.

EXAMPLE 23–18 Using the ::prtconf Dcmd

```shell
> ::prtconf
300015d3e08 SUNW,Sun-Blade-100
   300015d3c28 packages (driver not attached)
      300015d3868 SUNW,builtin-drivers (driver not attached)
      300015d3a48 disk-label (driver not attached)
      300015d32c8 terminal-emulator (driver not attached)
      300015d30e8 obp-tftp (driver not attached)
      300015d2f08 dropins (driver not attached)
      300015d2d28 kbd-translator (driver not attached)
      300015d2b48 ufs-file-system (driver not attached)
      300015d3a48 chosen (driver not attached)
      300015d2968 openprom (driver not attached)
```

You can display the node by using a macro, such as the ::devinfo dcmd, as shown in the following example.

EXAMPLE 23–19 Displaying Device Information for an Individual Node

```shell
> 300015d3e08::devinfo
300015d3e08 SUNW,Sun-Blade-100
   System properties at 0x300015abcd0:
      name='relative-addressing' type=int items=1
         value=00000001
      name='MMU_PAGEOFFSET' type=int items=1
         value=00001fff
      name='MMU_PAGESIZE' type=int items=1
         value=00002000
      name='PAGESIZE' type=int items=1
         value=00002000
   Driver properties at 0x300015abe00:
      name='pm-hardware-state' type=string items=1
         value='no-suspend-resume'
```

Chapter 23 • Debugging, Testing, and Tuning Device Drivers 521
Use ::prtconf to see where your driver has attached in the device tree, and to display device properties. You can also specify the verbose (-v) flag to ::prtconf to display the properties for each device node, as follows.

**EXAMPLE 23-20** Using the ::prtconf Dcmd in Verbose Mode

```bash
> ::prtconf -v
DEVDINFO
300015d3e08 SUNW,Sun-Blade-100
System properties at 0x300015abdc0:
   name='relative-addressing' type=int items=1
      value=00000001
   name='MMU_PAGEOFFSET' type=int items=1
      value=00001fff
   name='MMU_PAGESIZE' type=int items=1
      value=00002000
   name='PAGESIZE' type=int items=1
      value=00002000
Driver properties at 0x300015abe00:
   name='pm-hardware-state' type=string items=1
      value='no-suspend-resume'
...```

```
300015ce798 pci10b9,5229, instance #0
Driver properties at 0x300015abe00:
   name='target2-dcd-options' type=any items=4
      value=00.00.00.a4
   name='target1-dcd-options' type=any items=4
      value=00.00.00.a2
   name='target0-dcd-options' type=any items=4
      value=00.00.00.a4
```

Another way to locate instances of your driver is the ::devbindings dcmd. Given a driver name, the command displays a list of all instances of the named driver as demonstrated in the following example.

**EXAMPLE 23-21** Using the ::devbindings Dcmd to Locate Driver Instances

```bash
> ::devbindings dad
300015ce3d0 ide-disk (driver not attached)
300015c9a60 dad, instance #0
System properties at 0x300015ab400:
   name='lun' type=int items=1
      value=00000000
   name='target' type=int items=1
      value=00000000
   name='class_prop' type=string items=1
      value='ata'
   name='type' type=string items=1
      value='ata'
   name='class' type=string items=1
      value='dada'
...```

```
300015c9880 dad, instance #1
System properties at 0x300015ab080:
```
EXAMPLE 23–21  Using the ::devbindings Dcmd to Locate Driver Instances  (Continued)

name='lun' type=int items=1
value=00000000
name='target' type=int items=1
value=00000002
name='class_prop' type=string items=1
value='ata'
name='type' type=string items=1
value='ata'
name='class' type=string items=1
value='dada'

Retrieving Driver Soft State Information
A common problem when debugging a driver is retrieving the soft state for a particular driver instance. The soft state is allocated with the ddi_soft_state_zalloc(9F) routine. The driver can obtain the soft state through ddi_get_soft_state(9F). The name of the soft state pointer is the first argument to ddi_soft_state_init(9F). With the name, you can use mdb to retrieve the soft state for a particular driver instance through the ::softstate dcmd:

> *bst_state::softstate 0x3
702b7578

In this case, ::softstate is used to fetch the soft state for instance 3 of the bst sample driver. This pointer references a bst_soft structure that is used by the driver to track state for this instance.

Modifying Kernel Variables
You can use both kmdb and mdb to modify kernel variables or other kernel state. Kernel state modification with mdb should be done with care, because mdb does not stop the kernel before making modifications. Groups of modifications can be made atomically by using kmdb, because kmdb stops the kernel before allowing access by the user. The mdb debugger is capable of making single atomic modifications only.

Be sure to use the proper format specifier to perform the modification. The formats are:

- w – Writes the lowest two bytes of the value of each expression to the target beginning at the location specified by dot
- W – Writes the lowest 4 bytes of the value of each expression to the target beginning at the location specified by dot
- Z – Write the complete 8 bytes of the value of each expression to the target beginning at the location specified by dot

Use the ::sizeof dcmd to determine the size of the variable to be modified.
The following example overwrites the value of moddebug with the value 0x80000000.

EXAMPLE 23–22 Modifying a Kernel Variable With a Debugger

```
> moddebug/W 0x80000000
moddebug:   0 = 0x80000000
```

### Tuning Drivers

The Oracle Solaris OS provides kernel statistics structures so that you can implement counters for your driver. The DTrace facility enables you to analyze performance in real time. This section presents the following topics on device performance:

- "Kernel Statistics" on page 524 – The Oracle Solaris OS provides a set of data structures and functions for capturing performance statistics in the kernel. Kernel statistics (called *kstats*) enable your driver to export continuous statistics while the system is running. The *kstat* data is handled programmatically by using the *kstat* functions.
- "DTrace for Dynamic Instrumentation" on page 530 – DTrace enables you to add instrumentation to your driver dynamically so that you can perform tasks like analyzing the system and measuring performance. DTrace takes advantage of predefined *kstat* structures.

### Kernel Statistics

To assist in performance tuning, the Oracle Solaris kernel provides the *kstat(3KSTAT)* facility. The *kstat* facility provides a set of functions and data structures for device drivers and other kernel modules to export module-specific kernel statistics.

A *kstat* is a data structure for recording quantifiable aspects of a device’s usage. A *kstat* is stored as a null-terminated linked list. Each *kstat* has a common header section and a type-specific data section. The header section is defined by the *kstat_t* structure.

### Kernel Statistics Structure Members

The members of a *kstat* structure are:

- **ks_class[KSSTAT_STRLEN]**: Categorizes the *kstat* type as `bus`, `controller`, `device_error`, `disk`, `hat`, `kmem_cache`, `kstat`, `misc`, `net`, `nfs`, `pages`, `partition`, `rps`, `ufs`, `vm`, or `vmem`.
- **ks_crtme**: Time at which the *kstat* was created. *ks_crtme* is commonly used in calculating rates of various counters.
- **ks_data**: Points to the data section for the *kstat*.
- **ks_data_size**: Total size of the data section in bytes.
ks_instance  The instance of the kernel module that created this kstat. ks_instance is combined with ks_module and ks_name to give the kstat a unique, meaningful name.

ks_kid  Unique ID for the kstat.

ks_module[KSTAT_STRLEN]  Identifies the kernel module that created this kstat. ks_module is combined with ks_instance and ks_name to give the kstat a unique, meaningful name. KSTAT_STRLEN sets the maximum length of ks_module.

ks_name[KSTAT_STRLEN]  A name assigned to the kstat in combination with ks_module and ks_instance. KSTAT_STRLEN sets the maximum length of ks_module.

ks_ndata  Indicates the number of data records for those kstat types that support multiple records: KSTAT_TYPE_RAW, KSTAT_TYPE_NAMED, and KSTAT_TYPE_TIMER.

ks_next  Points to next kstat in the chain.

ks_resv  A reserved field.

ks_snaptime  The timestamp for the last data snapshot, useful in calculating rates.

ks_type  The data type, which can be KSTAT_TYPE_RAW for binary data, KSTAT_TYPE_NAMED for name/value pairs, KSTAT_TYPE_INTR for interrupt statistics, KSTAT_TYPE_IO for I/O statistics, and KSTAT_TYPE_TIMER for event timers.

Kernel Statistics Structures
The structures for the different kinds of kstats are:

kstat(9S)  Each kernel statistic (kstat) that is exported by device drivers consists of a header section and a data section. The kstat(9S) structure is the header portion of the statistic.

kstat_intr(9S)  Structure for interrupt kstats. The types of interrupts are:
- Hard interrupt – Sourced from the hardware device itself
- Soft interrupt – Induced by the system through the use of some system interrupt source
- Watchdog interrupt – Induced by a periodic timer call
- Spurious interrupt – An interrupt entry point was entered but there was no interrupt to service
Multiple service – An interrupt was detected and serviced just prior to returning from any of the other types.

Drivers generally report only claimed hard interrupts and soft interrupts from their handlers, but measurement of the spurious class of interrupts is useful for auto-vectored devices to locate any interrupt latency problems in a particular system configuration. Devices that have more than one interrupt of the same type should use multiple structures.

- **kstat_io(9S)** Structure for I/O kstats.
- **kstat_named(9S)** Structure for named kstats. A named kstat is an array of name-value pairs. These pairs are kept in the kstat_named structure.

**Kernel Statistics Functions**

The functions for using kstats are:

- **kstat_create(9F)**
  Allocate and initialize a kstat(9S) structure.

- **kstat_delete(9F)**
  Remove a kstat from the system.

- **kstat_install(9F)**
  Add a fully initialized kstat to the system.

- **kstat_named_init(9F), kstat_named_setstr(9F)**
  Initialize a named kstat. kstat_named_setstr() associates str, a string, with the named kstat pointer.

- **kstat_queue(9F)**
  A large number of I/O subsystems have at least two basic queues of transactions to be managed. One queue is for transactions that have been accepted for processing but for which processing has yet to begin. The other queue is for transactions that are actively being processed but not yet done. For this reason, two cumulative time statistics are kept: wait time and run time. Wait time is prior to service. Run time is during the service. The kstat_queue() family of functions manages these times based on the transitions between the driver wait queue and run queue:

  - **kstat_runq_back_to_waitq(9F)**
  - **kstat_runq_enter(9F)**
  - **kstat_runq_exit(9F)**
  - **kstat_waitq_enter(9F)**
  - **kstat_waitq_exit(9F)**
  - **kstat_waitq_to_runq(9F)**
Kernel Statistics for Oracle Solaris Ethernet Drivers

The kstat interface described in the following table is an effective way to obtain Ethernet physical layer statistics from the driver. Ethernet drivers should export these statistics to guide users in better diagnosis and repair of Ethernet physical layer problems. With exception of link_up, all statistics have a default value of 0 when not present. The value of the link_up statistic should be assumed to be 1.

The following example gives all the shared link setup. In this case mii is used to filter statistics.

```
kstat ce:0:mii:link_ *
```

<table>
<thead>
<tr>
<th>Kstat Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xcvr_addr</td>
<td>KSTAT_DATA_UINT32</td>
<td>Provides the MII address of the transceiver that is currently in use.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- (0) - (31) are for the MII address of the physical layer device in use for a given Ethernet device.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- (-1) is used where there is no externally accessible MII interface, and therefore the MII address is undefined or irrelevant.</td>
</tr>
<tr>
<td>xcvr_id</td>
<td>KSTAT_DATA_UINT32</td>
<td>Provides the specific vendor ID or device ID of the transceiver that is currently in use.</td>
</tr>
<tr>
<td>xcvr_inuse</td>
<td>KSTAT_DATA_UINT32</td>
<td>Indicates the type of transceiver that is currently in use. The IEEE aPhytType enumerates the following set:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- (0) other undefined</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- (1) no MII interface is present, but no transceiver is connected</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- (2) 10 Mbits/s Clause 7 10 Mbits/s Manchester</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- (3) 100BASE-T4 Clause 23 100 Mbits/s 8B/6T</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- (4) 100BASE-X Clause 24 100 Mbits/s 4B/5B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- (5) 100BASE-T2 Clause 32 100 Mbits/s PAM5X5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- (6) 1000BASE-X Clause 36 1000 Mbits/s 8B/10B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- (7) 1000BASE-T Clause 40 1000 Mbits/s 4D-PAM5</td>
</tr>
</tbody>
</table>

This set is smaller than the set specified by ifMauType, which is defined to include all of the above plus their half duplex/full duplex options. Since this information can be provided by the cap_* statistics, the missing definitions can be derived from the combination of xcvr_inuse and cap_* to provide all the combinations of ifMauType.

| cap_1000fdx    | KSTAT_DATA_CHAR   | Indicates the device is 1 Gbits/s full duplex capable.                      |
| cap_1000hdx    | KSTAT_DATA_CHAR   | Indicates the device is 1 Gbits/s half duplex capable.                      |
| cap_100fdx     | KSTAT_DATA_CHAR   | Indicates the device is 100 Mbits/s full duplex capable.                    |
### Table 23–2 Ethernet MII/GMII Physical Layer Interface Kernel Statistics (Continued)

<table>
<thead>
<tr>
<th>KstatVariable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cap_100hdx</td>
<td>KSTAT_DATA_CHAR</td>
<td>Indicates the device is 100 Mbits/s half duplex capable.</td>
</tr>
<tr>
<td>cap_10fdx</td>
<td>KSTAT_DATA_CHAR</td>
<td>Indicates the device is 10 Mbits/s full duplex capable.</td>
</tr>
<tr>
<td>cap_10hdx</td>
<td>KSTAT_DATA_CHAR</td>
<td>Indicates the device is 10 Mbits/s half duplex capable.</td>
</tr>
<tr>
<td>cap_asmpause</td>
<td>KSTAT_DATA_CHAR</td>
<td>Indicates the device is capable of asymmetric pause Ethernet flow control.</td>
</tr>
<tr>
<td>cap_pause</td>
<td>KSTAT_DATA_CHAR</td>
<td>Indicates the device is capable of symmetric pause Ethernet flow control.</td>
</tr>
<tr>
<td>cap_rem_fault</td>
<td>KSTAT_DATA_CHAR</td>
<td>Indicates the device is capable of remote fault indication.</td>
</tr>
<tr>
<td>cap_autoneg</td>
<td>KSTAT_DATA_CHAR</td>
<td>Indicates the device is capable of auto-negotiation.</td>
</tr>
<tr>
<td>adv_cap_1000fdx</td>
<td>KSTAT_DATA_CHAR</td>
<td>Indicates the device is advertising 1 Gbits/s full duplex capability.</td>
</tr>
<tr>
<td>adv_cap_1000hdx</td>
<td>KSTAT_DATA_CHAR</td>
<td>Indicates the device is advertising 1 Gbits/s half duplex capability.</td>
</tr>
<tr>
<td>adv_cap_100fdx</td>
<td>KSTAT_DATA_CHAR</td>
<td>Indicates the device is advertising 100 Mbits/s full duplex capability.</td>
</tr>
<tr>
<td>adv_cap_100hdx</td>
<td>KSTAT_DATA_CHAR</td>
<td>Indicates the device is advertising 100 Mbits/s half duplex capability.</td>
</tr>
<tr>
<td>adv_cap_10fdx</td>
<td>KSTAT_DATA_CHAR</td>
<td>Indicates the device is advertising 10 Mbits/s full duplex capability.</td>
</tr>
<tr>
<td>adv_cap_10hdx</td>
<td>KSTAT_DATA_CHAR</td>
<td>Indicates the device is advertising 10 Mbits/s half duplex capability.</td>
</tr>
<tr>
<td>adv_cap_asmpause</td>
<td>KSTAT_DATA_CHAR</td>
<td>Indicates the device is advertising the capability of asymmetric pause Ethernet flow control.</td>
</tr>
<tr>
<td>adv_cap_pause</td>
<td>KSTAT_DATA_CHAR</td>
<td>Indicates the device is advertising the capability of symmetric pause Ethernet flow control when adv_cap_pause is set to 1 and adv_cap_asmpause is set to 0. When adv_cap_asmpause is set to 1, adv_cap_pause has the following meaning:</td>
</tr>
<tr>
<td>adv_rem_fault</td>
<td>KSTAT_DATA_CHAR</td>
<td>Indicates the device is experiencing a fault that it is going to forward to the link partner.</td>
</tr>
<tr>
<td>adv_cap_autoneg</td>
<td>KSTAT_DATA_CHAR</td>
<td>Indicates the device is advertising the capability of auto-negotiation.</td>
</tr>
<tr>
<td>lp_cap_1000fdx</td>
<td>KSTAT_DATA_CHAR</td>
<td>Indicates the device is advertising the capability of 1 Gbits/s full duplex capability.</td>
</tr>
</tbody>
</table>

---

Tuning Drivers

Writing Device Drivers • October 2012
<table>
<thead>
<tr>
<th>Kstat Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lp_cap_1000hdx</td>
<td>KSTAT_DATA_CHAR</td>
<td>Indicates the link partner device is 1 Gbits/s half duplex capable.</td>
</tr>
<tr>
<td>lp_cap_100fdx</td>
<td>KSTAT_DATA_CHAR</td>
<td>Indicates the link partner device is 100 Mbits/s full duplex capable.</td>
</tr>
<tr>
<td>lp_cap_100hdx</td>
<td>KSTAT_DATA_CHAR</td>
<td>Indicates the link partner device is 100 Mbits/s half duplex capable.</td>
</tr>
<tr>
<td>lp_cap_10fdx</td>
<td>KSTAT_DATA_CHAR</td>
<td>Indicates the link partner device is 10 Mbits/s full duplex capable.</td>
</tr>
<tr>
<td>lp_cap_10hdx</td>
<td>KSTAT_DATA_CHAR</td>
<td>Indicates the link partner device is 10 Mbits/s half duplex capable.</td>
</tr>
<tr>
<td>lp_cap_asmpause</td>
<td>KSTAT_DATA_CHAR</td>
<td>Indicates the link partner device is capable of asymmetric pause Ethernet flow control.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Indicates the link partner device is capable of symmetric pause Ethernet flow control when lp_cap_pause is set to 1 and lp_cap_asmpause is set to 0. When lp_cap_asmpause is set to 1 and lp_cap_pause is set to 1, lp_cap_pause has the following meaning:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>□ lp_cap_pause = 0 Link partner will transmit pauses based on receive congestion.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>□ lp_cap_pause = 1 Link partner will receive pauses and slow down transmit to avoid congestion.</td>
</tr>
<tr>
<td>lp_rem_fault</td>
<td>KSTAT_DATA_CHAR</td>
<td>Indicates the link partner is experiencing a fault with the link.</td>
</tr>
<tr>
<td>lp_cap_autoneg</td>
<td>KSTAT_DATA_CHAR</td>
<td>Indicates the link partner device is capable of auto-negotiation.</td>
</tr>
<tr>
<td>link_asmpause</td>
<td>KSTAT_DATA_CHAR</td>
<td>Indicates the link is operating with asymmetric pause Ethernet flow control.</td>
</tr>
<tr>
<td>link_pause</td>
<td>KSTAT_DATA_CHAR</td>
<td>Indicates the resolution of the pause capability. Indicates the link is operating with symmetric pause Ethernet flow control when link_pause is set to 1 and link_asmpause is set to 0. When link_asmpause is set to 1 and is relative to a local view of the link, link_pause has the following meaning:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>□ link_pause = 0 This station will transmit pauses based on receive congestion.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>□ link_pause = 1 This station will receive pauses and slow down transmit to avoid congestion.</td>
</tr>
<tr>
<td>link_duplex</td>
<td>KSTAT_DATA_CHAR</td>
<td>Indicates the link duplex.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>□ link_duplex = 0 Link is down and duplex is unknown.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>□ link_duplex = 1 Link is up and in half duplex mode.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>□ link_duplex = 2 Link is up and in full duplex mode.</td>
</tr>
<tr>
<td>link_up</td>
<td>KSTAT_DATA_CHAR</td>
<td>Indicates whether the link is up or down.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>□ link_up = 0 Link is down.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>□ link_up = 1 Link is up.</td>
</tr>
</tbody>
</table>
DTrace for Dynamic Instrumentation

DTrace is a comprehensive dynamic tracing facility for examining the behavior of both user programs and the operating system itself. With DTrace, you can collect data at strategic locations in your environment, referred to as probes. DTrace enables you to record such data as stack traces, timestamps, the arguments to a function, or simply counts of how often the probe fires. Because DTrace enables you to insert probes dynamically, you do not need to recompile your code. For more information on DTrace, see the Oracle Solaris 11.1 Dynamic Tracing Guide.
Recommended Coding Practices

This chapter describes how to write drivers that are robust. Drivers that are written in accordance with the guidelines that are discussed in this chapter are easier to debug. The recommended practices also protect the system from hardware and software faults.

This chapter provides information on the following subjects:
- “Debugging Preparation Techniques” on page 531
- “Declaring a Variable Volatile” on page 534
- “Serviceability” on page 536

Debugging Preparation Techniques

Driver code is more difficult to debug than user programs because:
- The driver interacts directly with the hardware
- The driver operates without the protection of the operating system that is available to user processes

Be sure to build debugging support into your driver. This support facilitates both maintenance work and future development.

Use a Unique Prefix to Avoid Kernel Symbol Collisions

The name of each function, data element, and driver preprocessor definition must be unique for each driver.

A driver module is linked into the kernel. The name of each symbol unique to a particular driver must not collide with other kernel symbols. To avoid such collisions, each function and data element for a particular driver must be named with a prefix common to that driver. The
prefix must be sufficient to uniquely name each driver symbol. Typically, this prefix is the name of the driver or an abbreviation for the name of the driver. For example, \texttt{xx\_open()} would be the name of the \texttt{open(9E)} routine of driver \texttt{xx}.

When building a driver, a driver must necessarily include a number of system header files. The globally-visible names within these header files cannot be predicted. To avoid collisions with these names, each driver preprocessor definition must be given a unique name by using an identifying prefix.

A distinguishing driver symbol prefix also is an aid to deciphering system logs and panics when troubleshooting. Instead of seeing an error related to an ambiguous \texttt{attach()} function, you see an error message about \texttt{xx\_attach()}.

**Use \texttt{cmn\_err()} to Log Driver Activity**

Use the \texttt{cmn\_err(9F)} function to print messages to a system log from within the device driver. The \texttt{cmn\_err(9F)} function for kernel modules is similar to the \texttt{printf(3C)} function for applications. The \texttt{cmn\_err(9F)} function provides additional format characters, such as the \texttt{%b} format to print device register bits. The \texttt{cmn\_err(9F)} function writes messages to a system log. Use the \texttt{tail(1)} command to monitor these messages on \texttt{/var/adm/messages}.

```bash
% tail -f /var/adm/messages
```

**Use \texttt{ASSERT()} to Catch Invalid Assumptions**

Assertions are an extremely valuable form of active documentation. The syntax for \texttt{ASSERT(9F)} is as follows:

```c
void ASSERT(EXPRESSION)
```

The \texttt{ASSERT()} macro halts the execution of the kernel if a condition that is expected to be true is actually false. \texttt{ASSERT()} provides a way for the programmer to validate the assumptions made by a piece of code.

The \texttt{ASSERT()} macro is defined only when the \texttt{DEBUG} compilation symbol is defined. When \texttt{DEBUG} is not defined, the \texttt{ASSERT()} macro has no effect.

The following example assertion tests the assumption that a particular pointer value is not \texttt{NULL}:

```
ASSERT(ptr \! = NULL);
```

If the driver has been compiled with \texttt{DEBUG}, and if the value of \texttt{ptr} is \texttt{NULL} at this point in execution, then the following panic message is printed to the console:

```
panic: assertion failed: ptr \! = NULL, file: driver.c, line: 56
```
Note – Because ASSERT(9F) uses the DEBUG compilation symbol, any conditional debugging code should also use DEBUG.

Use mutex_owned() to Validate and Document Locking Requirements

The syntax for mutex_owned(9F) is as follows:

```c
int mutex_owned(kmutex_t *mp);
```

A significant portion of driver development involves properly handling multiple threads. Comments should always be used when a mutex is acquired. Comments can be even more useful when an apparently necessary mutex is not acquired. To determine whether a mutex is held by a thread, use mutex_owned() within ASSERT(9F):

```c
void helper(void)
{
    /* this routine should always be called with xsp's mutex held */
    ASSERT(mutex_owned(&xsp->mu));
    /* ... */
}
```

Note – mutex_owned() is only valid within ASSERT() macros. You should use mutex_owned() to control the behavior of a driver.

Use Conditional Compilation to Toggle Costly Debugging Features

You can insert code for debugging into a driver through conditional compiles by using a preprocessor symbol such as DEBUG or by using a global variable. With conditional compilation, unnecessary code can be removed in the production driver. Use a variable to set the amount of debugging output at runtime. The output can be specified by setting a debugging level at runtime with an ioctl or through a debugger. Commonly, these two methods are combined.

The following example relies on the compiler to remove unreachable code, in this case, the code following the always-false test of zero. The example also provides a local variable that can be set in /etc/system or patched by a debugger.

```c
#ifdef DEBUG
    /* comments on values of xxdebug and what they do */
    static int xxdebug;
    #define dcmn_err if (xxdebug) cmn_err
#endif
```
Declaring a Variable Volatile

volatile is a keyword that must be applied when declaring any variable that will reference a device register. Without the use of volatile, the compile-time optimizer can inadvertently delete important accesses. Neglecting to use volatile might result in bugs that are difficult to track down.

The correct use of volatile is necessary to prevent elusive bugs. The volatile keyword instructs the compiler to use exact semantics for the declared objects, in particular, not to remove or reorder accesses to the object. Two instances where device drivers must use the volatile qualifier are:

- When data refers to an external hardware device register, that is, memory that has side effects other than just storage. Note, however, that if the DDI data access functions are used to access device registers, you do not have to use volatile.
- When data refers to global memory that is accessible by more than one thread, that is not protected by locks, and that relies on the sequencing of memory accesses. Using volatile consumes fewer resources than using lock.
The following example uses volatile. A busy flag is used to prevent a thread from continuing while the device is busy and the flag is not protected by a lock:

```c
while (busy) {
    /* do something else */
}
```

The testing thread will continue when another thread turns off the busy flag:

```c
busy = 0;
```

Because busy is accessed frequently in the testing thread, the compiler can potentially optimize the test by placing the value of busy in a register and test the contents of the register without reading the value of busy in memory before every test. The testing thread would never see busy change and the other thread would only change the value of busy in memory, resulting in deadlock. Declaring the busy flag as volatile forces its value to be read before each test.

**Note** – An alternative to the busy flag is to use a condition variable. See “Condition Variables in Thread Synchronization” on page 70.

When using the volatile qualifier, avoid the risk of accidental omission. For example, the following code

```c
struct device_reg {
    volatile uint8_t csr;
    volatile uint8_t data;
};
struct device_reg *regp;
```

is preferable to the next example:

```c
struct device_reg {
    uint8_t csr;
    uint8_t data;
};
volatile struct device_reg *regp;
```

Although the two examples are functionally equivalent, the second one requires the writer to ensure that volatile is used in every declaration of type `struct device_reg`. The first example results in the data being treated as volatile in all declarations and is therefore preferred. As mentioned above, using the DDI data access functions to access device registers makes qualifying variables as volatile unnecessary.

If you are using Oracle Solaris Studio 12.2 with C++ 5.11, use `-xvector=no` to avoid generating MMX instructions.
Serviceability

To ensure serviceability, the driver must be enabled to take the following actions:

- Detect faulty devices and report the fault
- Remove a device as supported by the Oracle Solaris hot-plug model
- Add a new device as supported by the Oracle Solaris hot-plug model
- Perform periodic health checks to enable the detection of latent faults

Periodic Health Checks

A latent fault is one that does not show itself until some other action occurs. For example, a hardware failure occurring in a device that is a cold standby could remain undetected until a fault occurs on the master device. At this point, the system now contains two defective devices and might be unable to continue operation.

Latent faults that remain undetected typically cause system failure eventually. Without latent fault checking, the overall availability of a redundant system is jeopardized. To avoid this situation, a device driver must detect latent faults and report them in the same way as other faults.

You should provide the driver with a mechanism for making periodic health checks on the device. In a fault-tolerant situation where the device can be the secondary or failover device, early detection of a failed secondary device is essential to ensure that the secondary device can be repaired or replaced before any failure in the primary device occurs.

Periodic health checks can be used to perform the following activities:

- Check any register or memory location on the device whose value might have been altered since the last poll.
  Features of a device that typically exhibit deterministic behavior include heartbeat semaphores, device timers (for example, local \_b0\_t used by download), and event counters. Reading an updated predictable value from the device gives a degree of confidence that things are proceeding satisfactorily.
- Timestamp outgoing requests such as transmit blocks or commands that are issued by the driver.
  The periodic health check can look for any suspect requests that have not completed.
- Initiate an action on the device that should be completed before the next scheduled check.
  If this action is an interrupt, this check is an ideal way to ensure that the device’s circuitry can deliver an interrupt.
The appendixes provide the following background material:

- **Appendix A, “Hardware Overview,”** discusses multiplatform hardware issues for device drivers.
- **Appendix B, “Summary of Oracle Solaris DDI/DKI Services,”** provides tables of kernel functions for device drivers. Deprecated functions are indicated as well.
- **Appendix C, “Making a Device Driver 64-Bit Ready,”** provides guidelines for updating a device driver to run in a 64-bit environment.
- **Appendix D, “Console Frame Buffer Drivers,”** describes how to add the necessary interfaces to a frame buffer driver to enable the driver to interact with the Oracle Solaris kernel terminal emulator.
- **Appendix E, “pci.conf File,”** describes the `pci.conf(4)` configuration file.
This appendix discusses general issues about hardware that is capable of supporting the Oracle Solaris OS. The discussion includes the processor, bus architectures, and memory models that are supported by the operating system. Various device issues and the PROM used in Oracle platforms are also covered.

**Note** – The material in this appendix is for informational purposes only. This information might be of use during driver debugging. However, many of these implementation details are hidden from device drivers by the Oracle Solaris DDI/DKI interfaces.

This appendix provides information on the following subjects:

- "SPARC Processor Issues" on page 539
- “x86 Processor Issues” on page 541
- “Endianness” on page 542
- “Store Buffers” on page 543
- “System Memory Model” on page 543
- “Bus Architectures” on page 544
- “Bus Specifics” on page 545
- “Device Issues” on page 550
- "PROM on SPARC Machines” on page 552

## SPARC Processor Issues

This section describes a number of SPARC processor-specific topics such as data alignment, byte ordering, register windows, and availability of floating-point instructions. For information on x86 processor-specific topics, see “x86 Processor Issues” on page 541.
**Note** – Drivers should never perform floating-point operations, because these operations are not supported in the kernel.

**SPARC Data Alignment**

All quantities must be aligned on their natural boundaries, using standard C data types:

- `short` integers are aligned on 16-bit boundaries.
- `int` integers are aligned on 32-bit boundaries.
- `long` integers are aligned on 64-bit boundaries for SPARC systems. For information on data models, see Appendix C, “Making a Device Driver 64-Bit Ready.”
- `long long` integers are aligned on 64-bit boundaries.

Usually, the compiler handles any alignment issues. However, driver writers are more likely to be concerned about alignment because the proper data types must be used to access the devices. Because device registers are commonly accessed through a pointer reference, drivers must ensure that pointers are properly aligned when accessing the device.

**Member Alignment in SPARC Structures**

Because of the data alignment restrictions imposed by the SPARC processor, C structures also have alignment requirements. Structure alignment requirements are imposed by the most strictly aligned structure component. For example, a structure containing only characters has no alignment restrictions, while a structure containing a `long long` member must be constructed to guarantee that this member falls on a 64-bit boundary.

**SPARC Byte Ordering**

The SPARC processor uses big-endian byte ordering. The most significant byte (MSB) of an integer is stored at the lowest address of the integer. The least significant byte is stored at the highest address for words in this processor. For example, byte 63 is the least significant byte for 64-bit processors.
SPARC Register Windows

SPARC processors use register windows. Each register window consists of eight in registers, eight local registers, eight out registers, and eight global registers. Out registers are the in registers for the next window. The number of register windows ranges from 2 to 32, depending on the processor implementation.

Because drivers are normally written in C, the compiler usually hides the fact that register windows are used. However, you might have to use register windows when debugging the driver.

SPARC Multiply and Divide Instructions

The Version 7 SPARC processors do not have multiply or divide instructions. The multiply and divide instructions are emulated in software. Because a driver might run on a Version 7, Version 8, or Version 9 processor, avoid intensive integer multiplication and division. Instead, use bitwise left and right shifts to multiply and divide by powers of two.


x86 Processor Issues

Data types have no alignment restrictions. However, extra memory cycles might be required for the x86 processor to properly handle misaligned data transfers.

Note – Drivers should not perform floating-point operations, as these operations are not supported in the kernel.

x86 Byte Ordering

The x86 processors use little-endian byte ordering. The least significant byte (LSB) of an integer is stored at the lowest address of the integer. The most significant byte is stored at the highest address for data items in this processor. For example, byte 7 is the most significant byte for 64-bit processors.
Endianness

To achieve the goal of multiple-platform, multiple-instruction-set architecture portability, host bus dependencies were removed from the drivers. The first dependency issue to be addressed was the endianness, that is, byte ordering, of the processor. For example, the x86 processor family is little-endian while the SPARC architecture is big-endian.

Bus architectures display the same endianness types as processors. The PCI local bus, for example, is little-endian, the SBus is big-endian, the ISA bus is little-endian, and so on.

To maintain portability between processors and buses, DDI-compliant drivers must be endian neutral. Although drivers can manage their endianness by runtime checks or by preprocessor directives like #ifdef _LITTLE_ENDIAN in the source code, long-term maintenance can be troublesome. In some cases, the DDI framework performs the byte swapping using a software approach. In other cases, byte swapping can be done by hardware page-level swapping as in memory management unit (MMU) or by special machine instructions. The DDI framework can take advantage of the hardware features to improve performance.

Along with being endian-neutral, portable drivers must also be independent from data ordering of the processor. Under most circumstances, data must be transferred in the sequence instructed by the driver. However, sometimes data can be merged, batched, or reordered to streamline the data transfer, as illustrated in the following figure. For example, data merging can be applied to accelerate graphics display on frame buffers. Drivers have the option to advise the DDI framework to use other optimal data transfer mechanisms during the transfer.
**Store Buffers**

To improve performance, the CPU uses internal store buffers to temporarily store data. Using internal buffers can affect the synchronization of device I/O operations. Therefore, the driver needs to take explicit steps to make sure that writes to registers are completed at the proper time.

For example, consider the case where access to device space, such as registers or a frame buffer, is synchronized by a lock. The driver needs to check that the store to the device space has actually completed before releasing the lock. The release of the lock does not guarantee the flushing of I/O buffers.

To give another example, when acknowledging an interrupt, the driver usually sets or clears a bit in a device control register. The driver must ensure that the write to the control register has reached the device before the interrupt handler returns. Similarly, a device might require a delay, that is, driver busy-waits, after writing a command to the control register. In such a case, the driver must ensure that the write has reached the device before delaying.

Where device registers can be read without undesirable side effects, verification of a write can simply consist of reading the register immediately after the write. If that particular register cannot be read without undesirable side effects, another device register in the same register set can be used.

**System Memory Model**

The system memory model defines the semantics of memory operations such as `load` and `store` and specifies how the order in which these operations are issued by a processor is related to the order in which they reach memory. The memory model applies to both uniprocessors and shared-memory multiprocessors. Two memory models are supported: total store ordering (TSO) and partial store ordering (PSO).
Total Store Ordering (TSO)

TSO guarantees that the sequence in which store, FLUSH, and atomic load-store instructions appear in memory for a given processor is identical to the sequence in which they were issued by the processor.

Both x86 and SPARC processors support TSO.

Partial Store Ordering (PSO)

PSO does not guarantee that the sequence in which store, FLUSH, and atomic load-store instructions appear in memory for a given processor is identical to the sequence in which they were issued by the processor. The processor can reorder the stores so that the sequence of stores to memory is not the same as the sequence of stores issued by the CPU.

SPARC processors support PSO; x86 processors do not.

For SPARC processors, conformance between issuing order and memory order is provided by the system framework using the STBAR instruction. If two of the above instructions are separated by an STBAR instruction in the issuing order of a processor, or if the instructions reference the same location, the memory order of the two instructions is the same as the issuing order. Enforcement of strong data-ordering in DDI-compliant drivers is provided by the ddi_regs_map_setup(9F) interface. Compliant drivers cannot use the STBAR instruction directly.

See the SPARC Architecture Manual, Version 9, for more details on the SPARC memory model.

Bus Architectures

This section describes device identification, device addressing, and interrupts.

Device Identification

Device identification is the process of determining which devices are present in the system. Some devices are self-identifying meaning that the device itself provides information to the system so that the system can identify the device driver that needs to be used. SBus and PCI local bus devices are examples of self-identifying devices. On SBus, the information is usually derived from a small Forth program stored in the FCode PROM on the device. Most PCI devices provide a configuration space containing device configuration information. See the sbus(4) and pci(4) man pages for more information.

All modern bus architectures require devices to be self-identifying.
Supported Interrupt Types

The Oracle Solaris platform supports both polling and vectored interrupts. The Oracle Solaris DDI/DKI interrupt model is the same for both types of interrupts. See Chapter 8, "Interrupt Handlers," for more information about interrupt handling.

Bus Specifics

This section covers addressing and device configuration issues specific to the buses that the Oracle Solaris platform supports.

PCI Local Bus

The PCI local bus is a high-performance bus designed for high-speed data transfer. The PCI bus resides on the system board. This bus is normally used as an interconnect mechanism between highly integrated peripheral components, peripheral add-on boards, and host processor or memory systems. The host processor, main memory, and the PCI bus itself are connected through a PCI host bridge, as shown in Figure A–3.

A tree structure of interconnected I/O buses is supported through a series of PCI bus bridges. Subordinate PCI bus bridges can be extended underneath the PCI host bridge to enable a single bus system to be expanded into a complex system with multiple secondary buses. PCI devices can be connected to one or more of these secondary buses. In addition, other bus bridges, such as SCSI or USB, can be connected.

Every PCI device has a unique vendor ID and device ID. Multiple devices of the same kind are further identified by their unique device numbers on the bus where they reside.
The PCI host bridge provides an interconnect between the processor and peripheral components. Through the PCI host bridge, the processor can directly access main memory independent of other PCI bus masters. For example, while the CPU is fetching data from the cache controller in the host bridge, other PCI devices can also access the system memory through the host bridge. The advantage of this architecture is that this architecture separates the I/O bus from the processor’s host bus.

The PCI host bridge also provides data access mappings between the CPU and peripheral I/O devices. The bridge maps every peripheral device to the host address domain so that the processor can access the device through programmed I/O. On the local bus side, the PCI host bridge maps the system memory to the PCI address domain so that the PCI device can access the host memory as a bus master. Figure A–3 shows the two address domains.

**PCI Address Domain**

The PCI address domain consists of three distinct address spaces: configuration, memory, and I/O space.

**PCI Configuration Address Space**

Configuration space is defined geographically. The location of a peripheral device is determined by its physical location within an interconnected tree of PCI bus bridges. A device is located by its bus number and device (slot) number. Each peripheral device contains a set of well-defined configuration registers in its PCI configuration space. The registers are used not only to identify devices but also to supply device configuration information to the configuration framework. For example, base address registers in the device configuration space must be mapped before a device can respond to data access.
The method for generating configuration cycles is host dependent. In x86 machines, special I/O ports are used. On other platforms, the PCI configuration space can be memory-mapped to certain address locations corresponding to the PCI host bridge in the host address domain. When a device configuration register is accessed by the processor, the request is routed to the PCI host bridge. The bridge then translates the access into proper configuration cycles on the bus.

**PCI Configuration Base Address Registers**

The PCI configuration space consists of up to six 32-bit base address registers for each device. These registers provide both size and data type information. System firmware assigns base addresses in the PCI address domain to these registers.

Each addressable region can be either memory or I/O space. The value contained in bit 0 of the base address register identifies the type. A value of 0 in bit 0 indicates a memory space and a value of 1 indicates an I/O space. The following figure shows two base address registers: one for memory and the other for I/O types.

**PCI Memory Address Space**

PCI supports both 32-bit and 64-bit addresses for memory space. System firmware assigns regions of memory space in the PCI address domain to PCI peripherals. The base address of a region is stored in the base address register of the device’s PCI configuration space. The size of each region must be a power of two, and the assigned base address must be aligned on a boundary equal to the size of the region. Device addresses in memory space are memory-mapped into the host address domain so that data access to any device can be performed by the processor’s native load or store instructions.

**PCI I/O Address Space**

PCI supports 32-bit I/O space. I/O space can be accessed differently on different platforms. Processors with special I/O instructions, like the Intel processor family, access the I/O space with `in` and `out` instructions. Machines without special I/O instructions will map to the address locations corresponding to the PCI host bridge in the host address domain. When the processor
accesses the memory-mapped addresses, an I/O request will be sent to the PCI host bridge, which then translates the addresses into I/O cycles and puts them on the PCI bus. Memory-mapped I/O is performed by the native load/store instructions of the processor.

PCI Hardware Configuration Files

Hardware configuration files should be unnecessary for PCI local bus devices. However, on some occasions drivers for PCI devices need to use hardware configuration files to augment the driver private information. See the `driver.conf(4)` and `pci(4)` man pages for further details.

PCI Express

The standard PCI bus has evolved into PCI Express. PCI Express is the next generation high performance I/O bus for connecting peripheral devices in such applications as desktop, mobile, workstation, server, embedded computing and communication platforms.

PCI Express improves bus performance, reduces overall system cost and takes advantage of new developments in computer design. PCI Express uses a serial, point-to-point type interconnect for communication between two devices. Using switches enables users to connect a large number of devices together in a system. Serial interconnect implies fewer pins per device package, which reduces cost and makes the performance highly scalable.

The PCI Express bus has built-in features to accommodate the following technologies:

- QoS (Quality of Service)
- Hotplugging and hot swap
- Advanced power management
- RAS (Reliability, Available, Serviceable)
- Improved error handling
- MSI interrupts

A PCI Express interconnect that connects two devices together is called a link. A link can either be x1, x2, x4, x8, x12, x16 or x32 bidirectional signal pairs. These signals are called lanes. The bandwidth (x1) of each lane is 500 MB/sec in duplex mode. Although PCI-X and PCI Express have different hardware connections, the two buses are identical from a driver writer's point of view. PCI-X is a shared bus. For example, all the devices on the bus share a single set of data lines and signal lines. PCI-Express is a switched bus, which enables more efficient use of the bandwidth between the devices and the system bus.

For more information on PCI Express, please refer to the following web site:
http://www.pcisig.com/home
**SBus**

Typical SBus systems consist of a motherboard (containing the CPU and SBus interface logic), a number of SBus devices on the motherboard itself, and a number of SBus expansion slots. An SBus can also be connected to other types of buses through an appropriate bus bridge.

The SBus is geographically addressed. Each SBus slot exists at a fixed physical address in the system. An SBus card has a different address, depending on which slot it is plugged into. Moving an SBus device to a new slot causes the system to treat this device as a new device.

The SBus uses polling interrupts. When an SBus device interrupts, the system only knows which of several devices might have issued the interrupt. The system interrupt handler must ask the driver for each device whether that device is responsible for the interrupt.

**SBus Physical Address Space**

The following table shows the physical address space layout of the Sun UltraSPARC 2 computer. A physical address on the UltraSPARC 2 model consists of 41 bits. The 41-bit physical address space is further broken down into multiple 33-bit address spaces identified by PA(40:33).

<table>
<thead>
<tr>
<th>PA(40:33)</th>
<th>33-bit Space</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>0x000000000 - 0x07FFFFFFF</td>
<td>2 Gbytes main memory</td>
</tr>
<tr>
<td>0x80 – 0xDF</td>
<td>Reserved on Ultra 2</td>
<td>Reserved on Ultra 2</td>
</tr>
<tr>
<td>0xE0</td>
<td>Processor 0</td>
<td>Processor 0</td>
</tr>
<tr>
<td>0xE1</td>
<td>Processor 1</td>
<td>Processor 1</td>
</tr>
<tr>
<td>0xE2 – 0xFD</td>
<td>Reserved on Ultra 2</td>
<td>Reserved on Ultra 2</td>
</tr>
<tr>
<td>0xFE</td>
<td>0x000000000 - 0x1FFFFFFF</td>
<td>UPA Slave (FFB)</td>
</tr>
<tr>
<td>0xFF</td>
<td>0x000000000 - 0x1FFFFFFF</td>
<td>System I/O space</td>
</tr>
<tr>
<td></td>
<td>0x100000000 - 0x10FFFFFFF</td>
<td>SBus Slot 0</td>
</tr>
<tr>
<td></td>
<td>0x110000000 - 0x11FFFFFFF</td>
<td>SBus Slot 1</td>
</tr>
<tr>
<td></td>
<td>0x120000000 - 0x12FFFFFFF</td>
<td>SBus Slot 2</td>
</tr>
<tr>
<td></td>
<td>0x130000000 - 0x13FFFFFFF</td>
<td>SBus Slot 3</td>
</tr>
<tr>
<td></td>
<td>0x1D0000000 - 0x1DFFFFFFF</td>
<td>SBus Slot D</td>
</tr>
<tr>
<td></td>
<td>0x1E0000000 - 0x1EFFFFFFF</td>
<td>SBus Slot E</td>
</tr>
<tr>
<td></td>
<td>0x1F0000000 - 0x1FFFFFFF</td>
<td>SBus Slot F</td>
</tr>
</tbody>
</table>
Physical SBus Addresses

The SBus has 32 address bits, as described in the SBus Specification. The following table describes how the Ultra 2 uses the address bits.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 27</td>
<td>These bits are the SBus address lines used by an SBus card to address the contents of the card.</td>
</tr>
<tr>
<td>28 - 31</td>
<td>Used by the CPU to select one of the SBus slots. These bits generate the SlaveSelect lines.</td>
</tr>
</tbody>
</table>

This addressing scheme yields the Ultra 2 addresses shown in Table A-1. Other implementations might use a different number of address bits.

The Ultra 2 has seven SBus slots, four of which are physical. Slots 0 through 3 are available for SBus cards. Slots 4-12 are reserved. The slots are used as follows:

- Slots 0-3 are physical slots that have DMA-master capability.
- Slots D, E, and F are not actual physical slots, but refer to the onboard direct memory access (DMA), SCSI, Ethernet, and audio controllers. For convenience, these classes of devices are viewed as being plugged into slots D, E, and F.

Note – Some SBus slots are slave-only slots. Drivers that require DMA capability should use ddi_slaveonly(9F) to determine whether their device is in a DMA-capable slot. For an example of this function, see “attach() Entry Point” on page 104.

SBus Hardware Configuration Files

Hardware configuration files are normally unnecessary for SBus devices. However, on some occasions, drivers for SBus devices need to use hardware configuration files to augment the information provided by the SBus card. See the driver.conf(4) and sbus(4) man page for further details.

Device Issues

This section describes issues with special devices.
Timing-Critical Sections

While most driver operations can be performed without mechanisms for synchronization and protection beyond those provided by the locking primitives, some devices require that a sequence of events occur in order without interruption. In conjunction with the locking primitives, the function `ddi_enter_critical(9F)` asks the system to guarantee, to the best of its ability, that the current thread will neither be preempted nor interrupted. This guarantee stays in effect until a closing call to `ddi_exit_critical(9F)` is made. See the `ddi_enter_critical(9F)` man page for details.

Delays

Many chips specify that they can be accessed only at specified intervals. For example, the Zilog Z8530 SCC has a "write recovery time" of 1.6 microseconds. This specification means that a delay must be enforced with `drv_usecwait(9F)` when writing characters with an 8530. In some instances, the specifications do not make explicit what delays are needed, so the delays must be determined empirically.

Be careful not to compound delays for parts of devices that might exist in large numbers, for example, thousands of SCSI disk drives.

Internal Sequencing Logic

Devices with internal sequencing logic map multiple internal registers to the same external address. The various kinds of internal sequencing logic include the following types:

- The Intel 8251A and the Signetics 2651 alternate the same external register between two internal mode registers. Writing to the first internal register is accomplished by writing to the external register. This write, however, has the side effect of setting up the sequencing logic in the chip so that the next read/write operation refers to the second internal register.

- The NEC PD7201 PCC has multiple internal data registers. To write a byte into a particular register, two steps must be performed. The first step is to write into register zero the number of the register into which the following byte of data will go. The data is then written to the specified data register. The sequencing logic automatically sets up the chip so that the next byte sent will go into data register zero.

- The AMD 9513 timer has a data pointer register that points at the data register into which a data byte will go. When sending a byte to the data register, the pointer is incremented. The current value of the pointer register cannot be read.

Interrupt Issues

Note the following common interrupt-related issues:
- A controller interrupt does not necessarily indicate that both the controller and one of its slave devices are ready. For some controllers, an interrupt can indicate that either the controller is ready or one of its devices is ready but not both.
- Not all devices power up with interrupts disabled and can begin interrupting at any time.
- Some devices do not provide a way to determine that the board has generated an interrupt.
- Not all interrupting boards shut off interrupts when told to do so or after a bus reset.

**PROM on SPARC Machines**

Some platforms have a PROM monitor that provides support for debugging a device without an operating system. This section describes how to use the PROM on SPARC machines to map device registers so that they can be accessed. Usually, the device can be exercised enough with PROM commands to determine whether the device is working correctly.

See the `boot(1M)` man page for a description of the x86 boot subsystem.

The PROM has several purposes, including:

- Bringing the machine up from power on, or from a hard reset PROM reset command
- Providing an interactive tool for examining and setting memory, device registers, and memory mappings
- Booting the Oracle Solaris system.
  
  Simply powering up the computer and attempting to use its PROM to examine device registers can fail. While the device might be correctly installed, those mappings are specific to the Oracle Solaris OS and do not become active until the Oracle Solaris kernel is booted. Upon power up, the PROM maps only essential system devices, such as the keyboard.
- Taking a system crash dump using the `sync` command

**Open Boot PROM 3**

For complete documentation on the Open Boot PROM, see the *Open Boot PROM Toolkit User’s Guide* and the `monitor(1M)` man page. The examples in this section refer to a Sun4U architecture. Other architectures might require different commands to perform actions.

**Note** – The Open Boot PROM is currently used on Sun machines with an SBus or UPA/PCI. The Open Boot PROM uses an “ok” prompt. On older machines, you might have to type ‘n’ to get the “ok” prompt.

If the PROM is in `secure mode` (the `security-mode` parameter is not set to `none`), the PROM password might be required (set in the `security-password` parameter).
The `printenv` command displays all parameters and their values.

Help is available with the `help` command.

EMACS-style command-line history is available. Use Control-N (next) and Control-P (previous) to traverse the history list.

**Forth Commands**

The Open Boot PROM uses the Forth programming language. Forth is a stack-based language. Arguments must be pushed on the stack before running the correct command (called a *word*), and the result is left on the stack.

To place a number on the stack, type its value.

```
ok 57
ok 68
```

To add the two top values on the stack, use the `+` operator.

```
ok +
```

The result remains on the stack. The stack is shown with the `.s` word.

```
ok .s
bf
```

The default base is hexadecimal. The `hex` and `decimal` *words* can be used to switch bases.

```
ok decimal
ok .s
191
```

See the *Forth User's Guide* for more information.

**Walking the PROMs Device Tree**

The commands `pwd`, `cd`, and `ls` walk the PROM device tree to get to the device. The `cd` command must be used to establish a position in the tree before `pwd` will work. This example is from an Ultra 1 workstation with a cgsix framebuffer on an SBus.

```
ok cd /
```

To see the devices attached to the current node in the tree, use `ls`.

```
ok ls
f005a064 SUNW,UltraSPARC@0,0
f00598b0 sbus@lf,0
f00592dc counter-timer@lf,3c00
f004ec8 virtual-memory
```
The full node name can be used:

```
ok cd sbus@1f,0
ok ls
f006a4e4 cgsix@2,0
f0068194 SUNW,bpp@e,c800000
f0065370 ledma@e,8400010
f00612c espdma@e,8400000
f005a448 SUNW,pll@f,1304000
f005a394 sc@f,1300000
f005a24c zs@f,1000000
f005a174 zs@f,1100000
f005a7c0 eeprom@f,1200000
f0059f8c SUNW,fdtwo@f,1400000
f0059cc4 flashprom@f,0
f0059ec4 auxio@f,1900000
f0059d28 SUNW,CS4231@d,c000000
```

Rather than using the full node name in the previous example, you could also use an abbreviation. The abbreviated command-line entry looks like the following example:

```
ok cd sbus
```

The name is actually device@slot, offset (for SBus devices). The cgsix device is in slot 2 and starts at offset 0. If an SBus device is displayed in this tree, the device has been recognized by the PROM.

The .properties command displays the PROM properties of a device. These properties can be examined to determine which properties the device exports. This information is useful later to ensure that the driver is looking for the correct hardware properties. These properties are the same properties that can be retrieved with ddi_getprop(9F).

```
ok cd cgsix
ok .properties
character-set ISO8859-1
intr 00000005 00000000
interrupts 00000005
reg 00000002 00000000 01000000
dblbuf 00 00 00 00
vmsize 00 00 00 01
...
```

The reg property defines an array of register description structures containing the following fields:

```c
uint_t bustype; /* cookie for related bus type*/
uint_t addr; /* address of reg relative to bus */
uint_t size; /* size of this register set */
```
For the cgsix example, the address is 0.

Mapping the Device
A device must be mapped into memory to be tested. The PROM can then be used to verify proper operation of the device by using data-transfer commands to transfer bytes, words, and long words. If the device can be operated from the PROM, even in a limited way, the driver should also be able to operate the device.

To set up the device for initial testing, perform the following steps:

1. Determine the SBus slot number the device is in.
   In this example, the cgsix device is located in slot 2.

2. Determine the offset within the physical address space used by the device.
   The offset used is specific to the device. In the cgsix example, the video memory happens to start at an offset of 0x800000.

3. Use the select-dev word to select the SBus device and the map-in word to map the device in.
   The select-dev word takes a string of the device path as its argument. The map-in word takes an offset, a slot number, and a size as arguments to map. Like the offset, the size of the byte transfer is specific to the device. In the cgsix example, the size is set to 0x100000 bytes.

   In the following code example, the SBus path is displayed as an argument to the select-dev word, and the offset, slot number, and size values for the frame buffer are displayed as arguments to the map-in word. Notice the space between the opening quote and / in the select-dev argument. The virtual address to use remains on top of the stack. The stack is shown using the .s word. The stack can be assigned a name with the constant operation.

   ok " sbus@1f,0" select-dev
   ok 800000 2 100000 map-in
   ok .s
   ffe98000
   ok constant fb

Reading and Writing
The PROM provides a variety of 8-bit, 16-bit, and 32-bit operations. In general, a c (character) prefix indicates an 8-bit (one-byte) operation; a w (word) prefix indicates a 16-bit (two-byte) operation; and an L (longword) prefix indicates a 32-bit (four-byte) operation.

A suffix of ! indicates a write operation. The write operation takes the first two items off the stack. The first item is the address, and the second item is the value.

   ok 55 ffe98000 c!

A suffix of @ indicates a read operation. The read operation takes the address off the stack.
A suffix of ? is used to display the value without affecting the stack.

Be careful when trying to query the device. If the mappings are not set up correctly, trying to read or write could cause errors. Special words are provided to handle these cases. cprobe, wprobe, and lprobe, for example, read from the given address but return zero if the location does not respond, or nonzero if it does.

A region of memory can be shown with the dump word. This takes an address and a length, and displays the contents of the memory region in bytes.

In the following example, the fill word is used to fill video memory with a pattern. fill takes the address, the number of bytes to fill, and the byte to use. Use wfill and lfill for words and longwords. This fill example causes the cgsix to display simple patterns based on the byte passed.

```
ok " /sbus" select-dev
ok 000000 2 100000 map-in
ok constant fb
ok fb 10000 ff fill
ok fb 20000 0 fill
ok fb 18000 55 fill
ok fb 15000 3 fill
ok fb 10000 5 fillok fb 5000 f9 fill
```
This appendix discusses the interfaces provided by the Oracle Solaris DDI/DKI. These descriptions should not be considered complete or definitive, nor do they provide a thorough guide to usage. The descriptions are intended to describe what the functions do in general terms. See physio(9F) for more detailed information. The categories are:

- “Module Functions” on page 558
- "Device Information Tree Node (dev_info_t) Functions” on page 558
- "Device (dev_t) Functions” on page 558
- "Property Functions” on page 559
- "Device Software State Functions” on page 560
- "Memory Allocation and Deallocation Functions” on page 560
- "Kernel Thread Control and Synchronization Functions” on page 561
- "Task Queue Management Functions” on page 562
- "Interrupt Functions” on page 563
- "Programmed I/O Functions” on page 565
- "Direct Memory Access (DMA) Functions” on page 571
- "User Space Access Functions” on page 573
- "User Process Event Functions” on page 574
- "User Process Information Functions” on page 574
- "User Application Kernel and Device Access Functions” on page 575
- "Time-Related Functions” on page 576
- "Power Management Functions” on page 577
- "Fault Management Functions” on page 577
- "Kernel Statistics Functions” on page 578
- "Kernel Logging and Printing Functions” on page 579
- "Buffered I/O Functions” on page 579
- "Virtual Memory Functions” on page 580
- "Device ID Functions” on page 580
- "SCSI Functions” on page 581
- "Resource Map Management Functions” on page 583
- “System Global State” on page 583
Module Functions

The module functions are:

- mod_info: Query a loadable module
- mod_install: Add a loadable module
- mod_remove: Remove a loadable module

Device Information TreeNode (dev_info_t) Functions

The device information tree node functions are:

- ddi_binding_name(): Return driver binding name
- ddi_dev_is_sid(): Tell whether a device is self-identifying
- ddi_driver_major(): Return driver major device number
- ddi_driver_name(): Return normalized driver name
- ddi_node_name(): Return the dev_info node name
- ddi_get_devstate(): Check device state
- ddi_get_instance(): Get device instance number
- ddi_get_name(): Return driver binding name
- ddi_get_parent(): Find the parent of a device information structure
- ddi_root_node(): Get the root of the dev_info tree

Device (dev_t) Functions

The device functions are:

- ddi_create_minor_node(): Create a minor node for a device
- ddi_getminor(): Get kernel internal minor number from an external dev_t
- ddi_remove_minor_node(): Remove a minor mode for a device
- getmajor(): Get major device number
- getminor(): Get minor device number
makedevice() Make device number from major and minor numbers

**Property Functions**

The property functions are:

- `ddi_prop_exists()` Check for the existence of a property
- `ddi_prop_free()` Free resources consumed by property lookup
- `ddi_prop_get_int()` Look up integer property
- `ddi_prop_get_int64()` Look up 64-bit integer property
- `ddi_prop_lookup_byte_array()` Look up byte array property
- `ddi_prop_lookup_int_array()` Look up integer array property
- `ddi_prop_lookup_int64_array()` Look up 64-bit integer array property
- `ddi_prop_lookup_string()` Look up string property
- `ddi_prop_lookup_string_array()` Look up string array property
- `ddi_prop_remove()` Remove a property of a device
- `ddi_prop_remove_all()` Remove all properties of a device
- `ddi_prop_undefine()` Hide a property of a device
- `ddi_prop_update_byte_array()` Create or update byte array property
- `ddi_prop_update_int()` Create or update integer property
- `ddi_prop_update_int64()` Create or update 64-bit integer property
- `ddi_prop_update_int_array()` Create or update integer array property
- `ddi_prop_update_int64_array()` Create or update 64-bit integer array property
- `ddi_prop_update_string()` Create or update string property
- `ddi_prop_update_string_array()` Create or update string array property

**TABLE B-1** Deprecated Property Functions

<table>
<thead>
<tr>
<th>Deprecated Functions</th>
<th>Replacements</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ddi_getlongprop()</code></td>
<td>see <code>ddi_prop_lookup()</code></td>
</tr>
<tr>
<td><code>ddi_getlongprop_buf()</code></td>
<td><code>ddi_prop_lookup()</code></td>
</tr>
<tr>
<td><code>ddi_getprop()</code></td>
<td><code>ddi_prop_get_int()</code></td>
</tr>
</tbody>
</table>
### Device Software State Functions

The device software state functions are:

- `ddi_get_driver_private()`: Get the address of the device's private data area
- `ddi_get_soft_state()`: Get pointer to instance soft-state structure
- `ddi_set_driver_private()`: Set the address of the device's private data area
- `ddi_soft_state_fini()`: Destroy driver soft-state structure
- `ddi_soft_state_free()`: Free instance soft-state structure
- `ddi_soft_state_init()`: Initialize driver soft-state structure
- `ddi_soft_state_zalloc()`: Allocate instance soft-state structure

### Memory Allocation and Deallocation Functions

The memory allocation and deallocation functions are:

- `kmem_alloc()`: Allocate kernel memory
- `kmem_free()`: Free kernel memory
- `kmem_zalloc()`: Allocate zero-filled kernel memory

The following functions allocate and free memory intended to be used for DMA. See "Direct Memory Access (DMA) Functions" on page 571.

- `ddi_dma_mem_alloc()`: Allocate memory for DMA transfer
- `ddi_dma_mem_free()`: Free previously allocated DMA memory

The following functions allocate and free memory intended to be exported to user space. See "User Space Access Functions" on page 573.

- `ddi_umem_alloc()`: Allocate page-aligned kernel memory
ddi_umem_free()  Free page-aligned kernel memory

TABLE B-2  Deprecated Memory Allocation and Deallocation Functions

<table>
<thead>
<tr>
<th>Deprecated Function</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_iopb_alloc()</td>
<td>ddi_dma_mem_alloc()</td>
</tr>
<tr>
<td>ddi_iopb_free()</td>
<td>ddi_dma_mem_free()</td>
</tr>
<tr>
<td>ddi_mem_alloc()</td>
<td>ddi_dma_mem_alloc()</td>
</tr>
<tr>
<td>ddi_mem_free()</td>
<td>ddi_dma_mem_free()</td>
</tr>
</tbody>
</table>

Kernel Thread Control and Synchronization Functions

The kernel thread control and synchronization functions are:

- cv_broadcast()      Wake up all waiting threads
- cv_destroy()        Free an allocated condition variable
- cv_init()           Allocate a condition variable
- cv_signal()         Wake up one waiting thread
- cv_timedwait()      Await an event with timeout
- cv_timedwait_sig()  Await an event or signal with timeout
- cv_wait()           Await an event
- cv_wait_sig()       Await an event or signal
- ddi_can_receive_sig()  Determine whether the current thread can receive a signal
- ddi_enter_critical() Enter a critical region of control
- ddi_exit_critical()  Exit a critical region of control
- mutex_destroy()     Destroy mutual exclusion lock
- mutex_enter()       Acquire mutual exclusion lock
- mutex_exit()        Release mutual exclusion lock
- mutex_init()        Initialize mutual exclusion lock
- mutex_owned()       Determine whether current thread is holding mutual exclusion lock
- mutex_tryenter()    Attempt to acquire mutual exclusion lock without waiting
- rw_destroy()        Destroy a readers/writer lock
Task Queue Management Functions

The task queue management functions are listed below. See the taskq(9F) man page for more information about these interfaces.

**Task Queue Management Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rw_downgrade()</code></td>
<td>Downgrade a readers/writer lock holding from writer to reader</td>
</tr>
<tr>
<td><code>rw_enter()</code></td>
<td>Acquire a readers/writer lock</td>
</tr>
<tr>
<td><code>rw_exit()</code></td>
<td>Release a readers/writer lock</td>
</tr>
<tr>
<td><code>rw_init()</code></td>
<td>Initialize a readers/writer lock</td>
</tr>
<tr>
<td><code>rw_read_locked()</code></td>
<td>Determine whether readers/writer lock is held for read or write</td>
</tr>
<tr>
<td><code>rw_tryenter()</code></td>
<td>Attempt to acquire a readers/writer lock without waiting</td>
</tr>
<tr>
<td><code>rw_tryupgrade()</code></td>
<td>Attempt to upgrade readers/writer lock holding from reader to writer</td>
</tr>
<tr>
<td><code>sema_destroy()</code></td>
<td>Destroy a semaphore</td>
</tr>
<tr>
<td><code>sema_init()</code></td>
<td>Initialize a semaphore</td>
</tr>
<tr>
<td><code>sema_p()</code></td>
<td>Decrement semaphore and possibly block</td>
</tr>
<tr>
<td><code>sema_p_sig()</code></td>
<td>Decrement semaphore but do not block if signal is pending</td>
</tr>
<tr>
<td><code>sema_tryp()</code></td>
<td>Attempt to decrement semaphore but do not block</td>
</tr>
<tr>
<td><code>sema_v()</code></td>
<td>Increment semaphore and possibly unblock waiter</td>
</tr>
</tbody>
</table>
Interrupt Functions

The interrupt functions are:

- `ddi_intr_add_handler(9F)` Adds an interrupt handler.
- `ddi_intr_add_softint(9F)` Adds a soft interrupt handler.
- `ddi_intr_alloc(9F)` Allocates system resources and interrupt vectors for the specified type of interrupt.
- `ddi_intr_block_disable(9F)` Disables the specified range of interrupts. For MSI only.
- `ddi_intr_block_enable(9F)` Enables the specified range of interrupts. For MSI only.
- `ddi_intr_clr_mask(9F)` Clears an interrupt mask if the specified interrupt is enabled.
- `ddi_intr_disable(9F)` Disables the specified interrupt.
- `ddi_intr_dup_handler(9F)` Use with MSI-X only. Copies an address and data pair for an allocated interrupt vector to an unused interrupt vector on the same device.
- `ddi_intr_enable(9F)` Enables the specified interrupt.
- `ddi_intr_free(9F)` Releases the system resources and interrupt vectors for a specified interrupt handle.
- `ddi_intr_get_cap(9F)` Returns interrupt capability flags for the specified interrupt.
- `ddi_intr_get_hilevel_pri(9F)` Returns the minimum priority level for a high-level interrupt.
- `ddi_intr_get_navail(9F)` Returns the number of interrupts available for a particular hardware device and given interrupt type.
- `ddi_intr_get_nintrs(9F)` Get the number of interrupts that the device supports for the given interrupt type.
- `ddi_intr_get_pending(9F)` Read the interrupt pending bit if one is supported by either the host bridge or the device.
- `ddi_intr_get_pri(9F)` Returns the current software priority setting for the specified interrupt.
- `ddi_intr_get_softint_pri(9F)` Returns the soft interrupt priority for the specified interrupt.
**ddi_intr_get_supported_types(9F)**
Returns the hardware interrupt types that are supported by both the device and the host.

**ddi_intr_remove_handler(9F)**
Removes the specified interrupt handler.

**ddi_intr_remove_softint(9F)**
Remove the specified soft interrupt handler.

**ddi_intr_set_cap(9F)**
Sets the DDI_INTR_FLAG_LEVEL or DDI_INTR_FLAG_EDGE flag for the specified interrupt.

**ddi_intr_set_mask(9F)**
Sets an interrupt mask if the specified interrupt is enabled.

**ddi_intr_set_pri(9F)**
Sets the interrupt priority level for the specified interrupt.

**ddi_intr_set_softint_pri(9F)**
Changes the relative soft interrupt priority for the specified soft interrupt.

**ddi_intr_trigger_softint(9F)**
Trigger the specified soft interrupt.

To take advantage of the features of the new framework, use the above interfaces. Do not use the deprecated interfaces that are listed in the following table. These deprecated interfaces are retained for compatibility purposes only.

---

**TABLE B-3  Deprecated Interrupt Functions**

<table>
<thead>
<tr>
<th>Deprecated Interrupt Functions</th>
<th>Replacements</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_add_intr(9F)</td>
<td>Three-step process:</td>
</tr>
<tr>
<td></td>
<td>1. ddi_intr_alloc(9F)</td>
</tr>
<tr>
<td></td>
<td>2. ddi_intr_add_handler(9F)</td>
</tr>
<tr>
<td></td>
<td>3. ddi_intr_enable(9F)</td>
</tr>
<tr>
<td>ddi_add_softintr(9F)</td>
<td>ddi_intr_add_softint(9F)</td>
</tr>
<tr>
<td>ddi_dev_nintrs(9F)</td>
<td>ddi_intr_get_nintrs(9F)</td>
</tr>
<tr>
<td>ddi_get_iblock_cookie(9F)</td>
<td>Three-step process:</td>
</tr>
<tr>
<td></td>
<td>1. ddi_intr_alloc(9F)</td>
</tr>
<tr>
<td></td>
<td>2. ddi_intr_get_pri(9F)</td>
</tr>
<tr>
<td></td>
<td>3. ddi_intr_free(9F)</td>
</tr>
<tr>
<td>ddi_get_soft_iblock_cookie(9F)</td>
<td>Three-step process:</td>
</tr>
<tr>
<td></td>
<td>1. ddi_intr_add_softint(9F)</td>
</tr>
<tr>
<td></td>
<td>2. ddi_intr_get_softint_pri(9F)</td>
</tr>
<tr>
<td></td>
<td>3. ddi_intr_remove_softint(9F)</td>
</tr>
</tbody>
</table>
### Programmed I/O Functions

The programmed I/O functions are:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_dev_nregs()</td>
<td>Return the number of register sets a device has</td>
</tr>
<tr>
<td>ddi_dev_regs()</td>
<td>Return the size of a device’s register</td>
</tr>
<tr>
<td>ddi_regs_map_setup()</td>
<td>Set up a mapping for a register address space</td>
</tr>
<tr>
<td>ddi_regs_map_free()</td>
<td>Free a previously mapped register address space</td>
</tr>
<tr>
<td>ddi_device_copy()</td>
<td>Copy data from one device register to another device register</td>
</tr>
<tr>
<td>ddi_device_zero()</td>
<td>Zero fill the device</td>
</tr>
<tr>
<td>ddi_check_acc_handle()</td>
<td>Check data access handle</td>
</tr>
<tr>
<td>ddi_get8()</td>
<td>Read 8-bit data from mapped memory, device register, or DMA memory</td>
</tr>
<tr>
<td>ddi_get16()</td>
<td>Read 16-bit data from mapped memory, device register, or DMA memory</td>
</tr>
<tr>
<td>ddi_get32()</td>
<td>Read 32-bit data from mapped memory, device register, or DMA memory</td>
</tr>
<tr>
<td>ddi_get64()</td>
<td>Read 64-bit data from mapped memory, device register, or DMA memory</td>
</tr>
<tr>
<td>ddi_put8()</td>
<td>Write 8-bit data to mapped memory, device register, or DMA memory</td>
</tr>
<tr>
<td>ddi_put16()</td>
<td>Write 16-bit data to mapped memory, device register, or DMA memory</td>
</tr>
</tbody>
</table>

### Deprecated Interrupt Functions

<table>
<thead>
<tr>
<th>Deprecated Interrupt Functions</th>
<th>Replacements</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_intr_hilevel(9F)</td>
<td>Three-step process:</td>
</tr>
<tr>
<td></td>
<td>1. ddi_intr_alloc(9F)</td>
</tr>
<tr>
<td></td>
<td>2. ddi_intr_get_hilevel_pri(9F)</td>
</tr>
<tr>
<td></td>
<td>3. ddi_intr_free(9F)</td>
</tr>
<tr>
<td>ddi_remove_intr(9F)</td>
<td>Three-step process:</td>
</tr>
<tr>
<td></td>
<td>1. ddi_intr_disable(9F)</td>
</tr>
<tr>
<td></td>
<td>2. ddi_intr_remove_handler(9F)</td>
</tr>
<tr>
<td></td>
<td>3. ddi_intr_free(9F)</td>
</tr>
<tr>
<td>ddi_remove_softintr(9F)</td>
<td>ddi_intr_remove_softint(9F)</td>
</tr>
<tr>
<td>ddi_trigger_softintr(9F)</td>
<td>ddi_intr_trigger_softint(9F)</td>
</tr>
</tbody>
</table>
ddi_put32()  Write 32-bit data to mapped memory, device register, or DMA memory

ddi_put64()  Write 64-bit data to mapped memory, device register, or DMA memory

ddi_rep_get8()  Read multiple 8-bit data from mapped memory, device register, or DMA memory

ddi_rep_get16()  Read multiple 16-bit data from mapped memory, device register, or DMA memory

ddi_rep_get32()  Read multiple 32-bit data from mapped memory, device register, or DMA memory

ddi_rep_get64()  Read multiple 64-bit data from mapped memory, device register, or DMA memory

ddi_rep_put8()  Write multiple 8-bit data to mapped memory, device register, or DMA memory

ddi_rep_put16()  Write multiple 16-bit data to mapped memory, device register, or DMA memory

ddi_rep_put32()  Write multiple 32-bit data to mapped memory, device register, or DMA memory

ddi_rep_put64()  Write multiple 64-bit data to mapped memory, device register, or DMA memory

ddi.peek8()  Cautiously read an 8-bit value from a location

ddi.peek16()  Cautiously read a 16-bit value from a location

ddi.peek32()  Cautiously read a 32-bit value from a location

ddi.peek64()  Cautiously read a 64-bit value from a location

ddi.poke8()  Cautiously write an 8-bit value to a location

ddi.poke16()  Cautiously write a 16-bit value to a location

ddi.poke32()  Cautiously write a 32-bit value to a location

ddi.poke64()  Cautiously write a 64-bit value to a location

The general programmed I/O functions listed above can always be used rather than the mem, io, and pci_config functions that follow. However, the following functions can be used as alternatives in cases where the type of access is known at compile time.

ddi_io_get8()  Read 8-bit data from a mapped device register in I/O space

ddi_io_get16()  Read 16-bit data from a mapped device register in I/O space
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_get32()</td>
<td>Read 32-bit data from a mapped device register in I/O space</td>
</tr>
<tr>
<td>ddi_io_put8()</td>
<td>Write 8-bit data to a mapped device register in I/O space</td>
</tr>
<tr>
<td>ddi_io_put16()</td>
<td>Write 16-bit data to a mapped device register in I/O space</td>
</tr>
<tr>
<td>ddi_io_put32()</td>
<td>Write 32-bit data to a mapped device register in I/O space</td>
</tr>
<tr>
<td>ddi_io_rep_get8()</td>
<td>Read multiple 8-bit data from a mapped device register in I/O space</td>
</tr>
<tr>
<td>ddi_io_rep_get16()</td>
<td>Read multiple 16-bit data from a mapped device register in I/O space</td>
</tr>
<tr>
<td>ddi_io_rep_get32()</td>
<td>Read multiple 32-bit data from a mapped device register in I/O space</td>
</tr>
<tr>
<td>ddi_io_rep_put8()</td>
<td>Write multiple 8-bit data to a mapped device register in I/O space</td>
</tr>
<tr>
<td>ddi_io_rep_put16()</td>
<td>Write multiple 16-bit data to a mapped device register in I/O space</td>
</tr>
<tr>
<td>ddi_io_rep_put32()</td>
<td>Write multiple 32-bit data to a mapped device register in I/O space</td>
</tr>
<tr>
<td>ddi_mem_get8()</td>
<td>Read 8-bit data from a mapped device in memory space or DMA memory</td>
</tr>
<tr>
<td>ddi_mem_get16()</td>
<td>Read 16-bit data from a mapped device in memory space or DMA memory</td>
</tr>
<tr>
<td>ddi_mem_get32()</td>
<td>Read 32-bit data from a mapped device in memory space or DMA memory</td>
</tr>
<tr>
<td>ddi_mem_get64()</td>
<td>Read 64-bit data from a mapped device in memory space or DMA memory</td>
</tr>
<tr>
<td>ddi_mem_put8()</td>
<td>Write 8-bit data to a mapped device in memory space or DMA memory</td>
</tr>
<tr>
<td>ddi_mem_put16()</td>
<td>Write 16-bit data to a mapped device in memory space or DMA memory</td>
</tr>
<tr>
<td>ddi_mem_put32()</td>
<td>Write 32-bit data to a mapped device in memory space or DMA memory</td>
</tr>
<tr>
<td>ddi_mem_put64()</td>
<td>Write 64-bit data to a mapped device in memory space or DMA memory</td>
</tr>
<tr>
<td>ddi_mem_rep_get8()</td>
<td>Read multiple 8-bit data from a mapped device in memory space or DMA memory</td>
</tr>
</tbody>
</table>
ddi_mem_rep_get16()  Read multiple 16-bit data from a mapped device in memory space or DMA memory
ddi_mem_rep_get32()  Read multiple 32-bit data from a mapped device in memory space or DMA memory
ddi_mem_rep_get64()  Read multiple 64-bit data from a mapped device in memory space or DMA memory
ddi_mem_rep_put8()   Write multiple 8-bit data to a mapped device in memory space or DMA memory
ddi_mem_rep_put16()  Write multiple 16-bit data to a mapped device in memory space or DMA memory
ddi_mem_rep_put32()  Write multiple 32-bit data to a mapped device in memory space or DMA memory
ddi_mem_rep_put64()  Write multiple 64-bit data to a mapped device in memory space or DMA memory
pci_config_setup()  Set up access to PCI Local Bus Configuration space
pci_config_teardown() Tear down access to PCI Local Bus Configuration space
pci_config_get8()   Read 8-bit data from the PCI Local Bus Configuration space
pci_config_get16()  Read 16-bit data from the PCI Local Bus Configuration space
pci_config_get32()  Read 32-bit data from the PCI Local Bus Configuration space
pci_config_get64()  Read 64-bit data from the PCI Local Bus Configuration space
pci_config_put8()   Write 8-bit data to the PCI Local Bus Configuration space
pci_config_put16()  Write 16-bit data to the PCI Local Bus Configuration space
pci_config_put32()  Write 32-bit data to the PCI Local Bus Configuration space
pci_config_put64()  Write 64-bit data to the PCI Local Bus Configuration space

TABLE B-4   Deprecated Programmed I/O Functions

<table>
<thead>
<tr>
<th>Deprecated Function</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_getb()</td>
<td>ddi_get8()</td>
</tr>
<tr>
<td>ddi_getl()</td>
<td>ddi_get32()</td>
</tr>
<tr>
<td>ddi_getll()</td>
<td>ddi_get64()</td>
</tr>
<tr>
<td>ddi_getw()</td>
<td>ddi_get16()</td>
</tr>
<tr>
<td>ddi_io_getb()</td>
<td>ddi_io_get8()</td>
</tr>
</tbody>
</table>

Writing Device Drivers • October 2012
### Deprecated Programmed I/O Functions (Continued)

<table>
<thead>
<tr>
<th>Deprecated Function</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_io_getl()</td>
<td>ddi_io_get32()</td>
</tr>
<tr>
<td>ddi_io_getw()</td>
<td>ddi_io_get16()</td>
</tr>
<tr>
<td>ddi_io_putb()</td>
<td>ddi_io_put8()</td>
</tr>
<tr>
<td>ddi_io_putl()</td>
<td>ddi_io_put32()</td>
</tr>
<tr>
<td>ddi_io_putw()</td>
<td>ddi_io_put16()</td>
</tr>
<tr>
<td>ddi_io_rep_getb()</td>
<td>ddi_io_rep_get8()</td>
</tr>
<tr>
<td>ddi_io_rep_getl()</td>
<td>ddi_io_rep_get32()</td>
</tr>
<tr>
<td>ddi_io_rep_getw()</td>
<td>ddi_io_rep_get16()</td>
</tr>
<tr>
<td>ddi_io_rep_putb()</td>
<td>ddi_io_rep_put8()</td>
</tr>
<tr>
<td>ddi_io_rep_putl()</td>
<td>ddi_io_rep_put32()</td>
</tr>
<tr>
<td>ddi_io_rep_putw()</td>
<td>ddi_io_rep_put16()</td>
</tr>
<tr>
<td>ddi_map_regs()</td>
<td>ddi_regs_map_setup()</td>
</tr>
<tr>
<td>ddi_mem_getb()</td>
<td>ddi_mem_get8()</td>
</tr>
<tr>
<td>ddi_mem_getl()</td>
<td>ddi_mem_get32()</td>
</tr>
<tr>
<td>ddi_mem_getll()</td>
<td>ddi_mem_get64()</td>
</tr>
<tr>
<td>ddi_mem_getw()</td>
<td>ddi_mem_get16()</td>
</tr>
<tr>
<td>ddi_mem_putb()</td>
<td>ddi_mem_put8()</td>
</tr>
<tr>
<td>ddi_mem_putl()</td>
<td>ddi_mem_put32()</td>
</tr>
<tr>
<td>ddi_mem_putll()</td>
<td>ddi_mem_put64()</td>
</tr>
<tr>
<td>ddi_mem_putw()</td>
<td>ddi_mem_put16()</td>
</tr>
<tr>
<td>ddi_mem_rep_getb()</td>
<td>ddi_mem_rep_get8()</td>
</tr>
<tr>
<td>ddi_mem_rep_getl()</td>
<td>ddi_mem_rep_get32()</td>
</tr>
<tr>
<td>ddi_mem_rep_getll()</td>
<td>ddi_mem_rep_get64()</td>
</tr>
<tr>
<td>ddi_mem_rep_getw()</td>
<td>ddi_mem_rep_get16()</td>
</tr>
<tr>
<td>ddi_mem_rep_putb()</td>
<td>ddi_mem_rep_put8()</td>
</tr>
<tr>
<td>ddi_mem_rep_putl()</td>
<td>ddi_mem_rep_put32()</td>
</tr>
<tr>
<td>ddi_mem_rep_putll()</td>
<td>ddi_mem_rep_put64()</td>
</tr>
<tr>
<td>ddi_mem_rep_putw()</td>
<td>ddi_mem_rep_put16()</td>
</tr>
<tr>
<td>Deprecated Function</td>
<td>Replacement</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>ddi_peekc()</td>
<td>ddi_peek8()</td>
</tr>
<tr>
<td>ddi_peekd()</td>
<td>ddi_peek64()</td>
</tr>
<tr>
<td>ddi_peekl()</td>
<td>ddi_peek32()</td>
</tr>
<tr>
<td>ddi_peeks()</td>
<td>ddi_peek16()</td>
</tr>
<tr>
<td>ddi_pokec()</td>
<td>ddi_poke8()</td>
</tr>
<tr>
<td>ddi_poked()</td>
<td>ddi_poke64()</td>
</tr>
<tr>
<td>ddi_pokel()</td>
<td>ddi_poke32()</td>
</tr>
<tr>
<td>ddi_pokes()</td>
<td>ddi_poke16()</td>
</tr>
<tr>
<td>ddi_putb()</td>
<td>ddi_put8()</td>
</tr>
<tr>
<td>ddi_putl()</td>
<td>ddi_put32()</td>
</tr>
<tr>
<td>ddi_putll()</td>
<td>ddi_put64()</td>
</tr>
<tr>
<td>ddi_putw()</td>
<td>ddi_put16()</td>
</tr>
<tr>
<td>ddi_rep_getb()</td>
<td>ddi_rep_get8()</td>
</tr>
<tr>
<td>ddi_rep_getl()</td>
<td>ddi_rep_get32()</td>
</tr>
<tr>
<td>ddi_rep_getll()</td>
<td>ddi_rep_get64()</td>
</tr>
<tr>
<td>ddi_rep_getw()</td>
<td>ddi_rep_get16()</td>
</tr>
<tr>
<td>ddi_rep_putb()</td>
<td>ddi_rep_put8()</td>
</tr>
<tr>
<td>ddi_rep_putl()</td>
<td>ddi_rep_put32()</td>
</tr>
<tr>
<td>ddi_rep_putll()</td>
<td>ddi_rep_put64()</td>
</tr>
<tr>
<td>ddi_rep_putw()</td>
<td>ddi_rep_put16()</td>
</tr>
<tr>
<td>ddi_unmap_regs()</td>
<td>ddi_regs_map_free()</td>
</tr>
<tr>
<td>inb()</td>
<td>ddi_io_get8()</td>
</tr>
<tr>
<td>inl()</td>
<td>ddi_io_get32()</td>
</tr>
<tr>
<td>inw()</td>
<td>ddi_io_get16()</td>
</tr>
<tr>
<td>outb()</td>
<td>ddi_io_put8()</td>
</tr>
<tr>
<td>outl()</td>
<td>ddi_io_put32()</td>
</tr>
<tr>
<td>outw()</td>
<td>ddi_io_put16()</td>
</tr>
<tr>
<td>pci_config_getb()</td>
<td>pci_config_get8()</td>
</tr>
<tr>
<td>Deprecated Function</td>
<td>Replacement</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>pci_config_getl()</td>
<td>pci_config_get32()</td>
</tr>
<tr>
<td>pci_config_getll()</td>
<td>pci_config_get64()</td>
</tr>
<tr>
<td>pci_config_getw()</td>
<td>pci_config_get16()</td>
</tr>
<tr>
<td>pci_config_putb()</td>
<td>pci_config_put8()</td>
</tr>
<tr>
<td>pci_config_putll()</td>
<td>pci_config_put64()</td>
</tr>
<tr>
<td>pci_config_putw()</td>
<td>pci_config_put16()</td>
</tr>
<tr>
<td>repinsb()</td>
<td>ddi_io_rep_get8()</td>
</tr>
<tr>
<td>repinsd()</td>
<td>ddi_io_rep_get32()</td>
</tr>
<tr>
<td>repinsw()</td>
<td>ddi_io_rep_get16()</td>
</tr>
<tr>
<td>repoutsb()</td>
<td>ddi_io_rep_put8()</td>
</tr>
<tr>
<td>repoutsd()</td>
<td>ddi_io_rep_put32()</td>
</tr>
<tr>
<td>repoutsw()</td>
<td>ddi_io_rep_put16()</td>
</tr>
</tbody>
</table>

### Direct Memory Access (DMA) Functions

The DMA functions are:

- `ddi_dma_alloc_handle()`: Allocate a DMA handle
- `ddi_dma_free_handle()`: Free a DMA handle
- `ddi_dma_mem_alloc()`: Allocate memory for a DMA transfer
- `ddi_dma_mem_free()`: Free previously allocated DMA memory
- `ddi_dma_addr_bind_handle()`: Bind an address to a DMA handle
- `ddi_dma_buf_bind_handle()`: Bind a system buffer to a DMA handle
- `ddi_dma_unbind_handle()`: Unbind the address in a DMA handle
- `ddi_dma_nextcookie()`: Retrieve the subsequent DMA cookie
- `ddi_dma_getwin()`: Activate a new DMA window
- `ddi_dma_numwin()`: Retrieve number of DMA windows
- `ddi_dma_sync()`: Synchronize CPU and I/O views of memory
- `ddi_check_dma_handle()`: Check a DMA handle
Direct Memory Access (DMA) Functions

<table>
<thead>
<tr>
<th>Deprecated Function</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_dma_set_sbus64()</td>
<td>Allow 64-bit transfers on SBus</td>
</tr>
<tr>
<td>ddi_slaveonly()</td>
<td>Report whether a device is installed in a slave access-only location</td>
</tr>
<tr>
<td>ddi_iomin()</td>
<td>Find the minimum alignment and transfer size for DMA</td>
</tr>
<tr>
<td>ddi_dma_burstsizes()</td>
<td>Find out the allowed burst sizes for a DMA mapping</td>
</tr>
<tr>
<td>ddi_dma_devalign()</td>
<td>Find DMA mapping alignment and minimum transfer size</td>
</tr>
<tr>
<td>ddi_dmae_alloc()</td>
<td>Acquire a DMA channel</td>
</tr>
<tr>
<td>ddi_dmae_release()</td>
<td>Release a DMA channel</td>
</tr>
<tr>
<td>ddi_dmae_getattr()</td>
<td>Get the DMA engine attributes</td>
</tr>
<tr>
<td>ddi_dmae_prog()</td>
<td>Program a DMA channel</td>
</tr>
<tr>
<td>ddi_dmae_stop()</td>
<td>Terminate a DMA engine operation</td>
</tr>
<tr>
<td>ddi_dmae_disable()</td>
<td>Disable a DMA channel</td>
</tr>
<tr>
<td>ddi_dmae_enable()</td>
<td>Enable a DMA channel</td>
</tr>
<tr>
<td>ddi_dmae_getcnt()</td>
<td>Get the remaining DMA engine count</td>
</tr>
<tr>
<td>ddi_dmae_1stparty()</td>
<td>Configure the DMA channel cascade mode</td>
</tr>
<tr>
<td>ddi_dma_coff()</td>
<td>Convert a DMA cookie to an offset within a DMA handle</td>
</tr>
</tbody>
</table>

TABLE B-5  Deprecated Direct Memory Access (DMA) Functions
### Deprecated Direct Memory Access (DMA) Functions (Continued)

<table>
<thead>
<tr>
<th>Deprecated Function</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_iopb_alloc()</td>
<td>ddi_dma_mem_alloc()</td>
</tr>
<tr>
<td>ddi_iopb_free()</td>
<td>ddi_dma_mem_free()</td>
</tr>
<tr>
<td>ddi_mem_alloc()</td>
<td>ddi_dma_mem_alloc()</td>
</tr>
<tr>
<td>ddi_mem_free()</td>
<td>ddi_dma_mem_free()</td>
</tr>
<tr>
<td>hat_getkpfnum()</td>
<td>ddi_dma_addr_bind_handle(),</td>
</tr>
<tr>
<td></td>
<td>ddi_dma_buf_bind_handle(),</td>
</tr>
<tr>
<td></td>
<td>ddi_dma_nextcookie()</td>
</tr>
</tbody>
</table>

### User Space Access Functions

The user space access functions are:

- **ddi_copyin()**: Copy data to a driver buffer
- **ddi_copyout()**: Copy data from a driver
- **uiomove()**: Copy kernel data using a uio structure
- **ureadc()**: Add character to a uio structure
- **uwritec()**: Remove a character from a uio structure
- **getminor()**: Get minor device number.
- **ddi_model_convert_from()**: Determine a data model type mismatch
- **IOC_CONVERT_FROM()**: Determine whether there is a need to translate M_IOCTL contents
- **STRUCT_DECL()**: Establish the handle to application data in a possibly differing data model
- **STRUCT_HANDLE()**: Establish the handle to application data in a possibly differing data model
- **STRUCT_INIT()**: Establish the handle to application data in a possibly differing data model
- **STRUCT_SET_HANDLE()**: Establish the handle to application data in a possibly differing data model
- **SIZEOF_PTR()**: Return the size of pointer in specified data model
- **SIZEOF_STRUCT()**: Return the size of a structure in the specified data model
- **STRUCT_SIZE()**: Return the size of a structure in the application data model
- **STRUCT_BUF()**: Return a pointer to the native mode instance of the structure
User Process Event Functions

The user process event functions are:

- pollwakeup() Inform a process that an event has occurred
- proc_ref() Get a handle on a process to signal
- proc_unref() Release a handle on a process to signal
- proc_signal() Send a signal to a process

User Process Information Functions

The user process information functions are:

- ddi_get_cred() Return a pointer to the credential structure of the caller
- drv_priv() Determine process credentials privilege
- ddi_get_pid() Return the process ID
User Application Kernel and Device Access Functions

The user application kernel and device access functions are:

- **ddi_dev_nregs()**
  - Return the number of register sets a device has

- **ddi_dev_regsize()**
  - Return the size of a device’s register

- **ddi_devmap_segmap(), devmap_setup()**
  - Set up a user mapping to device memory using the devmap framework

- **devmap_devmem_setup()**
  - Export device memory to user space

- **devmap_load()**
  - Validate memory address translations

- **devmap_unload()**
  - Invalidate memory address translations

- **devmap_do_ctxmgt()**
  - Perform device context switching on a mapping

- **devmap_set_ctx_timeout()**
  - Set the timeout value for the context management callback

- **devmap_default_access()**
  - Default driver memory access function

- **ddi_umem_alloc()**
  - Allocate page-aligned kernel memory

- **ddi_umem_free()**
  - Free page-aligned kernel memory

- **ddi_umem_lock()**
  - Lock memory pages

- **ddi_umem_unlock()**
  - Unlock memory pages

- **ddi_umem_iostartup()**
  - Setup I/O requests to application memory

- **devmap_umem_setup()**
  - Export kernel memory to user space

- **ddi_model_convert_from()**
  - Determine data model type mismatch

---

**Table B-7**  Deprecated User Process Information Functions

<table>
<thead>
<tr>
<th>Deprecated Functions</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>drv_getparm()</td>
<td>ddi_get_pid(), ddi_get_cred()</td>
</tr>
</tbody>
</table>

**Table B-8**  Deprecated User Application Kernel and Device Access Functions

<table>
<thead>
<tr>
<th>Deprecated Function</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_mapdev()</td>
<td>devmap_setup()</td>
</tr>
<tr>
<td>ddi_mapdev_intercept()</td>
<td>devmap_load()</td>
</tr>
<tr>
<td>ddi_mapdev_no intercept()</td>
<td>devmapUnload()</td>
</tr>
</tbody>
</table>
### Time-Related Functions

The time-related functions are:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ddi_get_lbolt()</td>
<td>Return the number of clock ticks since reboot</td>
</tr>
<tr>
<td>ddi_get_time()</td>
<td>Return the current time in seconds</td>
</tr>
<tr>
<td>ddi_periodic_add()</td>
<td>Issue nanosecond periodic timeout requests</td>
</tr>
<tr>
<td>ddi_periodic_delete()</td>
<td>Cancel nanosecond periodic timeout requests</td>
</tr>
<tr>
<td>delay()</td>
<td>Delay execution for a specified number of clock ticks</td>
</tr>
<tr>
<td>drv_hztosec()</td>
<td>Convert clock ticks to microseconds</td>
</tr>
<tr>
<td>drv_usectohz()</td>
<td>Convert microseconds to clock ticks</td>
</tr>
<tr>
<td>drv_usecwait()</td>
<td>Busy-wait for specified interval</td>
</tr>
<tr>
<td>gethrtime()</td>
<td>Get high-resolution time</td>
</tr>
<tr>
<td>gethrvtime()</td>
<td>Get high-resolution LWP virtual time</td>
</tr>
<tr>
<td>timeout()</td>
<td>Execute a function after a specified length of time</td>
</tr>
<tr>
<td>untimout()</td>
<td>Cancel the previous time out function call</td>
</tr>
</tbody>
</table>

#### TABLE B–9  Deprecated Time-Related Functions

<table>
<thead>
<tr>
<th>Deprecated Function</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>drv_getparm()</td>
<td>ddi_get_lbolt(), ddi_get_time()</td>
</tr>
</tbody>
</table>
Power Management Functions

The power management functions are:

- `ddi_removing_power()`: Check if device loses power with DDI_SUSPEND
- `pci_report_pmcap()`: Report the power management capability of a PCI device
- `pm_busy_component()`: Mark a component as busy
- `pm_idle_component()`: Mark a component as idle
- `pm_raise_power()`: Raise the power level of a component
- `pm_lower_power()`: Lower the power level of a component
- `pm_power_has_changed()`: Notify the power management framework of an autonomous power level change
- `pm_trans_check()`: Device power cycle advisory check

### TABLE B–10  Deprecated Power Management Functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ddi_dev_is_needed()</code></td>
<td>Inform the system that a device’s component is required</td>
</tr>
<tr>
<td><code>pm_create_components()</code></td>
<td>Create power-manageable components</td>
</tr>
<tr>
<td><code>pm_destroy_components()</code></td>
<td>Destroy power-manageable components</td>
</tr>
<tr>
<td><code>pm_get_normal_power()</code></td>
<td>Get the normal power level of a device component</td>
</tr>
<tr>
<td><code>pm_set_normal_power()</code></td>
<td>Set the normal power level of a device component</td>
</tr>
</tbody>
</table>

Fault Management Functions

The fault management functions are:

- `ddi_fm_init()`: Allocates and initializes resources based on declared fault management capabilities
- `ddi_fm_fini()`: Cleans up resources that were allocated for this device instance to support fault management capabilities declared to `ddi_fm_init()`
- `ddi_fm_capable()`: Returns the capability bit mask currently set for this device instance
- `ddi_fm_handler_register()`: Registers an error handler callback routine with the IO Fault Management framework
Kernel Statistics Functions

The kernel statistics (kstats) functions are:

- `kstat_create()`: Create and initialize a new kstat
- `kstat_delete()`: Remove a kstat from the system
- `kstat_install()`: Add a fully initialized kstat to the system
- `kstat_named_init()`: Initialize a named kstat
- `kstat_runq_back_to_waitq()`: Record a transaction migration from run queue to the wait queue
- `kstat_runq_enter()`: Record a transaction addition to the run queue
- `kstat_runq_exit()`: Record a transaction removal from the run queue
- `kstat_waitq_enter()`: Record a transaction addition to the wait queue
- `kstat_waitq_exit()`: Record a transaction removal from the wait queue
- `kstat_waitq_to_runq()`: Record a transaction migration from the wait queue to the run queue
Kernel Logging and Printing Functions

The kernel logging and printing functions are:

- `cmn_err()`: Display an error message
- `vcmn_err()`: Display an error message
- `ddi_report_dev()`: Announce a device
- `strlog()`: Submit messages to the log driver
- `ddi_dev_report_fault()`: Report a hardware failure
- `scsi_errmsg()`: Display a SCSI request sense message
- `scsi_log()`: Display a SCSI-device-related message
- `scsi_vu_errmsg()`: Display a SCSI request sense message

Buffered I/O Functions

The buffered I/O functions are:

- `physio()`: Perform physical I/O
- `aphysio()`: Perform asynchronous physical I/O
- `anocancel()`: Prevent cancellation of an asynchronous I/O request
- `minphys()`: Limit the `physio()` buffer size
- `biowait()`: Suspend processes pending completion of block I/O
- `biodone()`: Release the buffer after buffer I/O transfer and notify blocked threads
- `bioerror()`: Indicate the error in a buffer header
- `geterror()`: Return an I/O error
- `bp_mapin()`: Allocate virtual address space
- `bp_mapout()`: Deallocate virtual address space
- `disksort()`: Use a single-direction elevator seek strategy to sort for buffers
- `getrbuf()`: Get a raw buffer header
- `freerbuf()`: Free a raw buffer header
- `biosize()`: Return the size of a buffer structure
- `boinit()`: Initialize a buffer structure
- `biofini()`: Uninitialize a buffer structure
**Virtual Memory Functions**

The virtual memory functions are:

- **ddi_btop()**  Convert device bytes to pages (round down)
- **ddi_btopr()** Convert device bytes to pages (round up)
- **ddi_ptob()**  Convert device pages to bytes
- **btop()**     Convert size in bytes to size in pages (round down)
- **btopr()**    Convert size in bytes to size in pages (round up)
- **ptob()**     Convert size in pages to size in bytes

**TABLE B–11  Deprecated Virtual Memory Functions**

<table>
<thead>
<tr>
<th>Deprecated Functions</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>hat_getkpfnum()</td>
<td>devmap(), ddi_dma_*_bind_handle(), ddi_dma_nextcookie()</td>
</tr>
</tbody>
</table>

**Device ID Functions**

The device ID functions are:

- **ddi_devid_init()** Allocate a device ID structure
- **ddi_devid_free()** Free a device ID structure
- **ddi_devid_register()** Register a device ID
- **ddi_devid_unregister()** Unregister a device ID
- **ddi_devid_compare()** Compare two device IDs
- **ddi_devid_sizeof()** Return the size of a device ID
- **ddi_devid_valid()** Validate a device ID
- **ddi_devid_str_encode()** Encode a device ID and minor_name into a null-terminated ASCII string; return a pointer to that string
**SCSI Functions**

The SCSI functions are:

- `scsi_probe()`: Probe a SCSI device
- `scsi_unprobe()`: Free resources allocated during initial probing
- `scsi_alloc_consistent_buf()`: Allocate an I/O buffer for SCSI DMA
- `scsi_free_consistent_buf()`: Free a previously allocated SCSI DMA I/O buffer
- `scsi_init_pkt()`: Prepare a complete SCSI packet
- `scsi_destroy_pkt()`: Free an allocated SCSI packet and its DMA resource
- `scsi_setup_cdb()`: Set up SCSI command descriptor block (CDB)
- `scsi_transport()`: Start a SCSI command
- `scsi_poll()`: Run a polled SCSI command
- `scsi_ifgetcap()`: Get SCSI transport capability
- `scsi_ifsetcap()`: Set SCSI transport capability
- `scsi_sync_pkt()`: Synchronize CPU and I/O views of memory
- `scsi_abort()`: Abort a SCSI command
- `scsi_reset()`: Reset a SCSI bus or target
- `scsi_reset_notify()`: Notify the target driver of bus resets
- `scsi_cname()`: Decode a SCSI command
- `scsi_dname()`: Decode a SCSI peripheral device type
- `scsi_mname()`: Decode a SCSI message
- `scsi_rname()`: Decode a SCSI packet completion reason
- `scsi_sname()`: Decode a SCSI sense key
- `scsi_errmsg()`: Display a SCSI request sense message
- `scsi_log()`: Display a SCSI-device-related message

`ddi_devid_str_decode()` Decodes the device ID and minor_name from a previously encoded string; allocate and return pointers to the extracted parts

`ddi_devid_str_free()` Free all strings returned by the `ddi_devid_*` functions
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>scsi_vu_errmsg()</td>
<td>Display a SCSI request sense message</td>
</tr>
<tr>
<td>scsi_hba_init()</td>
<td>SCSI HBA system initialization routine</td>
</tr>
<tr>
<td>scsi_hba_fini()</td>
<td>SCSI HBA system completion routine</td>
</tr>
<tr>
<td>scsi_hba_attach_setup()</td>
<td>SCSI HBA attach routine</td>
</tr>
<tr>
<td>scsi_hba_detach()</td>
<td>SCSI HBA detach routine</td>
</tr>
<tr>
<td>scsi_hba_probe()</td>
<td>Default SCSI HBA probe function</td>
</tr>
<tr>
<td>scsi_hba_tran_alloc()</td>
<td>Allocate a transport structure</td>
</tr>
<tr>
<td>scsi_hba_tran_free()</td>
<td>Free a transport structure</td>
</tr>
<tr>
<td>scsi_hba_pkt_alloc()</td>
<td>Allocate a scsi_pkt structure</td>
</tr>
<tr>
<td>scsi_hba_pkt_free()</td>
<td>Free a scsi_pkt structure</td>
</tr>
<tr>
<td>scsi_hba_lookup_capstr()</td>
<td>Return an index matching capability string</td>
</tr>
</tbody>
</table>

### TABLE B-12  Deprecated SCSI Functions

<table>
<thead>
<tr>
<th>Deprecated Function</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>free_pktiopb()</td>
<td>scsi_free_consistent_buf()</td>
</tr>
<tr>
<td>get_pktiopbs()</td>
<td>scsi_alloc_consistent_buf()</td>
</tr>
<tr>
<td>makecom_g0()</td>
<td>scsi_setup_cdb()</td>
</tr>
<tr>
<td>makecom_g0_s()</td>
<td>scsi_setup_cdb()</td>
</tr>
<tr>
<td>makecom_g1()</td>
<td>scsi_setup_cdb()</td>
</tr>
<tr>
<td>makecom_g5()</td>
<td>scsi_setup_cdb()</td>
</tr>
<tr>
<td>scsi_dmafree()</td>
<td>scsi_destroy_pkt()</td>
</tr>
<tr>
<td>scsi_dmget()</td>
<td>scsi_init_pkt()</td>
</tr>
<tr>
<td>scsi_hba_attach()</td>
<td>scsi_hba_attach_setup()</td>
</tr>
<tr>
<td>scsi_pktalloc()</td>
<td>scsi_init_pkt()</td>
</tr>
<tr>
<td>scsi_pktfree()</td>
<td>scsi_destroy_pkt()</td>
</tr>
<tr>
<td>scsi_resalloc()</td>
<td>scsi_init_pkt()</td>
</tr>
<tr>
<td>scsi_resfree()</td>
<td>scsi_destroy_pkt()</td>
</tr>
<tr>
<td>scsi_slave()</td>
<td>scsi_probe()</td>
</tr>
<tr>
<td>scsi_unslave()</td>
<td>scsi_unprobe()</td>
</tr>
</tbody>
</table>
Resource Map Management Functions

The resource map management functions are:

- `rmallocmap()` Allocate a resource map
- `rmallocmap_wait()` Allocate a resource map, wait if necessary
- `rmfree MAP()` Free a resource map
- `rmalloc()` Allocate space from a resource map
- `rmalloc_wait()` Allocate space from a resource map, wait if necessary
- `rmfree()` Free space back into a resource map

System Global State

- `ddi_in_panic()` Determine whether the system is in panic state

Utility Functions

The following list provides a subset of the utility functions that are available.

- `nulldev()` Zero return function
- `nodev()` Error return function
- `nochpoll()` Error return function for non-pollable devices
- `ASSERT()` Expression verification
- `bcopy()` Copy data between address locations in the kernel
- `bzero()` Clear memory for a given number of bytes
- `bcmp()` Compare two byte arrays
- `ddi_ffs()` Find the first bit set in a long integer
- `ddi_fls()` Find the last bit set in a long integer
- `swab()` Swap bytes in 16-bit halfwords
- `strcat()` Append two strings
- `strncat()` Append two strings, with length limit
- `strlcat()` Append two strings, with length and buffer limit
- `strcmp()` Compare two null-terminated strings
Utility Functions

- **strncmp()**: Compare two null-terminated strings, with length limit.
- **strlen()**: Determine the number of non-null bytes in a string.
- **strnlen()**: Determine the number of non-null bytes in a string, with length limit.
- **strcpy()**: Copy a string from one location to another.
- **strncpy()**: Copy a string from one location to another, with length limit.
- **strlcpy()**: Copy a string from one location to another with length limit and buffer limit.
- **strcasecmp()**: Case insensitive version of `strcmp()`.
- **strncasecmp()**: Case insensitive version of `strncmp()`.
- **strchr()**: Find a character in a string.
- **strstr()**: Locate the first occurrence of one string in another string.
- **strcasestr()**: Case insensitive version of `strstr()`.
- **strfree()**: Frees memory associated with a string.
- **sprintf(), vsprintf()**: Format characters in memory.
- **snprintf()**: Format characters in memory with buffer size specified.
- **numtos()**: Convert an integer to a decimal string.
- **stoi()**: Convert a decimal string to an integer.
- **max()**: Return the larger of two integers.
- **min()**: Return the lesser of two integers.
- **va_arg()**: Finds the next value in a variable argument list.
- **va_copy()**: Copies the state of a variable argument list.
- **va_end()**: Deletes pointer to a variable argument list.
- **va_start()**: Finds the pointer to the start of a variable argument list.

See the *man pages section 9: DDI and DKI Kernel Functions* for detailed information about the utility functions.
Making a Device Driver 64-Bit Ready

This appendix provides information for device driver writers who are converting their device drivers to support the 64-bit kernel. It presents the differences between 32-bit and 64-bit device drivers and describes the steps to convert 32-bit device drivers to 64-bit. This information is specific to regular character and block device drivers only.

This appendix provides information on the following subjects:
- “Introduction to 64-Bit Driver Design” on page 585
- “General Conversion Steps” on page 586
- “Well Known ioctl Interfaces” on page 593

Introduction to 64-Bit Driver Design

For drivers that only need support for the 32-bit kernel, existing 32-bit device drivers will continue to work without recompilation. However, most device drivers require some changes to run correctly in the 64-bit kernel, and all device drivers require recompilation to create a 64-bit driver module. The information in this appendix will help you to enable drivers for 32-bit and 64-bit environments to be generated from common source code, thus increasing code portability and reducing the maintenance effort.

Before starting to modify a device driver for the 64-bit environment, you should understand how the 32-bit environment differs from the 64-bit environment. In particular, you must be familiar with the C language data type models ILP32 and LP64. See the following table.

<table>
<thead>
<tr>
<th>CType</th>
<th>ILP32</th>
<th>LP64</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
<td>16</td>
</tr>
</tbody>
</table>

TABLE C-1 Comparison of ILP32 and LP64 Data Types
The driver-specific issues due to the differences between ILP32 and LP64 are the subject of this appendix.

In addition to general code cleanup to support the data model changes for LP64, driver writers have to provide support for both 32-bit and 64-bit applications.

The `ioctl(9E)`, `devmap(9E)`, and `mmap(9E)` entry points enable data structures to be shared directly between applications and device drivers. If those data structures change size between the 32-bit and 64-bit environments, then the entry points must be modified so that the driver can determine whether the data model of the application is the same as that of the kernel. When the data models differ, data structures can be adjusted. See "I/O Control Support for 64-Bit Capable Device Drivers" on page 280, "32-bit and 64-bit Data Structure Macros" on page 283, and "Associating Kernel Memory With User Mappings" on page 182.

In many drivers, only a few `ioctl`s need this kind of handling. The other `ioctl`s should work without change as long as these `ioctl`s pass data structures that do not change in size.

### General Conversion Steps

The sections below provide information on converting drivers to run in a 64-bit environment. Driver writers might need to perform one or more of the following tasks:

1. Use fixed-width types for hardware registers.
2. Use fixed-width common access functions.
3. Check and extend use of derived types.
4. Check changed fields within DDI data structures.
5. Check changed arguments of DDI functions.
6. Modify the driver entry points that handle user data, where needed.
7. Check structures that use 64-bit long types on x86 platforms.

These steps are explained in detail below.
After each step is complete, fix all compiler warnings, and use lint to look for other problems. The SC5.0 (or newer) version of lint should be used with -Xarch=v9 and -errchk=longptr64 specified to find 64-bit problems.

**Note** – Do not ignore compilation warnings during conversion for LP64. Warnings that were safe to ignore previously in the ILP32 environment might now indicate a more serious problem.

After all the steps are complete, compile and test the driver as both a 32-bit and 64-bit module.

**Use Fixed-Width Types for Hardware Registers**

Many device drivers that manipulate hardware devices use C data structures to describe the layout of the hardware. In the LP64 data model, data structures that use `long` or unsigned `long` to define hardware registers are almost certainly incorrect, because `long` is now a 64-bit quantity. Start by including `<sys/inttypes.h>`, and update this class of data structure to use `int32_t` or `uint32_t` instead of `long` for 32-bit device data. This approach preserves the binary layout of 32-bit data structures. For example, change:

```c
struct device_regs {
    ulong_t addr;
    uint_t count;
}; /* Only works for ILP32 compilation */
```

to:

```c
struct device_regs {
    uint32_t addr;
    uint32_t count;
}; /* Works for any data model */
```

**Use Fixed-Width Common Access Functions**

The Oracle Solaris DDI allows device registers to be accessed by access functions for portability to multiple platforms. Previously, the DDI common access functions specified the size of data in terms of bytes, words, and so on. For example, `ddi_getl(9F)` is used to access 32-bit quantities. This function is not available in the 64-bit DDI environment, and has been replaced by versions of the function that specify the number of bits to be acted on.

These routines were added to the 32-bit kernel in the Solaris 2.6 operating environment, to enable their early adoption by driver writers. For example, to be portable to both 32-bit and 64-bit kernels, the driver must use `ddi_get32(9F)` to access 32-bit data rather than `ddi_getl(9F)`.

All common access routines are replaced by their fixed-width equivalents. See the `ddi_get8(9F), ddi_put8(9F), ddi_rep_get8(9F), and ddi_rep_put8(9F)` man pages for details.
Check and Extend Use of Derived Types

System-derived types, such as size_t, should be used where possible so that the resulting variables make sense when passed between functions. The new derived types uintptr_t or intptr_t should be used as the integral type for pointers.

Fixed-width integer types are useful for representing explicit sizes of binary data structures or hardware registers, while fundamental C language data types, such as int, can still be used for loop counters or file descriptors.

Some system-derived types represent 32-bit quantities on a 32-bit system but represent 64-bit quantities on a 64-bit system. Derived types that change size in this way include: clock_t, daddr_t, dev_t, ino_t, intptr_t, off_t, size_t, ssize_t, time_t,uintptr_t, and timeout_id_t.

When designing drivers that use these derived types, pay particular attention to the use of these types, particularly if the drivers are assigning these values to variables of another derived type, such as a fixed-width type.

Check Changed Fields in DDI Data Structures

The data types of some of the fields within DDI data structures, such as buf(9S), have been changed. Drivers that use these data structures should make sure that these fields are being used appropriately. The data structures and the fields that were changed in a significant way are listed below.

buf Structure Changes

The fields listed below pertain to transfer size, which can now exceed more than 4 Gbytes.

```c
size_t b_bcount; /* was type unsigned int */
size_t b_resid; /* was type unsigned int */
size_t b_bufsize; /* was type long */
```

ddi_dma_attr

The ddi_dma_attr(9S) structure defines attributes of the DMA engine and the device. Because these attributes specify register sizes, fixed-width data types have been used instead of fundamental types.

ddi_dma_cookie Structure Changes

```c
uint32_t dmac_address; /* was type unsigned long */
size_t dmac_size; /* was type u_int */
```

The ddi_dma_cookie(9S) structure contains a 32-bit DMA address, so a fixed-width data type has been used to define the address. The size has been redefined as size_t.
**csi_arq_status Structure Changes**

```c
uint_t sts_rqpkt_state; /* was type u_long */
uint_t sts_rqpkt_statistics; /* was type u_long */
```

These fields in the structure do not need to grow and have been redefined as 32-bit quantities.

**scsi_pkt Structure Changes**

```c
uint_t pkt_flags; /* was type u_long */
int pkt_time; /* was type long */
ssize_t pkt_resid; /* was type long */
uint_t pkt_state; /* was type u_long */
uint_t pkt_statistics; /* was type u_long */
```

Because the `pkt_flags`, `pkt_state`, and `pkt_statistics` fields in the `scsi_pkt(9S)` structure do not need to grow, these fields have been redefined as 32-bit integers. The data transfer size `pkt_resid` field *does* grow and has been redefined as `ssize_t`.

**Check Changed Arguments of DDI Functions**

This section describes the DDI function argument data types that have been changed.

**getrbuf() Argument Changes**

```c
struct buf *getrbuf(int sleepflag);
```

In previous releases, `sleepflag` was defined as a type `long`.

**drv_getparm() Argument Changes**

```c
int drv_getparm(unsigned int parm, void *value_p);
```

In previous releases, `value_p` was defined as type `unsigned long`. In the 64-bit kernel, `drv_getparm(9F)` can fetch both 32-bit and 64-bit quantities. The interface does not define data types of these quantities, and simple programming errors can occur.

The following new routines offer a safer alternative:

```c
clock_t ddi_get_lbolt(void);
time_t ddi_get_time(void);
cred_t *ddi_get_cred(void);
pid_t ddi_get_pid(void);
```

Driver writers are strongly urged to use these routines instead of `drv_getparm(9F)`.

**delay() and timeout() Argument Changes**

```c
void delay(clock_t ticks);
timeout_id_t timeout(void (*func)(void *), void *arg, clock_t ticks);
```
The `ticks` argument to the `delay(9F)` and `timeout(9F)` routines has been changed from `long` to `clock_t`.

**rmallocmap() and rmallocmap_wait() Argument Changes**

```c
struct map *rmallocmap(size_t mapsize);
struct map *rmallocmap_wait(size_t mapsize);
```

The `mapsize` argument to the `rmallocmap(9F)` and `rmallocmap_wait(9F)` routines has been changed from `ulong_t` to `size_t`.

**scsi_alloc_consistent_buf() Argument Changes**

```c
struct buf *scsi_alloc_consistent_buf(struct scsi_address *ap,
    struct buf *bp, size_t datalen, uint_t bflags,
    int (*callback)(caddr_t), caddr_t arg);
```

In previous releases, `datalen` was defined as an `int` and `bflags` was defined as a `ulong`.

**uiomove() Argument Changes**

```c
int uiomove(caddr_t address, size_t nbytes,
    enum uio_rw rwflag, uio_t *uio_p);
```

The `nbytes` argument was defined as a `type long`, but because `nbytes` represents a size in bytes, `size_t` is more appropriate.

**cv_timedwait() and cv_timedwait_sig() Argument Changes**

```c
int cv_timedwait(kcondvar_t *cvp, kmutex_t *mp, clock_t timeout);
int cv_timedwait_sig(kcondvar_t *cvp, kmutex_t *mp, clock_t timeout);
```

In previous releases, the `timeout` argument to the `cv_timedwait(9F)` and `cv_timedwait_sig(9F)` routines was defined to be of type `long`. Because these routines represent time in ticks, `clock_t` is more appropriate.

**ddi_device_copy() Argument Changes**

```c
int ddi_device_copy(ddi_acc_handle_t src_handle,
    caddr_t src_addr, ssize_t src_advcnt,
    ddi_acc_handle_t dest_handle, caddr_t dest_addr,
    ssize_t dest_advcnt, size_t bytecount, uint_t dev_datasz);
```

The `src_advcnt`, `dest_advcnt`, `dev_datasz` arguments have changed type. These arguments were previously defined as `long`, `long`, and `ulong_t` respectively.

**ddi_device_zero() Argument Changes**

```c
int ddi_device_zero(ddi_acc_handle_t handle,
    caddr_t dev_addr, size_t bytecount, ssize_t dev_advcnt,
    uint_t dev_datasz);
```
In previous releases, `dev_advcnt` was defined as a type `long` and `dev_datasz` as a `ulong_t`.

**ddi_dma_mem_alloc() Argument Changes**

```c
int ddi_dma_mem_alloc(ddi_dma_handle_t handle,
    size_t length, ddi_device_acc_attr_t *accattrp,
    uint_t flags, int (*waitfp)(caddr_t), caddr_t arg,
    caddr_t *kaddrp, size_t *real_length,
    ddi_acc_handle_t *handlep);
```

In previous releases, `length`, `flags`, and `real_length` were defined with types `uint_t`, `ulong_t`, and `uint_t *`.

**Modify Routines That Handle Data Sharing**

If a device driver shares data structures that contain `longs` or pointers with a 32-bit application using `ioctl(9E)`, `devmap(9E)`, or `mmap(9E)`, and the driver is recompiled for a 64-bit kernel, the binary layout of data structures will be incompatible. If a field is currently defined in terms of `type long` and 64-bit data items are not used, change the data structure to use data types that remain as 32-bit quantities (int and unsigned int). Otherwise, the driver needs to be aware of the different structure shapes for ILP32 and LP64 and determine whether a model mismatch between the application and the kernel has occurred.

To handle potential data model differences, the `ioctl()`, `devmap()`, and `mmap()` driver entry points, which interact directly with user applications, need to be written to determine whether the argument came from an application using the same data model as the kernel.

**Data Sharing in ioctl()**

To determine whether a model mismatch exists between the application and the driver, the driver uses the `FMODELS` mask to determine the model type from the `ioctl()` `mode` argument. The following values are OR-ed into mode to identify the application data model:

- **FLP64** – Application uses the LP64 data model
- **FILP32** – Application uses the ILP32 data model

The code examples in "I/O Control Support for 64-Bit Capable Device Drivers" on page 280 show how this situation can be handled using `ddi_model_convert_from(9F)`.

**Data Sharing in devmap()**

To enable a 64-bit driver and a 32-bit application to share memory, the binary layout generated by the 64-bit driver must be the same as the layout consumed by the 32-bit application. The mapped memory being exported to the application might need to contain data-model-dependent data structures.
Few memory-mapped devices face this problem because the device registers do not change size when the kernel data model changes. However, some pseudo-devices that export mappings to the user address space might want to export different data structures to ILP32 or LP64 applications. To determine whether a data model mismatch has occurred, devmap(9E) uses the model parameter to describe the data model expected by the application. The model parameter is set to one of the following values:

- **DDI_MODEL_ILP32** – The application uses the ILP32 data model
- **DDI_MODEL_LP64** – The application uses the LP64 data model

The model parameter can be passed untranslated to the ddi_model_convert_from(9F) routine or to STRUCT_INIT(). See “32-bit and 64-bit Data Structure Macros” on page 283.

**Data Sharing in mmap()**

Because mmap(9E) does not have a parameter that can be used to pass data model information, the driver’s mmap(9E) entry point can be written to use the new DDI function ddi_model_convert_from(9F). This function returns one of the following values to indicate the application’s data type model:

- **DDI_MODEL_ILP32** – Application expects the ILP32 data model
- **DDI_MODEL_LP64** – Application expects the LP64 data model
- **DDI_FAILURE** – Function was not called from mmap(9E)

As with ioctl() and devmap(), the model bits can be passed to ddi_model_convert_from(9F) to determine whether data conversion is necessary, or the model can be handed to STRUCT_INIT().

Alternatively, migrate the device driver to support the devmap(9E) entry point.

**Check Structures with 64-bit Long Data Types on x86-Based Platforms**

You should carefully check structures that use 64-bit long types, such as uint64_t, on the x86 platforms. The alignment and size can differ between compilation in 32-bit mode versus a 64-bit mode. Consider the following example.

```c
#include <stdio.h>
#include <sys>

struct myTestStructure {
    uint32_t  my1stInteger;
    uint64_t  my2ndInteger;
};

main()
{

```
struct myTestStructure a;
printf("sizeof myTestStructure is: %d", sizeof(a));
printf("offset to my2ndInteger is: %d", (uintptr_t)&a.bar - (uintptr_t)&a);
}

On a 32-bit system, this example displays the following results:

sizeof myTestStructure is: 12
offset to my2ndInteger is: 4

Conversely, on a 64-bit system, this example displays the following results:

sizeof myTestStructure is: 16
offset to my2ndInteger is: 8

Thus, the 32-bit application and the 64-bit application view the structure differently. As a result, trying to make the same structure work in both a 32-bit and 64-bit environment can cause problems. This situation occurs often, particularly in situations where structures are passed into and out of the kernel through ioctl() calls.

Well Known ioctl Interfaces

Many ioctl(9E) operations are common to a class of device drivers. For example, most disk drivers implement many of the dkio(7I) family of ioctls. Many of these interfaces copy in or copy out data structures from the kernel, and some of these data structures have changed size in the LP64 data model. The following section lists the ioctls that now require explicit conversion in 64-bit driver ioctl routines for the dkio, fbio(7I), cdio(7I), and mtio(7I) families of ioctls.

<table>
<thead>
<tr>
<th>ioctl command</th>
<th>Affected data structure</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>DKIOC GAPART</td>
<td>dk_map</td>
<td>dkio(7I)</td>
</tr>
<tr>
<td>DKIOC SAPART</td>
<td>dk_allmap</td>
<td></td>
</tr>
<tr>
<td>DKIOC GVTOC</td>
<td>partition</td>
<td>dkio(7I)</td>
</tr>
<tr>
<td>DKIOC VTOC</td>
<td>vtoc</td>
<td></td>
</tr>
<tr>
<td>FBIOPUTCMAP</td>
<td>fbcmap</td>
<td>fbio(7I)</td>
</tr>
<tr>
<td>FBIO GETCMAP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FBIOPUTCMAPI</td>
<td>fbcmap_i</td>
<td>fbio(7I)</td>
</tr>
<tr>
<td>FBIO GETCMAPI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FBIOC CURSOR</td>
<td>fbcursor</td>
<td>fbio(7I)</td>
</tr>
<tr>
<td>FBIO CURSOR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Well Known ioctl Interfaces

<table>
<thead>
<tr>
<th>ioctl command</th>
<th>Affected data structure</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDROMREADMODE1</td>
<td>cdrom_read</td>
<td>cdio(7I)</td>
</tr>
<tr>
<td>CDROMREADMODE2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CDROMCDDA</td>
<td>cdrom_cdda</td>
<td>cdio(7I)</td>
</tr>
<tr>
<td>CDROMCDXA</td>
<td>cdrom_cdxa</td>
<td>cdio(7I)</td>
</tr>
<tr>
<td>CDROMSUBCODE</td>
<td>cdrom_subcode</td>
<td>cdio(7I)</td>
</tr>
<tr>
<td>MTIOTOP</td>
<td>mtop</td>
<td>mtio(7I)</td>
</tr>
<tr>
<td>MTIOCGET</td>
<td>mtget</td>
<td>mtio(7I)</td>
</tr>
<tr>
<td>MTIOCGETDRIVETYPE</td>
<td>mt drivetype_request</td>
<td>mtio(7I)</td>
</tr>
<tr>
<td>USCSICMD</td>
<td>uscsi_cmd</td>
<td>scsi_free_consistent_buf(9F)</td>
</tr>
</tbody>
</table>

Device Sizes

The nblocks property is exported by each slice of a block device driver. This property contains the number of 512-byte blocks that each slice of the device can support. The nblocks property is defined as a signed 32-bit quantity, which limits the maximum size of a slice to 1 Tbyte.

Disk devices that provide more than 1 Tbyte of storage per disk must define the Nbblocks property, which should still contain the number of 512 byte blocks that the device can support. However, Nbblocks is a signed 64-bit quantity, which removes any practical limit on disk space.

The nblocks property is now deprecated. All disk devices should provide the Nbblocks property.
Drivers for frame buffers that are used for the system console must provide interfaces to enable the system to display text on the console. The Oracle Solaris OS provides enhanced visual I/O interfaces to enable the kernel terminal emulator to display text directly on the console frame buffer. This appendix describes how to add the necessary interfaces to a frame buffer driver to enable the driver to interact with the Oracle Solaris kernel terminal emulator.

### Oracle Solaris Consoles and the Kernel Terminal Emulator

The role of the kernel terminal emulator is to render text onto the console frame buffer in the proper position and representation determined by the frame buffer’s screen height, width, and pixel depth mode. The terminal emulator also drives scrolling, controls a software cursor, and interprets ANSI terminal escape sequences. The terminal emulator accesses the console frame buffer in either VGA text mode or pixel mode, depending upon the graphics card. To be used as a Oracle Solaris console frame buffer driver, your frame buffer driver must be compatible with the Oracle Solaris kernel terminal emulator. The target platform is the most significant factor that determines whether you need to modify your frame buffer driver to make your driver compatible with the Oracle Solaris kernel terminal emulator.

- **x86 platforms** – Console frame buffer drivers do not need to be modified because x86 console frame buffer drivers already support the console frame buffer interfaces.
- **SPARC platforms** – Console frame buffer drivers should use the interfaces described in this appendix to enable the driver to interact with the Oracle Solaris kernel terminal emulator.

### x86 Platform Console Communication

On x86 platforms, the Oracle Solaris kernel terminal emulator module (tem) uses VGA text mode exclusively to interact with the vgatext module. The vgatext module uses industry standard VGA text mode to interact with x86 compatible frame buffer devices. Because the
vga text module already supports the console frame buffer interfaces, x86 frame buffer drivers are compatible with the kernel tem module. You do not need to add special interfaces to x86 frame buffer drivers.

The remainder of this appendix applies to SPARC platforms only.

**SPARC Platform Console Communication**

SPARC frame buffer drivers typically do not operate in VGA text mode. SPARC frame buffer drivers typically are required to send pixel patterns that depict the text and images displayed. The kernel tem requires SPARC drivers to support specific interfaces to facilitate rendering data to the screen, perform scrolling, and display a text cursor. How the driver actually renders data sent from the tem onto the screen depends on the device. The driver typically draws the data into video memory according to the hardware and video mode.

The Oracle Solaris OS provides interfaces that enable the kernel terminal emulator to drive compatible console frame buffers directly. The advantages of converting a driver to be compatible with the kernel terminal emulator are:

- Dramatically improved performance, particularly for scrolling
- Enhanced ANSI text color capabilities
- The ability to start a login session on the console frame buffer even when the system console stream is directed out the serial port

SPARC console frame buffer drivers are not required to be compatible with the kernel terminal emulator. If the console frame buffer driver is not compatible with the kernel terminal emulator, the system uses the FCode terminal emulator in the OpenBoot PROM.

The console frame buffer is identified through the EEPROM screen environment variable. The system determines whether the console frame buffer is compatible with the kernel terminal emulator module by checking whether the frame buffer driver exports the tem-support DDI property. If the tem-support property is exported, then the system issues the VIS_DEVINIT I/O control (ioctl) command to the frame buffer driver during system boot, while configuring the console. If the tem-support DDI property is exported and the VIS_DEVINIT ioctl command succeeds and returns a compatible version number to the tem, the system configures the system console to utilize that frame buffer driver through the kernel terminal emulator. See the ioctl(9E) man page for information about the I/O control driver entry point.

SPARC drivers that support the kernel terminal emulator should export the tem-support DDI property. This property indicates that the driver supports the kernel terminal emulator. If a frame buffer driver exports the tem-support DDI property, then that driver will be handled early in the boot process, while the console is being configured. If a frame buffer driver does not export the tem-support property, then that driver might not be handled early enough in the boot process.
tem-support When set to 1, this DDI property indicates that this driver is compatible with the console kernel frame buffer interface.

The kernel terminal emulator module interacts with the console frame buffer driver through two major interfaces:

- Through ioctl interfaces during normal system operation
- Through polled I/O interfaces during standalone mode

The following section provides detailed information.

**Console Visual I/O Interfaces**

The kernel terminal emulator interacts with the console frame buffer driver through two interfaces. During normal system activity (after a successful boot of the system), communication between the kernel terminal emulator and the console frame buffer driver is through ioctl interfaces. During standalone mode (before system boot or during debugging), communication between the kernel terminal emulator and the console frame buffer driver is through polled I/O interfaces. All activity between the kernel terminal emulator and the console frame buffer driver is initiated by the kernel terminal emulator, with the exception of a callback function used by the console frame buffer driver to notify the kernel terminal emulator of changes in the video mode.

The console visual I/O interfaces are documented in detail in the `visual_io(7I)` man page. For more information on the video mode change callback function, see “Video Mode Change Callback Interface” on page 598.

**I/O Control Interfaces**

During normal system activity, the kernel terminal emulator communicates with the console frame buffer driver through the ioctl interfaces listed in the following table:

<table>
<thead>
<tr>
<th>ioctl Name</th>
<th>Corresponding Data Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VIS_DEVINIT</td>
<td>vis_devinit</td>
<td>Initializes the session between the terminal emulator module and the frame buffer. See &quot;VIS_DEVINIT&quot; on page 599.</td>
</tr>
<tr>
<td>VIS_DEVFINI</td>
<td>Not Applicable</td>
<td>Terminates the session between the terminal emulator module and the frame buffer. See &quot;VIS_DEVFINI&quot; on page 601.</td>
</tr>
</tbody>
</table>
**Polled I/O Interfaces**

The polled I/O interfaces provide the same functionality as the VIS_CONSDISPLAY, VIS_CONSCOPY, and VIS_CONSCURSOR ioctl interfaces. The polled I/O interfaces are called only when the operating system is quiesced and in standalone mode. See “Implementing Polled I/O in Console Frame Buffer Drivers” on page 605 for more information.

While in standalone mode, the kernel terminal emulator communicates with the console frame buffer driver through the polled I/O interfaces listed in the following table:

<table>
<thead>
<tr>
<th>Polled I/O Function</th>
<th>Corresponding Data Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(*display)()</td>
<td>vis_consdisplay</td>
<td>Displays pixels as a rectangle.</td>
</tr>
<tr>
<td>(*copy)()</td>
<td>vis_conscopy</td>
<td>Copies a rectangle of pixels (scroll).</td>
</tr>
<tr>
<td>(*cursor)()</td>
<td>vis_conscursor</td>
<td>Displays or hides a text cursor.</td>
</tr>
</tbody>
</table>

**Video Mode Change Callback Interface**

The console frame buffer driver and the kernel terminal emulator must be in agreement about the video mode at all times. Video mode includes the console screen height, width, and depth in pixels. Video mode also includes whether communication between the kernel terminal emulator and the console frame buffer is in VGA text mode or pixel mode.
In order for the console frame buffer driver to notify the kernel terminal emulator of changes in the video mode, the console frame buffer driver is initialized with the address of the (*modechg_cb)() kernel terminal emulator callback function described in the following table:

<table>
<thead>
<tr>
<th>Callback Function</th>
<th>Corresponding Data Structures</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(*modechg_cb)()</td>
<td>vis_modechg_arg</td>
<td>Keep the terminal emulator module synchronized with the driver video mode (screen height, width, and pixel depth).</td>
</tr>
<tr>
<td></td>
<td>vis_devinit</td>
<td></td>
</tr>
</tbody>
</table>

### Implementing the Visual I/O Interfaces in Console Frame Buffer Drivers

Except for the video mode change callback, all activity between the driver and the kernel terminal emulator is initiated by the tem (terminal emulator module). This means that the tem issues all of the ioctl commands described in this document. The following sections provide implementation details for each ioctl command. For more information, see the visual_io(7I) man page and the /usr/include/sys/visual_io.h include file. See "Video Mode Change Callback Interface" on page 598 for detailed information about the video mode change callback function.

**Note** - Each ioctl command should determine whether the FKIOCTL is set in the ioctl flag argument and return EPERM if that bit is not set.

### VIS_DEVINIT

The VIS_DEVINIT ioctl command initializes the frame buffer driver as the system console device. This ioctl passes the address of a vis_devinit structure.

The tem first loads the address of its video mode change callback function into the modechg_cb field of the vis_devinit structure and loads its soft state into the modechg_arg field. The tem then issues the VIS_DEVINIT ioctl command. The frame buffer driver then initializes itself and returns a summary of its configuration back to the tem by setting the version, width, height, linebytes, depth, mode, and polledio fields in the vis_devinit structure. The vis_devinit structure is shown in the following code.

```c
struct vis_devinit {
    /*
     * This set of fields are used as parameters passed from the
     * layered frame buffer driver to the terminal emulator.
     */
```
To implement the `VIS_DEVINIT` ioctl command in the console framebuffer driver, follow these general steps:

1. **Define a structure to contain the console-specific state.** This structure is private to the console framebuffer driver. This structure is referred to as `consinfo` in this appendix. The `consinfo` structure contains information such as:
   - Current size of the blit buffer
   - Pointer to the blit buffer
   - Color map information
   - Driver rendering mode information such as line pitch
   - Background color
   - Video memory address
   - Terminal emulator callback address

2. **Allocate memory:**
   a. Allocate a blit buffer large enough to store a reasonable default sized rectangle of pixels at the highest video depth. Additional memory can be allocated if an incoming request exceeds the size of the buffer. The frame buffer driver’s largest font is 12×22. Assuming `DEFAULT_HEIGHT` is 12, `DEFAULT_WIDTH` is 22, and the maximum video depth is 32, the buffer size should be 8448 bytes (`DEFAULT_HEIGHT × DEFAULT_WIDTH × 32`).
   b. Allocate a `vis_polledio` structure.
   c. Allocate a buffer to hold a cursor. This buffer should be the size of the largest character. This buffer will not change size.

3. **Obtain the video change callback address and callback context of the term from `modechg_cb` and `modechg_ctx` and store this information in the `consinfo` structure.

4. **Populate the `vis_polledio` structure with entry point addresses for the polled display, copy, and cursor functions.

5. **Provide the appropriate information in the fields of the `vis_devinit` structure that was passed to the driver by the term:**
   a. Set the version field to `VIS_CONS_REV`, which is a constant defined in the `/usr/include/sys/visual_io.h` header file.
   b. Set the mode field to `VIS_PIXEL`. 

Implementing the Visual I/O Interfaces in Console Frame Buffer Drivers
c. Set the polledio field to the address of the vis_polledio structure.
d. Set the height field to the video mode height in pixels.
e. Set the width field to the video mode width in pixels.
f. Set the depth field to the frame buffer pixel depth in bytes (for example, a 32-bit pixel depth would be 4 bytes).
g. Set the linebytes field to the value of height × width × depth.
   This information is sent from the driver to the tem by using the vis_devinit structure.
   This information tells the terminal emulator how to render information and pass it to
   the graphics driver.
   Whenever the console frame buffer driver changes its video mode (specifically height,
   width, or depth), the driver must call the video mode change callback function of the tem
to update the vis_devinit structure and to pass this structure back to the terminal emulator.
The terminal emulator passes its mode change callback function address in the modechg_cb
field of the vis_devinit structure. The mode change callback function has the following
function signature:

   typedef void (*vis_modechg_cb_t)
   (struct vis_modechg_arg *, struct vis_devinit *);

   As shown in the preceding typedef, the mode change callback function takes two
arguments. The first argument is the modechg_arg and the second argument is the
vis_devinit structure. The modechg_arg is sent from the tem to the driver during the
VIS_DEVINIT ioctl command initialization. The driver must send the modechg_arg back to
the tem with each video mode change callback.
6. Initialize the context of the kernel console. Specific requirements vary depending upon the
capability of the graphics device. This initialization might include such steps as setting the
draw engine state, initializing the palette, or locating and mapping video memory or the
rendering engine so that data can be blitted onto the screen.
7. Return the vis_devinit structure to the caller.

**VIS_DEFINI**

The VIS_DEFINI ioctl command releases the driver’s console resources and finishes the
session.

To implement the VIS_DEVFINI ioctl command in the console frame buffer driver, follow
these general steps:

1. Reset the console frame buffer driver state.
2. Clear the polled I/O entry points and the kernel terminal emulator video change function
callback address.
3. Release memory.
The **VIS_CONSDISPLAY** ioctl command displays a rectangle of pixels at a specified location. This display is also referred to as blitting a rectangle. The **vis_consdisplay** structure contains the information necessary to render a rectangle at the video depth that both the driver and the tem are using. The **vis_consdisplay** structure is shown in the following code.

```c
struct vis_consdisplay {
    screen_pos_t row; /* Row (in pixels) to display data at */
    screen_pos_t col; /* Col (in pixels) to display data at */
    screen_size_t width; /* Width of data (in pixels) */
    screen_size_t height; /* Height of data (in pixels) */
    unsigned char *data; /* Address of pixels to display */
    unsigned char fg_color; /* Foreground color */
    unsigned char bg_color; /* Background color */
};
```

To implement the **VIS_CONSDISPLAY** ioctl command in the console frame buffer driver, follow these general steps:

1. Copy the **vis_consdisplay** structure.
2. Validate the display parameters. Return an error if any of the display parameters is out of range.
3. Calculate the size of the rectangle to be blitted into video memory. Validate this size against the size of the blit buffer created during **VIS_DEVINIT**. Allocate additional memory for the blit buffer if necessary.
4. Retrieve the blit data. This data has been prepared by the kernel terminal emulator at the agreed upon pixel depth. That depth is the same pixel depth that was conveyed by the tem during **VIS_DEVINIT**. The pixel depth is updated whenever the device driver changes video modes through callback to the tem. Typical pixel depths are 8-bit color map indexed, and 32-bit TrueColor.
5. Invalidate any user context so that user applications cannot simultaneously access the frame buffer hardware through user memory mappings. This step is neither allowed nor necessary in polled I/O mode because user applications are not running. Be sure to hold a lock so that users cannot restore the mapping through a page fault until the **VIS_CONSDISPLAY** ioctl completes.
6. Establish the driver-specific console rendering context.
7. If the frame buffer is running in 8-bit color indexed mode, restore the kernel console color map that the tem set up through a previous **VIS_PUTCMP** ioctl. A lazy color map loading scheme is recommended to optimize performance. In a lazy scheme, the console frame buffer only restores colors it has actually used since the **VIS_DEVINIT** ioctl was issued.
8. Display the data passed from the tem at the pixel coordinates sent by the tem. You might need to transform the RGB pixel data byte order.
**VIS_CONSCOPY**

The **VIS_CONSCOPY** ioctl command copies a rectangular region of pixels from one location to another location. One use for this ioctl is to scroll.

To implement the **VIS_CONSCOPY** ioctl command in the console frame buffer driver, follow these general steps:

1. Copy the **vis_conscopy** structure. The **vis_conscopy** structure describes the source and target rectangle sizes and locations.
2. Validate the display parameters. Return an error if any of the display parameters is out of range.
3. Invalidate any user context so that user applications cannot simultaneously access the frame buffer hardware through user memory mappings. This step is neither allowed nor necessary in polled I/O mode because user applications are not running. Be sure to hold a lock so that users cannot restore the mapping through a page fault until the **VIS_CONSDISPLAY** ioctl completes.
4. Call the function to copy the rectangle.

---

**Note** – For optimal performance, use the rendering engine of the graphic device to implement the copy function. You need to decide how to do the context management within the driver to set up the rendering engine for best performance.

---

**VIS_CONSCURSOR**

The **VIS_CONSCURSOR** ioctl command displays or hides a cursor. The **vis_conscursor** structure is shown in the following code.

```c
struct vis_conscursor {
    screen_pos_t row;       /* Row to display cursor (in pixels) */
    screen_pos_t col;       /* Col to display cursor (in pixels) */
    screen_size_t width;    /* Width of cursor (in pixels) */
    screen_size_t height;   /* Height of cursor (in pixels) */
    color_t         fg_color;  /* Foreground color */
    color_t         bg_color;  /* Background color */
    short           action;    /* Show or Hide cursor */
};
```

To implement the **VIS_CONSCOPY** ioctl command in the console frame buffer driver, follow these general steps:

1. Copy the **vis_conscursor** structure from the kernel terminal emulator.
2. Validate the display parameters. Return an error if any of the display parameters are out of range.
3. Invalidate any user context so that user applications cannot simultaneously access the frame buffer hardware through user memory mappings. This step is neither allowed nor necessary in polled I/O mode because user applications are not running. Be sure to hold a lock so that users cannot restore the mapping through a page fault until the VIS_CONSDISPLAY ioctl completes.

4. The terminal emulator can call the VIS_CONSCOPY ioctl with one of the following two actions: SHOW_CURSOR and HIDE_CURSOR. The following steps describe how to implement this functionality by reading and writing video memory. You might also be able to use the rendering engine to do this work. Whether you can use the rendering engine depends on the frame buffer hardware.

Take these steps to implement the SHOW_CURSOR functionality:

a. Save the pixels within the rectangle where the cursor will be drawn. These saved pixels will be needed to hide the cursor.

b. Scan all the pixels on the screen bounded by the rectangle where the cursor will be drawn. Within this rectangle, replace the pixels that match the specified cursor foreground color ($fg\_color$) with white pixels. Replace the pixels that match the specified cursor background color ($bg\_color$) with black pixels. The visual effect is of a black cursor over white text. This method works with any foreground and background color of text. Attempting to invert colors based upon color map position is not feasible. More sophisticated strategies, such as attempting color inversion using HSB coloring (Hue, Saturation, Brightness), are not necessary.

To implement the HIDE_CURSOR functionality, replace the pixels beneath the cursor rectangle with the pixels saved from the previous SHOW_CURSOR action.

**VIS_PUTCMP**

The VIS_PUTCMP ioctl command establishes the console color map. The terminal emulator calls this function to set up the color map of the kernel. The vis_cmap structure is shown in the following code. This structure only applies to 8-bit color indexed mode.

```c
struct vis_cmap {
    int index;    /* Index into colormap to start updating */
    int count;    /* Number of entries to update */
    unsigned char *red; /* List of red values */
    unsigned char *green; /* List of green values */
    unsigned char *blue; /* List of blue values */
};
```

The VIS_PUTCMP ioctl command is similar to the FBIOPUTCMAP command. The VIS_PUTCMP command is specific to the frame buffer terminal-emulator compatible console code.
**VIS_GETCMAP**

The terminal emulator calls the **VIS_GETCMAP ioctl** command to retrieve the console color map.

**Implementing Polled I/O in Console Frame Buffer Drivers**

The polled I/O interfaces are implemented as functions in the driver and are called directly by the kernel terminal emulator. The driver passes the address of its polled I/O entry points to the terminal emulator during the execution of the **VIS_DEVINIT ioctl** command. The **VIS_DEVINIT** command is initiated by the terminal emulator.

The **vis_polledio** structure is shown in the following code.

```c
typedef void * vis_opaque_arg_t;
struct vis_polledio {
    struct vis_polledio_arg *arg;
    void (*display)(vis_opaque_arg_t, struct vis_consdisplay *);
    void (*copy)(vis_opaque_arg_t, struct vis_conscopy *);
    void (*cursor)(vis_opaque_arg_t, struct vis_conscursor *);
};
```

The polled I/O interfaces provide the same functionality as the **VIS_CONSDISPLAY**, **VIS_CONSCOPY**, and **VIS_CONSCURSOR ioctl** interfaces. The polled I/O interfaces should follow the same steps that are described above for the respective ioctl commands. The polled I/O interfaces must very strictly adhere to the additional restrictions that are described in the remainder of this section.

The polled I/O interfaces are called only when the operating system is quiesced and in standalone mode. The system enters *standalone* mode whenever the user enters OpenBoot PROM or enters the kmdb debugger, or when the system panics. Only one CPU and one thread are active. All other CPUs and threads are stopped. Timesharing, DDI interrupts, and system services are turned off.

Standalone mode severely restricts driver functionality but simplifies driver synchronization requirements. For example, a user application cannot access the console frame buffer driver by way of the driver’s memory mappings from within a polled I/O routine.

In standalone mode, the console frame buffer driver must *not* perform any of the following actions:

- Wait for interrupts
- Wait for mutexes
- Allocate memory
- Use DDI or LDI interfaces
- Use system services
These restrictions are not difficult to obey since the polled I/O functions are relatively simple operations. For example, when working with the rendering engine, the console frame buffer driver can poll a bit in the device rather than wait for an interrupt. The driver can use pre-allocated memory to render blit data. DDI or LDI interfaces should not be needed.

Frame Buffer Specific Configuration Module

When the driver-specific fbconf( ) module causes a change in resolution or color depth, that fbconf( ) module must send an ioctl to the frame buffer driver. This ioctl triggers the frame buffer driver to call the terminal emulator’s mode change callback function with the new screen size and depth. The frame buffer driver and the terminal emulator must agree about the video mode at all times. When the frame buffer driver and the terminal emulator do not agree about the video mode, the information on the screen is illegible and meaningless.

The X Window System Frame Buffer Specific DDX Module

When the X Window System exits to the command line, the frame buffer’s DDX module must send an ioctl to the frame buffer driver. This ioctl triggers the frame buffer driver to call the terminal emulator’s mode change callback function. This communication keeps the frame buffer driver and the terminal emulator in agreement about the video mode if the X Window System starts and then changes the video resolution before exiting. The frame buffer driver and the terminal emulator must agree about the video mode at all times. When the frame buffer driver and the terminal emulator do not agree about the video mode, the information on the screen is illegible and meaningless.

Developing, Testing, and Debugging Console Frame Buffer Drivers

Debugging a console frame buffer driver on an active system can be problematic.

- Errors that are encountered in the early stages of booting the system do not generate a core dump.
- Error or informative messages might not be displayed correctly on the screen.
- USB keyboard input might fail.

This section offers some suggestions to help you develop, test, and debug console frame buffer drivers.
Testing the I/O Control Interfaces

To test the ioctl commands, create additional ioctl entry points that are callable from a user application. Be sure to copy in the arguments appropriately. Use the ddi_copyin(9F) and ddi_copyout(9F) routines to transfer data to and from user address space. Then write an application to validate rendering, scrolling, and cursor behavior. This way, these ioctl commands do not affect your console while you develop and test the commands.

To ensure that the ioctl commands are working correctly, boot the system and log in. Check whether you get expected behavior when you execute commands such as prstat(1M), ls(1), vi(1), and man(1).

Execute the following script to validate that ANSI color is working correctly:

```
#!/bin/bash
printf \n
\n\n\037;40m Color List \033m\n\n
printf \033;30m Color 30 black\033m
printf \033;31m Color 31 red\033m
printf \033;32m Color 32 green\033m
printf \033;33m Color 33 yellow\033m
printf \033;34m Color 34 blue\033m
printf \033;35m Color 35 purple\033m
printf \033;36m Color 36 cyan\033m
printf \033;37m Color 37 white\033m

printf \033;40m Backlight 40 black \033m
printf \033;41m Backlight 41 red \033m
printf \033;42m Backlight 42 green \033m
printf \033;43m Backlight 43 yellow\033m
printf \033;44m Backlight 44 blue\033m
printf \033;45m Backlight 45 purple\033m
printf \033;46m Backlight 46 cyan \033m
printf \033;47m Backlight 47 white \033m
```

Testing the Polled I/O Interfaces

The polled I/O interfaces are only available under the following circumstances:

- When you enter the OpenBoot PROM by using the L1+A keystroke sequence
- When you boot the system with a standalone debugger such as kmd(1)
- When the system panics

The polled I/O interfaces only become available at a certain point in the boot process. Polled I/O requests issued from the OpenBoot PROM before the system is running are not rendered. Similarly, kmd prompts issued before the console is configured are not rendered.

To test the polled I/O interfaces, enter the OpenBoot PROM by using the L1+A keystroke sequence. To validate that the polled I/O interfaces are being used, type the following command at the OpenBoot PROM ok prompt:

```
ok 1b emit ." [32m This is a test" 1b emit ." [m"
```
The polled I/O interfaces are working properly if the following statements are true:

- The result of the above command is that the phrase *This is a test* is displayed in green.
- The OpenBoot PROM continues to function correctly.
- Scrolling performs as expected.
- The cursor displays correctly.
- The system can be reentered and continued repeatedly.

**Testing the Video Mode Change Callback Function**

To determine whether the video mode change callback function is working properly, log in to the system and use `fbconfig(1M)` to change the resolution and depth of the frame buffer several times. If the console continues to display text properly, the video mode change callback function is working correctly. The kernel terminal emulator might adjust the font size to accommodate different screen sizes, but that is not significant to the console frame buffer driver.

To determine whether the X Window System and the console frame buffer driver interact correctly, switch between the X Window System and the command line several times while modifying the X Window System’s video resolution and the command line resolution in different ways. If the X Window System exits and the console characters are not displayed correctly, either the X Window System did not notify the driver console code that the video mode changed or the driver did not call the kernel terminal emulator’s video mode change callback function.

**Additional Suggestions for Testing Console Frame Buffer Drivers**

During boot, the system sends messages to `/var/adm/messages` if the system fails to locate or successfully load a kernel terminal emulator compatible frame buffer driver. To monitor these messages, type the following command in a separate window:

```
% tail -f /var/adm/messages
```

To avoid problems with USB while debugging the driver, change the EEPROM `input-device` NVRAM configuration parameter to use a serial port instead of the keyboard. See the `eeprom(1M)` man page for more information about this parameter.
This section describes the `pci.conf` file, its uses, and the syntax.

**Description**

The `pci.conf` is introduced to save PCI configurations, such as Number of VFs (Virtual Functions) of a particular PF (Physical Function) on the system. This file has a few purposes:

- To persist the PCI configuration, so VFs can be created automatically upon boot.
- Since the configuration file is part of the `boot_archive`, VFs can be used during the boot.

This file is also used by non-IOV system configurations when VFs are used on bare-metal systems. Currently, it contains only VF-related configurations. In the future, more PCI bus specific configuration and even device specific workarounds may make their way to it. Number of VF configurations is saved in "[System_Configuration]" section, which looks like:

```
[System Configuration]
[[path=<pf_device_path>]]
num-vf=<num_of_vf>
```

**System Configuration Section**

The `[System Configuration]` section of the file is interpreted by the Oracle Solaris PCIe framework. Unrecognized keywords are flagged as errors. There is only one `[System Configuration]` section in the entire file and it must reside at the beginning of the file.

The `[System Configuration]` section is comprised of a series of subsections. Each subsection must have an unique text label, followed by a list of filters enclosed in double square brackets which matches the device(s) of interest. The content of each subsection is a list of actions that the framework will take against each matched device. For example:
[System Configuration]
new_elkg_driver [[id=0x8086,0x1000,,0x108e,]] [[classcode=0x020000]]
num-vf=4

The filter within the double brackets matches all Intel devices with device ID 0x1000, a Sun subsystem vendor ID, and network controller class code in the system. Oracle Solaris sets the device’s number of VFs to 4.

**Note** – Device path can be used in the filter to narrow the scope of the filter to a single device instance.

### Device Configuration Section

The [Device Configuration] section is comprised of the same type of subsections as the [System Configuration] section with the difference that the content is only interpreted by the drivers of the devices that match the filter.

**Device Configuration**

```
# label must be file-globally unique
igbe-sriov-test [[path=/pci@0,0/network@2]]
# number of rx/tx ring pairs for each VF
dma-channel-distribution=2,2,8,4
```

### Syntax

```
/etc/pci.conf = <system section><device section>
<system section> = "[System_Configuration]" {<framework subsection>}*
<device section> = "[Device_Configuration]" {<device subsection>]*
<framework subsection> = <label> [ {filter}]*
[ {<framework action>}+ ]
<device subsection> = <label> [ {filter}]+* {devicenvlist}
<filter> = [ "[" "id" "=" "(vendorid) "," (deviceid) "," (revisionid) "," (subsystem-vendorid) "," (subsystemid)]"]
[ ["classcode" "=" "classcode" ,(,"mask") ] ] [ ["path" "=" "devpath" ] ]
<framework action> = [ "num-vf" "=" "<val>]
<devicenvlist> = [ {name string} "=" "<value string>" ]
```

### References

See also `pci_param_get(9F)`.
Index

Numbers and Symbols
64-bit device drivers, 280, 585

A
add_drv command, 249, 434
description of, 494
device name, 431
address spaces, description of, 57
agent, definition, 222
allocb() function, 443–444
alternate access mechanisms, 566
aphysio() function, 272
aread() entry point, asynchronous data transfers, 269
ASSERT() macro, 491, 532–533
associating kernel memory with user applications, 182
asynchronous communication drivers, testing, 499
asynchronous data transfers
block drivers, 299
character drivers, 269
USB, 441–442
attach() entry point, 437–438, 452–455
active power management, 453
description of, 104–109
network drivers, 383–384, 404
system power management, 455
auto-request sense mode, 327
autoconfiguration
of block devices, 289–290
of character devices, 263
overview, 93
autoconfiguration (Continued)
routines, 42
of SCSI HBA drivers, 347
of SCSI target drivers, 315
Automated System Recovery Unit (ASRU),
definition, 223
autosubvention threshold, 211
autovectored interrupts, 122
avoiding data loss while testing, 509–510
awrite() entry point, asynchronous data
transfers, 269

B
binary compatibility
description of, 58
potential problems, 591
binding a driver to a device, 62
binding a driver to a USB device, 430–432
biodone() function, 295
blitting, 602
block driver
autoconfiguration of, 289
buf structure, 293
cb_ops structure, 96
overview, 44
slice number, 289
block driver entry points, 288
close() function, 292
open() function, 291
strategy() function, 293
booting the kmdb debugger
  on SPARC systems, 513
  on x86 systems, 513
buf structure
  changes to, 588
  description of, 293
buffer allocation, DMA, 165
buffered I/O functions, 579–580
burst sizes, DMA, 164
bus
  architectures, 544
  PCI architectures, 545
  SBus architecture, 549
  SCSI, 307
bus-master DMA, 154, 156
bus-nexus device drivers, description of, 59
byte ordering, 542

c
  cache, description of, 171
callback functions
    description of, 51
    example of, 163
cb_ops structure, description of, 96
cconfig command, 491–492
cfgadm_usb command, 457
character device driver
  physio() function, 272
  autoconfiguration, 263
  cb_ops structure, 96
  close() entry point, 266
  data transfers, 266
  device polling, 276
  entry points for, 262
  I/O control mechanism, 278
  memory mapping, 275
  minphys() function, 273
  open() entry point, 264–265
  overview, 45–46
  physio() function, 271
  strategy() entry point, 274
  checksumming, 392–393, 395, 396
  cloning SCSI HBA driver, 342
close() entry point
  block drivers, 292
  description of, 266
cmn_err() function, 250
  debugging, 532
  description of, 53
  example of, 303
compatible property, description of, 63
compiling drivers, 491–492
condition variable functions, 561–562
  cv_broadcast(), 71
  cv_destroy(), 70
  cv_init(), 70
  cv_timedwait(), 72
  cv_timedwait_sig(), 73
  cv_wait(), 71
  cv_wait_sig(), 73
condition variables
  and mutex locks, 70
  routines, 71
.conf files, See hardware configuration files
configuration, testing device drivers, 503–511
configuration descriptor clouds, 450–451
configuration entry points
  attach() function, 104
  detach() function, 109
  getinfo() function, 110
configuration files, hardware, See hardware configuration files
console frame buffer drivers, 595
  debugging, 606
  kernel terminal emulator, 595
  polled I/O interfaces, 598, 605–606
  standalone mode, 598, 605–606
  video mode change callback interface, 598, 601, 606
  visual I/O interfaces, 597
context management, See device context management
current context of device driver, 52
cookies, DMA, 154
copying data
  copyin() function, 267
  copyout() function, 267
CPR (CheckPoint and Resume), 455
crash command, 512
crash dumps, saving, 510
csi_arq_status structure, changes to, 589
cv_timedwait() function, changes to, 590
cv_timedwait_sig() function, changes to, 590

data alignment for SPARC, 540
data sharing
  using devmap(), 591
  using ioctl(), 591
  using mmap(), 592
data storage classes, 67
data structures
  dev_ops structure, 95–96
  GLDv2, 414, 416–418
  mod1drv structure, 95
data transfers, character drivers, 266
ddi_cb_register() function, 135–137
ddi_cb_unregister() function, 135–137
DDI-compliant drivers
  byte ordering, 542
  compliance testing, 498
ddi_create_minor_node() function, 106
ddi_device_copy() function, 590
ddi_device_zero() function, 591
ddi_devvid_free() function, 239
DDI/DKI
  See also LDI
  design considerations, 50
  and disk performance, 304
  overview, 58
  purpose in kernel, 57
ddi_dma_attr structure, 158, 588
ddi_dma_cookie structure, 588
ddi_dma_getwin() function, 155
ddi_dma_mem_alloc() function, 591
ddi_dma_nextseg() function, 156
ddi_driver_major() function, 289
ddi_enter_critical() function, 551
ddi_eventcookie_t, 240–241
ddi_fm_capable() function, 228
ddi_fm_ereport_post() function, 228–229, 231
ddi_fm_fini() function, 227–228
ddi_fm_init() function, 226–227
ddi_fm_service_impact() function, 232–233
DDI function tables, 557–584
ddi_get_cred() function, 589, 591
ddi_get_driver_private() function, 312, 410
ddi_get_instance() function, 416
ddi_get_lbolt() function, 589
ddi_get_pid() function, 589
ddi_get_time() function, 589
DDI_INFO_DEVT2DEVINFO, 110
DDI_INFO_DEVT2INSTANCE, 110
ddi_intr_add_handler() function, 124, 125, 128
ddi_intr_add_softint() function, 126
ddi_intr_alloc() function, 124, 125, 137–139
ddi_intr_block_disable() function, 125
ddi_intr_block_enable() function, 125
DDI_INTR_CLAIMED, 146
ddi_intr_clr_mask() function, 125, 128
ddi_intr_disable() function, 124, 125
ddi_intr_dup_handler() function, 124, 125
ddi_intr_enable() function, 124, 125
ddi_intr_free() function, 124, 125
ddi_intr_get_cap() function, 125
ddi_intr_get_hilevel_pri() function, 126, 147
ddi_intr_get_nintrs() function, 125
ddi_intr_get_pending() function, 124, 125
ddi_intr_get_pri() function, 126, 147
ddi_intr_get_supported_types() function, 126, 147
ddi_intr_get_supported_types() function, 125
ddi_intr_hilevel() function, 123
ddi_intr_remove_handler() function, 124, 125
ddi_intr_remove_softint() function, 126
ddi_intr_set_cap() function, 125
ddi_intr_set_mask() function, 125, 127
ddi_intr_set_nreq() function, 137–139
ddi_intr_set_pri() function, 126
ddi_intr_set_supported_types() function, 125
ddi_intr_set_supported_types() function, 126
ddi_intr_set_supported_types() function, 123, 126
DDI_INTR_UNCLAIMED, 145
ddi_log_sysevent() function, 85
ddi_model_convert_from() function, 591
ddi_prop_free() function, 242
ddi_prop_get_int() function, 376
Index

ddi_prop_lookup() function, 79
ddi_prop_lookup_string() function, 242
ddi_prop_op() function, 81
ddi_regs_map_setup() function, 116
ddi_removing_power() function, 212
DDI_RESUME, detach() function, 212
ddi_set_driver_private() function, 312
DDI_SUSPEND, detach() function, 212
ddi_umem_alloc() function, 183
ddi_umem_free() function, 186
DDX module, 606
deadman kernel feature, 503
DEBUG symbol, 491, 532–533
debugging
  ASSERT() macro, 532–533
coding hints, 531
common tasks, 516–524
conditional compilation, 533
console frame buffer drivers, 606
DEBUG symbol, 532–533
detecting kernel memory leaks, 518
displaying kernel data structures, 519–521
system file, 506
kbd debugger, 512–515
kmem_flags, 508–509
mdb debugger, 515–516
moddebug, 507–508
postmortem, 512
preparing for disasters, 509
setting up a serial connection, 504
setting up a SPARC test system, 505–506
setting up an x86 test system, 506
system registers, 517–518
tools, 511–524
using kernel variables, 523–524
using the SPARC PROM for device debugging, 552
writing mdb commands, 518
delay() function, 590
  changes to, 590
dependency, 205–206
deprecated device access functions, 575–576
deprecated DMA functions, 572–573
deprecated interrupt functions, 564–565
deprecated memory allocation functions, 561
deprecated power management functions, 577
deprecated programmed I/O functions, 568–571
deprecated property functions, 559–560
deprecated SCSI functions, 582–583
deprecated time-related functions, 576
deprecated user application kernel functions, 575–576
deprecated user process information functions, 575
deprecated user space access functions, 574
deprecated virtual memory functions, 580
descriptor tree, 435–437, 438
dev_adcent argument, ddi_device_copy(), changes to, 590
detach() entry point
  active power management, 453
description of, 109–110
hot removal, 449–450
network drivers, 383–384
system power management, 455
detecting kernel memory leaks with mdb, 518
dev_advcent argument, ddi_device_zero(), changes to, 591
dev_datasz argument, ddi_device_copy(), changes to, 590
dev_datasz argument, ddi_device_zero(), changes to, 591
dev_info_t functions, 558
dev_ops structure, description of, 95–96
dev_t functions, 558–559
devfsadm command, 494
device
  alternate settings, 430
  composite, 432–433, 456
  configurations, 430
  endpoints, 430
  interface number, 456
  interfaces, 430
  quiesce, 42, 141
  resume, 42, 141
  splitting interfaces, 433, 457
device access functions
  block drivers, 291
  character drivers, 264–265
deprecated, 575–576
table, 575–576
Index

device configuration, entry points, 100
device context management, 187
  entry points, 190
  model, 188
  operation, 189
device-dependency, `power.conf` entry, 206
device-dependency-property, `power.conf` entry, 206
device directory, recovering, 510–511
device drivers
  See also compiling drivers
  See also linking drivers
  See also loading drivers
  64-bit drivers, 280, 585
  access from within kernel, 235
  aliases, 494
  binding to device node, 62, 430–432
  bindings, 434
  block driver, 44
  configuration descriptor clouds, 450–451
  context, 52
  debugging, 503–530
    coding hints, 531
    setting up a serial connection, 504
    tools, 511–524
    using the PROM, 552
definition, 39
entry points, 40
error handling, 497
header files, 488
hubd USB hub driver, 449
interface association nodes, 433–434
loadable interface, 97
modifying information with `update_driv`, 494
modifying permissions, 494
module configuration, 489
network drivers, 381–425
offlining, 449
packaging, 495
printing messages, 53
purpose in kernel, 55
source files, 489
standard character driver, 45–46
testing, 496, 503–511
tuning, 524–530
device drivers (Continued)
  USB driver, 427–459
  usb_ia USB interface association driver, 433–434
  usb_mid USB multi-interface driver, 432, 433–434, 449–450, 456
  using kstats, 524–530
device ID functions, 580–581
device information
  binding a driver to a device, 62
  binding a driver to a USB device, 430–432
  compatible device names, 430–432
  di_link_next_by_lnode() function, 252
  di_link_next_by_node() function, 252
  di_link_private_get() function, 253
  di_link_private_set() function, 253
  di_link_spectype() function, 252
  di_link_t, 252
  di_link_to_lnode() function, 252
  di_lnode_devinfo() function, 252
  di_lnode_devt() function, 252
  di_lnode_name() function, 252
  di_lnode_next() function, 252
  di_lnode_private_get() function, 253
  di_lnode_private_set() function, 253
  di_lnode_t, 252
  di_node_t, 252
  di_walk_link() function, 252
  di_walk_lnode() function, 252
  DINFOLOYR, 252
LDI, 239
lnode, 251–253
nblocks property, 594
Nblocks property, 594
property values, 239–240
self-identifying, 544
tree structure, 59
device interrupts, See interrupts; interrupt handling
device layering, See LDI
device memory
  D_DEVMAP flag in cb_ops, 97
  mapping, 47, 177–186
device node, 430
device number, description of, 58
device polling, 145
device polling (Continued)
  in character drivers, 276
  chpoll() function, 276
  poll() function, 276
device power management
  components, 203
  definition of, 201–202
  dependency, 205–206
  entry points, 208
  interfaces, 206
  model, 203
  pm_busy_component() function, 204, 207, 452–455
  pm_idle_component() function, 204, 207, 452–455
  pm_lower_power() function, 205
  pm_raise_power() function, 203–204, 205, 207, 452–455
  pm_power() entry point, 452–455
  pm_power() function, 208
  power() entry points, 204–205
  power() function, 208
  power levels, 204–205
  state transitions, 206
  usb_create_pm_components() function, 452–455
USB devices, 452–455
device registers, mapping, 104
device state in power management, 211
device tree
  displaying, 60
  navigating, in debugger, 521–523
  overview, 59
  purpose in kernel, 56
device usage, 236
  See LDI
/devices directory
  description of, 57
  displaying the device tree, 62
devmap_entry_point, devmap_access() function, 191–192
devmap_entry_points
  devmap_access() function, 199
  devmap_contextmg() function, 193
  devmap_dup() function, 194–195
  devmap() function, 180
  devmap_map() function, 190
  devmap_unmap() function, 195–197
devmap_functions
  devmap_devmem_setup() function, 181
  devmap_load() function, 199
  devmap_umem_setup() function, 185
  devmap_unload() function, 199
diagnosis engine, definition, 222
  .dict dictionary files, 224
  disaster recovery, 510–511
disk
  I/O controls, 304
  performance, 304
disk driver testing, 499
  DDI, See DDI/DMI
  DL_ETHER, network statistics, 412
  DLIOCRAW, ioctl() function, 408
  DLPI (Data Link Provider Interface), See network drivers, GLDv2
  DLPI primitives, DL_GET_STATISTICS_REQ, 410
DMA
  buffer allocation, 165
  burst sizes, 164
  callbacks, 169
  cookie, 154, 155
  freeing handle, 169
  freeing resources, 168–169
  handle, 154, 155, 161
  object, 153
  object locking, 161
  operations, 156–161
  physical addresses, 155
  private buffer allocation, 165–167
  register structure, 163
  resource allocation, 162–164
  restrictions, 157
  transfers, 156, 271–272
  virtual addresses, 155
  windows, 155, 173
DMA functions, 571–573
  deprecated, 572–573
driver binding name, 62
driver.conf files, See hardware configuration files
driver module entry points, See entry points
drv_getparm() function, changes to, 589
drv_usewait(9F), 551
DTrace
  definition, 530
task queues, 92
dump() entry point, block drivers, 303
DVMA
  SBus slots supporting, 550
  virtual addresses, 155
dynamic memory allocation, 53
events
  asynchronous notification, 240–241
  attributes, 86–89
  description of, 83–84
  hotplug notification, 449
Eversholt fault tree (eft) rules, 229
exporting device memory to user applications, 181
external registers, 551

Eeprom(1M) command, 608
EHCI (Enhanced Host Controller Interface), 428
ENA (Error Numeric Association), 228–229
entry points
  attach() function, 104–109, 213–215, 437–438, 452–455
  active power management, 453
  system power management, 455
for block drivers, 288
for character drivers, 262
definition, 40
detach() function, 109–110, 212–213, 453
  hot removal, 449–450
  system power management, 455
for device power management, 208
device context management, 190
for device configuration, 100
for network drivers, 419–423
ioctl() function, 278
power() function, 208–210, 452–455
probe() function, 101–104
quiesce() function, 42
SCSA HBA summary, 334
  system power management, 211
ereport, definition, 222
ereset, definition, 222
error handling, 497
error messages, printing, 53, 303
system file, 506
/etc/driver_aliases file, 434
Ethernet V2, See DL_ETHER
event registry, 224, 229
fault, definition, 222
fault event, definition, 222
fault management
  access or DMA handle error, 232–233
  agent, 222
  Automated System Recovery Unit (ASRU), 223
  DDI_FM_* I/O controller errors, 230–232
  ddi_fm_capable() function, 228
  ddi_fm_ereport_post() function, 228–229, 231
  ddi_fm_fini() function, 227–228
  ddi_fm_init() function, 226–227
  ddi_fm_service_impact() function, 232–233
  DDL_SERVICE_* service impact values, 232–233
diagnosis engine, 222
dict dictionary files, 224
eft diagnosis engine, 230–232
ENA (Error Numeric Association), 228–229
ereset, 222, 226
ereset events, 222, 228–229
error handling, 225–233
event registry, 224, 229, 230–232
Eversholt fault tree (eft) rules, 229
fault, 222
fault event, 222, 225
fault management capabilities, 226
fault management capabilities, declaring, 226–227
fault management capability bit mask, 228
fault management capability properties, 226
fault management resources, cleaning up, 227–228
fault manager daemon fmd, 222–225
fault messages, 224–225
Field Replaceable Unit (FRU), 223
fmadm command, 223–224
fault management (Continued)
  fmdump command, 223
  I/O Fault Services, 221
  interfaces, 577–578
  list suspect, 223, 224–225
  pci_ereport_post() function, 229–230
  pci_ereport_setup() function, 229–230
  pci_ereport_setup() function, 227
  pci_ereport_teardown() function, 228, 229–230
  .po message files, 224
  response agent, 223–224
  retire agent, 223–224
  suspect list, 223
  topology of system, 225
Fault Management Architecture (FMA), See fault management
faults, latent fault, definition of, 536
fbconfig(1M) command, 608
fbconfig() module, 606
FDDI (Fibre Distributed Data Interface), 405–406
fibre distributed data interface, See DL_FDDI
Field Replaceable Unit (FRU), definition, 223
file system I/O, 288–289
_fini() entry point
  example of, 99
  network drivers, 382–383
  required implementation, 41
first-party DMA, 155, 157
flags argument, ddi_dma_mem_alloc(), changes to, 591
flow of control for power management, 216
fmadm command, 223–224
fmd fault manager daemon, 222–225
fmdump command, 223
freemsg() function, 443–444
functions
  See also condition variable functions
  See also device power management
  See individual functions
  See also LDI functions
  See specific function name
fuser command, display device usage
  information, 256–257

G

GCC, 491–492
gcc command, 491–492
generic device name, 63
getinfo() entry point, 110
getmajor() function, 289
getrbuf() function, changes to, 589
getting major numbers, example of, 289
GLD (Generic LAN Driver), See network drivers
gld() entry point, 404
gld() functions, 404
gld_intr() function, 425
gld_mac_alloc() function, 423
gld_mac_free() function, 423
gld_mac_info structure, 404
description of, 414–416
GLDv2 arguments, 418
network drivers, 410
used in gld_intr() function, 425
gld_recv() function, 424–425
gld_register() function, 424
gld_sched() function, 425
gld_stats structure, network driver, 411
gld_unregister() function, 424
gld9(F) function, network driver, 410
gldm_get_stats(), description of, 411
gldm_private structure, 415
GLDv2 data structures
  gld_mac_info, 414–416
  gld_stats, 416–418
GLDv2 entry points
  gldm_get_stats(), 422
  gldm_intr(), 422
  gldm_ioctl(), 423
  gldm_reset(), 419
  gldm_send(), 421–422
  gldm_set_mac_addr(), 420
  gldm_set_multicast(), 420
  gldm_set_promiscuous(), 421
  gldm_start(), 419
  gldm_stop(), 419–420
GLDv2 ioctl functions, 408–409
GLDv2 network statistics, 410–414
GLDv2 service routines
  gld_inr() function, 425
  gld_mac_alloc() function, 423
  gld_mac_free() function, 423
  gld_recv() function, 424–425
  gld_register() function, 424
  gld_sched() function, 425
  gld_unregister() function, 424

GLDv2 symbols
  GLD_BADARG, 423
  GLD_FAILURE, 423
  GLD_MAC_PROMISC_MULTI, 419
  GLD_MAC_PROMISC_NONE, 419
  GLD_MAC_PROMISC_PHYS, 419
  GLD_MULTI_DISABLE, 420
  GLD_MULTI_ENABLE, 420
  GLD_NOLINK, 421
  GLD_NORESOURCES, 425
  GLD_NOTSUPPORTED, 420
  GLD_SUCCESS, 423

graphics devices, device context management of, 187

H
handle, DMA, 154, 161, 169
hardening drivers, 221
hardware checksumming, 392–393, 395, 396
hardware configuration files, 489, 492
  PCI devices, 548
  SBus devices, 550
  SCSI target devices, 311
  where to place, 494
hardware context, 187
hardware state in power management, 211
HBA driver, See SCSI HBA driver
header files for device drivers, 488
high-level mutexes, interrupts, 147
host bus adapter transport layer, 333
hot-plug, See hotplugging
hotpluggable drivers, See hotplugging
hotplugging, 54
  and SCSI HBA driver, 54, 375–376
  USB device, 449–451
hub driver, 428–429
hubd USB hub driver, 449

I
I/O
  asynchronous data transfers, 269, 299
  byte stream, 45
  disk controls, 304
  DMA transfers, 271
  file system structure, 288–289
  miscellaneous control of, 278–283
  multiplexing, 276
  polled I/O interfaces, 598, 605–606
  programmed transfers, 269
  scatter/gather structures, 267
  synchronous data transfers, 269, 295
  visual I/O interfaces, 597
IEEE 802.3, 405
IEEE 802.5, 405–406
ILP32
  use in devmap(), 592
  use in ioctl(), 591
  use in mmap(), 592
ILP64, use in mmap(), 592
  _info() entry point
    example of, 100
    required implementation, 41
  _init() entry point
    network drivers, 382–383
    example of, 99
    required implementation, 41
instance numbers, 101
interface association nodes, 433–434
internal mode registers, 551
internal sequencing logic, 551
interrupt functions, 563–565
interrupt handlers
  functionality, 145–147
  registering, 128
interrupt handling, 121–151
  callback handler function, 135–137
  clearing masks, 128
  ddi_cb_register() function, 135–137
  ddi_cb_unregister() function, 135–137
interrupt handling (Continued)
ddi_intr_add_handler() function, 124, 125, 128
ddi_intr_add_softint() function, 126
ddi_intr_alloc() function, 124, 125, 137–139
ddi_intr_block_disable() function, 125
ddi_intr_block_enable() function, 125
ddi_intr_clr_mask() function, 125, 128
ddi_intr_disable() function, 124, 125
ddi_intr_dup_handler() function, 124, 125
ddi_intr_enable() function, 124, 125
ddi_intr_free() function, 124, 125
ddi_intr_get_cap() function, 125
ddi_intr_get_hilevel_pri() function, 126, 147
ddi_intr_get_nintrs() function, 125
ddi_intr_get_pending() function, 125, 127
ddi_intr_get_pri() function, 126, 147
ddi_intr_get_softint_pri() function, 126
ddi_intr_get_supported_types() function, 125
ddi_intr_hilevel() function, 123
ddi_intr_remove_handler() function, 124, 125
ddi_intr_remove_softint() function, 126
ddi_intr_set_cap() function, 125
ddi_intr_set_mask() function, 125, 127
ddi_intr_set_nreq() function, 137–139
ddi_intr_set_pri() function, 126
ddi_intr_set_softint_pri() function, 126, 127
ddi_intr_trigger_softint() function, 123, 126
gld_intr() function, 425
high-level interrupts, 122, 125, 147
overview, 50
pending interrupts, 127
setting masks, 127
software interrupts, 124, 127, 147
interrupt property, definition, 50
Interrupt Resource Management, 134–145
interrupts
allocating, 137–139
callback support, 135–137
capability functions, 125
changing soft interrupt priority example, 127
checking pending interrupts example, 127
clearing interrupt masks example, 127
common problems with, 551
interrupts (Continued)
description of, 121
handling high-level interrupts examples, 147–151
handling low-level interrupts example, 150
high-level mutexes, 147
initialization and destruction functions, 125
interrupt handling example, 146
legacy defined, 122
message-signaled defined, 122
MSI defined, 122
MSI implementation, 123
MSI-X defined, 122
MSI-X implementation, 124
network drivers, 410
priority levels, 122
priority management functions, 126
registering legacy interrupt example, 129–130
registering legacy interrupts, 128–131
registering MSI interrupts, 131–133
registering MSI interrupts example, 131–133
removing legacy interrupt example, 130–131
removing MSI interrupts example, 133
requesting, 137–139
setting interrupt masks example, 127
soft interrupt functions, 126
software interrupts, 147
types of, 122
using legacy, 123
writing handlers, 121–151
ioctl(9E) driver entry point, 596
ioctl() function
character drivers, 278–280
commands, 593
DLIOCRAW, 408
Iovec structure, 267
IRM, See Interrupt Resource Management
ISO 8802-3, 405
ISO 9314-2, 405–406
ISR (interrupt service routine), 146

K
 KERNEL symbol, 491
kernel debugger
See kmdb debugger
device tree, 56
memory
allocation, 53
associating with user applications, 182
detecting leaks with mdb, 518
module directory, 493–494
overview, 55
kernel data structures, 519–521
kernel logging functions, 579
kernel statistics, See kstats
kernel statistics functions, 578
kernel terminal emulator, 595
kernel thread functions, 561–562
kernel variables
setting, 506–507
use with debuggers, 523–524
using, 506
kmdb debugger, 512–515
booting on SPARC systems, 513
booting on x86 systems, 513
macros, 514–515
setting breakpoints, 513–514
kmdb kernel debugger, 503
kmem_alloc() function, 53
kmem_flags kernel variable, 508–509
kmem_free() function, 239
kstats
See network statistics
definition, 524–530
Ethernet drivers, 527–530
functions, 526, 578
structure members, 524
structures, 525
task queues, 91–92

LDI
ld command, 491–492
LDI, 235–257
definition, 57
device access, 236
device consumer, 235
device information, 236
device layering, 251–257
device usage, 236, 251–257, 256–257
event notification interfaces, 240–241
fuser command, 256–257
kernel device consumer, 235
layered driver, 235
layered driver handle, 237–241, 242–249
layered identifier, 236–237, 242–249
libdevinfo interfaces, 251–257
prtconf command, 253–256
target device, 235, 237–241
LDI functions
ldi_add_event_handler() function, 240–241
ldi_aread() function, 238–239
ldi_awrite() function, 238–239
ldi_close() function, 237–238, 242
ldi_devmap() function, 238–239
ldi_dump() function, 238–239
ldi_get_dev() function, 239
ldi_get_devid() function, 239
ldi_get_eventcookie() function, 240–241
ldi_get_minor_name() function, 239
ldi_get_otyp() function, 239
ldi_get_size() function, 239
ldi_get_msg() function, 238–239
ldi_ident_from_dev() function, 236–237, 242
ldi_ident_from_dip() function, 236–237
ldi_ident_from_stream() function, 236–237
ldi_ident_release() function, 236–237, 242
ldi_ioctl() function, 238–239
ldi_open_by_dev() function, 237–238
ldi_open_by_devid() function, 237–238
ldi_open_by_name() function, 237–238, 242
ldi_poll() function, 238–239
ldi_prop_exists() function, 239–240
ldi_prop_get_int() function, 239–240
ldi_prop_get_int64() function, 239–240

latent fault, definition of, 536
layered driver handle, See LDI
Layered Driver Interface, See LDI
layered identifier, See LDI

L

Index

621
LDI functions (Continued)
  ldi_prop_lookup_byte_array() function, 239–240
  ldi_prop_lookup_int_array() function, 239–240
  ldi_prop_lookup_int64_array() function, 239–240
  ldi_prop_lookup_string_array() function, 239–240
  ldi_prop_lookup_string() function, 239–240
  ldi_putmsg() function, 238–239
  ldi_read() function, 238–239
  ldi_remove_event_handler() function, 240–241
  ldi_strategy() function, 238–239
  ldi_write() function, 238–239, 242

LDI types
  ldi_callback_id_t, 240–241
  ldi_handle_t, 237–241
  ldi_ident_t, 236–237

leaf devices, description of, 59
legacy interrupts
  defined, 122
  using, 123
length argument, ddi_dma_mem_alloc(), changes to, 591
libdevinfo(), displaying the device tree, 61
libdevinfo device information library, 251–257
linking drivers, 491–492
lint command, 64-bit environment, 587
list suspect, definition, 223
lnode, 251–253
loadable module functions, 558
loading drivers, 491–492
  addDrv command, 494
  hardware configuration file, 492
loading modules, 41, 493–494
loading test modules, 507–508
locking primitives, types of, 67
locks
  manipulating, 561–562
  mutex, 68–69
  readers/writer, 69
  scheme for, 74
LP64
  use in devmap(), 592

LP64 (Continued)
  use in ioctl(), 591
lso_basic_tcp_ipv4() structure, 393
LUN bits, 323

M
mac_alloc() function, 383
mac_callbacks MAC entry points structure, 385–386
mac_capab_lso() structure, 393
mac_fini_ops() function, 382–383
mac_hcksum_get() function, 392–393, 395
mac_hcksum_set() function, 392–393, 396
mac_init_ops() function, 382–383
mac_link_update() function, 397
mac_lso_get() function, 393, 395
mac_register() function, 383–384
mac_register MAC registration information structure, 383, 385–386
mac_rx() function, 395–397
mac_tx_update() function, 395, 397
mac_unregister() function, 383–384
major numbers
  description of, 58
  example of, 289
makedevice() function, 289
mapsize argument, rmallocmap(), changes to, 590
mcl_getcapab() entry point, 386–393
mcl_getprop() entry point, 399
mcl_getstat() entry point, 397–398
mcl_propinfo() entry point, 399
mcl_setprop() entry point, 399
mcl_tx() entry point, 394–395
mcl_unicst() entry point, 395–397
mdb
  detecting kernel memory leaks, 518
  writing commands, 518
mdb debugger, 515–516
  navigating device tree with, 521–523
  retrieving soft state information, 523
  running, 515–516
memory allocation, description of, 53
memory allocation functions, 560–561
deprecated, 561
memory leaks, detecting with mdb, 518
memory management unit, description of, 57
memory mapping
device context management of, 187
device memory management, 47, 177–186, 275
memory model
SPARC, 543
store buffers, 543
message-signaled interrupts, defined, 122
minor device node, 105
modifying permissions of, 494
minor numbers, 58
minphys() function, 273
bulk requests, 445
mmap() function, driver notification, 197
mod_install() function, network drivers, 382–383
mod_remove() function, network drivers, 382–383
mod_debug kernel variable, 508
modinfo command, 251, 507–508
mod_drv structure, description of, 95
mod_linkage structure, description of, 95
mod_load command, 507–508
modular debugger, See mdb debugger
module directory, 493–494
module functions, 558
module_info structure, network drivers, 409
mod_unload command, 507–508
description of, 495
mount() function, block drivers, 291
msgb() structure, 445, 447
MSI interrupts
defined, 122
implementation, 123
MSI-X interrupts
defined, 122
implementation, 124
multiplexing I/O, 276
multiprocessor considerations, 189
multithreading
and condition variables, 70
D_MP flag in cb_ops structure, 97
execution environment, 57
and locking primitives, 67
thread synchronization, 70
mutex
functions, 68
locks, 68–69
manipulating, 561
related panics, 74
routines, 68
mutex_enter() function, 122
mutex_exit() function, 122
mutex_init() function, 437
mutex_owned() function, example of, 533
mutual-exclusion locks, See mutex

N
name property, description of, 63
naming
unique prefix for driver symbols, 40, 531–532
Nblocks property, definition, 594
nbblocks property, deprecated, 594
nbblocks property, use in block device drivers, 289
Nblocks property, use in block device drivers, 289
 nbytes argument, uiomove(), changes to, 590
network drivers
attach() entry point, 383–384, 404
detach() entry point, 383–384
DL Ether, 405
DL FDDI, 405–406
DL TPR, 405–406
description of, 495
entry points, 385–386, 399–403
Ethernet V2 packet processing, 405
FDDI (Fibre Distributed Data Interface), 405–406
_init() entry point, 382–383
gld() entry point, 404
gld() functions, 404
gld() functions, 404
gld_mac_info structure, 404, 407–408
gld_register() function, 407–408
GLDv2, 403–425
hardware checksumming, 392–393, 395, 396
IEEE 802.3, 405
IEEE 802.5, 405–406
_init() entry point, 382–383
ISO 8802-3, 405
ISO 9314-2, 405–406
lso_basic_tcp_ipv4() structure, 393
network drivers (Continued)

- **mac_alloc() function**, 383
- **mac_callbacks structure**, 385–386
- **mac_capab_lso() structure**, 393
- **mac_fini_ops() function**, 382–383
- **mac_hcksum_get() function**, 392–393, 395
- **mac_hcksum_set() function**, 392–393, 396
- **mac_init_ops() function**, 382–383
- **mac_link_update() function**, 397
- **mac_lso_get() function**, 393, 395
- **mac_register() function**, 383–384
- **mac_register structure**, 383, 385–386
- **mac_rx() function**, 395–397
- **mac_tx_update() function**, 395, 397
- **MACtypeidentifier**, 385
- **mac_unregister() function**, 383–384
- **MACversionnumber**, 383
- **mc_getcapab() entrypoint**, 386–393
- **mc_getprop() entrypoint**, 399
- **mc_getstat() entrypoint**, 397–398
- **mc_propinfo() entrypoint**, 399
- **mc_setprop() entrypoint**, 399
- **mc_tx() entrypoint**, 394–395
- **mc_unicst() entrypoint**, 395–397
- **mod_install() function**, 382–383
- **mod_remove() function**, 382–383
- **open() entrypoint**, 406
- **SAP (Service Access Point)**, 405
- **SNAP processing**, 405–406
- **source routing**, 406
- **testing**, 499
- **TPR (Token Passing Ring)**, 405–406

network statistics

- **DL_ETHER**, 412
- **gld_stats**, 411
- **gldm_get_stats()**, 411
- **kstat structures**, 410

nexus, See bus nexus device drivers

- **nexus driver**, 428–429
- **no-involuntary-power-cycles property**, 207
- **nvlist_alloc structure, description of**, 87

O

- **object locking**, 161
- **offlining**, 449
- **OHCI (Open Host Controller Interface)**, 428
- **open() entry point**
  - **block drivers**, 291
  - **character drivers**, 264
  - **network drivers**, 406

P

- **packaging**, 495
- **partial store ordering**, 544
- **PCI bus**, 545
  - **configuration address space**, 546
  - **configuration base address registers**, 547
  - **hardware configuration files**, 548
  - **I/O address space**, 547
  - **memory address space**, 547
- **PCI configuration functions, alternate access mechanisms**, 566
- **PCI devices**, 545
  - **pci_ereport_post() function**, 229–230
  - **pci_ereport_setup() function**, 227, 229–230
  - **pci_ereport_teardown() function**, 228, 229–230
- **physical DMA**, 155
- **physio() function, description of**, 271
- **pipe, alternate setting**, 457–458

pipes

- **closing**, 440
- **default control**, 437, 439
- **flushing**, 447–448
- **mutex initialization**, 438
- **opening**, 440
- **policy**, 442
- **USB device communication**, 438–448
- **USB devices**, 430
- **use before attach()**, 435

pixel depth mode, 595

- **pm_busy_component() function**, 452–455
- **pm_idle_component() function**, 452–455
- **pm_lower_power() function**, 453
- **pm_raise_power() function**, 452–455

.po message files, 224
polled I/O interfaces, 598, 605–606
postmortem debugging, 512
power cycle, 207
power() entry point, 452–455
power management
  See also device power management
  See also system power management
  flow of control, 216
  USB devices, 451–455
power management functions, 577
  deprecated, 577
Predictive Self-Healing, 222
  See also fault management
prefix
  unique prefix for driver symbols, 40, 531–532
print() entry point, block drivers, 303
printing functions, 579
printing messages, 53
probe() entry point
  description of, 101–104
  SCSI target drivers, 315
processor issues
  SPARC, 539, 541
  x86, 541
programmed I/O, 269
programmed I/O functions, 565–571
  deprecated, 568–571
PROM commands, 552
prop_op() entry point, description of, 81
properties
  class property, 311
  ddi_prop_op, 81
  device node name property, 63
  LDI, 239–240
  nblocks property, 289
  Nbblocks property, 289
  nblocks property, 594
  Nblocks property, 594
  no-involuntary-power-cycles, 207
  overview, 50, 77
  pm-hardware-state property, 211, 214, 317
  prtconf, 79
  reg property, 211
  removable-media, 206
properties (Continued)
  reporting device properties, 81
  SCSI HBA properties, 376
  SCSI target driver, 377
  size property, 263
  types of, 77
property functions, 559–560
prtconf command
  displaying device names, 430–432
  displaying interfaces, 433
  displaying kernel device usage information, 253–256
  displaying properties, 79
  displaying the bound driver, 431
  displaying the device tree, 61
pseudo device driver, 39

Q
queuing, 379
quiesce a device, 42, 141
quiesce() entry point, 42

R
read() entry point, synchronous data transfers, 269
readers/writer locks, 69
  manipulating, 561
real_length argument, ddi_dma_mem_alloc(), changes to, 591
recovering the device directory, 510–511
reg property, 77
register structure, DMA, 163
removable-media, 206
resource map functions, 583
resume a device, 42, 141
retire agent, definition, 223–224
rmallocmap() function, changes to, 590
rmallocmap_wait() function, changes to, 590
Index

S
S_IFCHR, 106
SAP (Service Access Point), 405
saving crash dumps, 510
SBus
  address bits, 550
  geographical addressing, 549
  hardware configuration files, 550
  physical address space, 549
  slots supporting DVMA, 550
scatter-gather
  DMA engines, 156
  I/O, 267
scheduling tasks, 89–92
SCSA, 308, 332
  global data definitions, 329
  HBA transport layer, 333
  interfaces, 334
SCSI
  architecture, 308
  bus, 307
scsi_functions
  scsi_alloc_consistent_buf() function, 322
  scsi_destroy_pkt() function, 322
  scsi_dmafree() function, 326
  scsi_free_consistent_buf() function, 322
  scsi_ifgetcap() function, 324
  scsi_ifsetcap() function, 324
  scsi_init_pkt() function, 320
  scsi_probe() function, 352
  scsi_setup_cdb() function, 323
  scsi_sync_pkt() function, 322, 326
  scsi_transport() function, 324
  scsi_unprobe() function, 352
  summary, 310
scsi_structures
  scsi_address structure, 338
  scsi_device structure, 338
  scsi_hba_tran structure, 335
  scsi_pkt structure, 339
  scsi_alloc_consistent_buf() function, changes to, 590
  scsi_device structure, 312
SCSI functions (Continued)
  deprecated, 582–583
scsi_hba_functions
  scsi_hba_attach_setup() function, 376
  scsi_hba_lookup_capstr() function, 368
  scsi_hba_pkt_alloc() function, 353
  scsi_hba_functions, scsi_hba_pkt_free() function, 360
scsi_hba_functions
  scsi_hba_probe() function, 352
  summary list, 343
SCSI HBA driver
  abort and reset management, 373
  autoconfiguration, 347
  capability management, 368
  cloning, 342
  command state structure, 344
  command timeout, 367
  command transport, 362
  configuration properties, 376
  data structures, 335
  DMA resources, 356
  driver instance initialization, 352
  entry points summary, 334
  header files, 344
  and hotplugging, 54, 375–376
  initializing a transport structure, 348
  installation, 376
  interrupt handling, 364
  overview, 332–333, 333
  properties, 378
  resource allocation, 353
SCSI HBA driver entry points
  by category, 351
  tran_abort() function, 373
  tran_dmafree() function, 361
  tran_getcap() function, 368
  tran_init_pkt() function, 353
  tran_reset() function, 373
  tran_reset_notify() function, 374
  tran_setcap() function, 370
  tran_start() function, 362
  tran_sync_pkt() function, 361
  tran_tgt_free() function, 353

626  Writing Device Drivers • October 2012
SCSI HBA driver entry points (Continued)
  tran_tgt_init() function, 352
  tran_tgt_probe() function, 352
scsi_hba_tran structures, scsi_pkt structure, 341
scsi_pkt structure, 313
  changes to, 589
SCSI target driver
  auto-request sense mode, 327
  autoconfiguration of, 315
  building a command, 323
  callback routine, 325
  data structures, 311
  initializing a command descriptor block, 323
overview, 307
properties, 311, 317, 377
resource allocation, 320
reusing packets, 326
SCSI routines, 310
  transporting a command, 324
segmap() entry point
  description of, 177, 275
  driver notification, 197
self-identifying devices, 544
serial connection, 504
serviceability
  add new device, 536
  detect faulty device, 536
  perform periodic health checks, 536
  remove faulty device, 536
  report faults, 536
single device node, 430
size property, 263
slice number for block devices, 289
SNAP (Sub-Net Access Protocol), 405–406
snoop command, network drivers, 408
snooping kernel variable, 503
soft interrupts, 124
soft state information
  LDI, 242–249
  retrieving in mdb, 523
  USB, 438
software interrupts, changing priority, 127
software state functions, 560
Solaris kernel, See kernel
source compatibility, description of, 58
source files for device drivers, 489
SPARC processor
  byte ordering, 540
  data alignment, 540
  floating point operations, 539
  multiply and divide instructions, 541
  register windows, 541
  structure member alignment, 540
special files, description of, 57
src_advcnt argument, ddi_device_copy(), changes to, 590
standalone mode, 597, 605
state structure, 51, 105, 242–249
storage classes, driver data, 67
store buffers, 543
strategy() entry point
  block drivers, 293
  character drivers, 274
streaming access, 166
STREAMS
  See network drivers, GLDv2
  cb_ops structure, 97
drivers, 46
Sun Studio, 491–492
suspect list, definition, 223
synchronous data transfers
  block drivers, 295
  character drivers, 269
USB, 441–442
system calls, 55
system global state functions, 583
system power management
  description of, 202
  entry points, 211
  model, 210
  policy, 211
  saving hardware state, 211
USB devices, 455
system registers, reading and writing, 517–518

T
tagged queuing, 379
tapedrivers, testing, 498

task queues, 89–92
  definition, 90
  interfaces, 90, 562

tem (terminal emulator module), 595–596

See also kernel terminal emulator
tem-support DDI property, 595, 596

test modules, 506

testing
  asynchronous communication drivers, 499
  configurations, 496
  console frame buffer drivers, 606
  DDI compliance, 498
  device drivers, 496
  disk drivers, 499
  functionality, 496
  installation and packaging, 498
  network drivers, 499
  tape drivers, 498–499

testing debuggers, avoiding data loss, 509–510

testing device drivers, 503–511

third-party DMA, 154, 157

thread synchronization
  condition variables, 70–72
  mutex_init, 68
  mutex locks, 68–69
  per instance mutex, 104
  readers/writer locks, 69

threads
  preemption of, 67
  task queues, 89–92

*ticks argument, delay(), changes to, 590
*ticks argument, timeout(), changes to, 590

time-related functions, 576
  deprecated, 576

timeout argument, cv_timedwait(), changes to, 590

timeout() function, 590
  changes to, 590

tip connection, 504

total store ordering, 544

TPR (Token Passing Ring), 405–406

tran_abort() entry point, SCSI HBA drivers, 373
tran_destroy_pkt() entry point, SCSI HBA drivers, 360
tran_dmafree() entry point, SCSI HBA drivers, 361
tran_getcap() entry point, SCSI HBA drivers, 368
tran_init_pkt() entry point, SCSI HBA drivers, 353
tran_reset() entry point, SCSI HBA drivers, 373
tran_reset_notify() entry point, SCSI HBA drivers, 374
tran_setcap() entry point, SCSI HBA drivers, 370
tran_start() entry point, SCSI HBA drivers, 362
tran_sync_pkt() entry point, SCSI HBA drivers, 361

tuning device drivers, 524–530
  DTrace, 530
  kstats, 524–530

U

UHCI (Universal Host Controller Interface), 428

uiomove() example, 269

uiomove() function
  changes to, 590
  example of, 269

unloading drivers, 495

unloading test modules, 507–508

untagged queuing, 379

update_drv command, 251, 434
  description of, 494

USB device
  alternate settings, 430
  compatible device names, 430–432
  composite, 432–433, 456
  configuration descriptors, 435–437
  current configuration, 430
  endpoints, 430
    bulk, 438–439
    control, 438–439
    default, 439
    interrupt, 438–439
    isochronous, 438–439
  hotplugging, 449–451
    callbacks, 449
    insertion, 449
    reinsertion, 450–451
    removal, 449–450
  interface number, 456
  interfaces, 430
USB device (Continued)
    multiple configurations, 430
    power management, 451–455
       active, 453–455
       device, 452–455
       passive, 455
       system, 455
    remote wakeup, 452
    splitting interfaces, 433, 457
    states, 448–456
USB drivers, 428–429
    asynchronous transfer callbacks, 441
    bulk data transfer requests, 445
    control data transfer requests, 444–445
    data transfer
       callback status flags, 441, 444
       completion reasons, 441, 444
    data transfer requests, 442–447
    descriptor tree, 435–437, 438
    event notification, 449
    hubd USB hub driver, 449
    interfaces, 428
    interrupt data transfer requests, 445–446
    isochronous data transfer requests, 446–447
    message blocks, 443–444
    mutex initialization, 437
    pipes, 430, 438
       closing, 440
       default control, 435, 437, 439
       flushing, 447–448
       opening, 440
    registering, 437–438
    registering for events, 449
    set alternate, 457–458
    set configuration, 457
    synchronous control requests, 444–445
    usb_iad USB interface association driver, 433–434
    usb_miad USB multi-interface driver, 432, 433–434, 449–450, 456
    versioning, 437
USB functions
    cfgadm_usb command, 457
    usb_alloc_bulk_req() function, 442–443
    usb_alloc_ctrl_req() function, 442–443

USB functions (Continued)
    usb_alloc_intr_req() function, 442–443
    usb_alloc_isoc_req() function, 442–443
    usb_client_attach() function, 437–438
    usb_client_detach() function, 438
    usb_clr_feature() function, 458
    usb_create_pm_components() function, 452–455
    usb_free_bulk_req() function, 442–443
    usb_free_ctrl_req() function, 442–443
    usb_free_descr_tree() function, 438
    usb_free_dev_data() function, 438
    usb_free_intr_req() function, 442–443
    usb_free_isoc_req() function, 442–443
    usb_get_addr() function, 459
    usb_get_alt_if() function, 457–458
    usb_get_cfg() function, 457
    usb_get_current_frame_number() function, 446
    usb_get_dev_data() function, 435–437, 437–438, 439
    usb_get_if_number() function, 456
    usb_get_max_pkts_per_isoc_request() function, 446
    usb_get_status() function, 458–459
    usb_get_string_descr() function, 458
    usb_handle_remote_wakeup() function, 452, 453
    usb_lookup_ep_data() function, 437, 440
    usb owns_device() function, 456
    usb_parse_data() function, 435–437
    usb_pipe_bulk_xfer() function, 440–447
    usb_pipe_close() function, 440, 447
    usb_pipe_ctrl_xfer() function, 440–447
    usb_pipe_ctrl_xfer_wait() function, 443, 444–445
    usb_pipe_drain_reqs() function, 447–448
    usb_pipe_get_max_bulk_transfer_size() function, 445
    usb_pipe_get_private() function, 458
    usb_pipe_get_state() function, 440, 447–448
    usb_pipe_intr_xfer() function, 440–447
    usb_pipe_set_private() function, 458
    usb_pipe_set_private() function, 458

629
USB functions (Continued)

usb_pipe_stop_intr_polling() function, 443, 445–446
usb_pipe_stop_isoc_polling() function, 443, 447
usb_print_descr_tree() function, 438
usb_register_hotplug_cbs() function, 449
usb_set_alt_if() function, 457–458
usb_set_cfg() function, 457
usb_unregister_hotplug_cbs() function, 449
usb_ia USB interface association driver, 433–434
usb_midi USB multi-interface driver, 432, 433–434, 449–450, 456

USB structures

usb_alloc_intr_request, 445–446
usb_bulk_request, 442–443, 445
usb_callback_flags, 441, 444
usb_completion_reason, 441, 444
usb_ctrl_request, 442–443, 444–445
usb_intr_request, 442–443
usb_isoc_request, 442–443, 447
usb_request_attributes, 443

USB 2.0 specification, 427
USBA (Solaris USB Architecture), 427–459
USBA 2.0 framework, 427–459
user application kernel functions

deprecated, 575–576
table, 575–576

user process event functions, 574
user process information functions, 574–575
deprecated, 575
user space access functions, 573–574
deprecated, 574
utility functions, table, 583–584

virtual memory

address spaces, 57
memory management unit (MMU), 57

virtual memory functions

deprecated, 580
table, 580

visual I/O interfaces, 597

volatile keyword, 534

W

windows, DMA, 173

write() function

synchronous data transfers, 269
user address example, 266

X

x86 processor

byte ordering, 541
data alignment, 541
floating point operations, 541

V

/var/adm/messages file, 608
VGA text mode, 595
vgatext module, 595–596
video mode, 596–597, 597, 598, 606
virtual addresses, description of, 57
virtual DMA, 155