Legal Notices

The information in this document is subject to change without notice.

Hewlett-Packard makes no warranty of any kind with regard to this manual, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be held liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Warranty

A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Sales and Service Office.

U.S. Government License

Proprietary computer software. Valid license from HP required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

Copyright Notice

Copyright © 1999–2004 Hewlett-Packard Development Company L.P. All rights reserved. Reproduction, adaptation, or translation of this document without prior written permission is prohibited, except as allowed under the copyright laws.

Trademark Notices

UNIX® is a registered trademark in the United States and other countries, licensed exclusively through The Open Group.

Ethernet is a registered trademark of Xerox Corporation.

All other product names are trademarks, registered trademarks, or service marks of their respective owners.

Printing History

New editions of this manual will incorporate all material updated since the previous edition. The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (minor corrections that are incorporated at a reprinting do not cause the date to change.) The manual part number changes when extensive technical changes are incorporated.

Edition/Part Number/Date

First/B2355-90669/September 1999
Second/B2355-90727/August 2000
Third/B2355-90728/November 2000
Fourth/B2355-90760/July 2002
Fifth/5187-4556/February 2004
## 1. Overview of the Driver Environment

- How the I/O Subsystem is Structured ................................................. 26
- General I/O System (GIO) ................................................................. 26
- Context-Dependent I/O Modules (CDIOs) ............................................. 27

## 2. HP-UX I/O Subsystem Features

- General Driver Topics .................................................................................. 32
- Overview of Driver Types ............................................................................. 33
  - Device, Pseudo, and Interface Drivers ....................................................... 33
  - Block and Character Devices .................................................................... 33
  - Features of Block Devices .......................................................................... 34
  - Features of Character Device Drivers ....................................................... 34
- Major and Minor Numbers ........................................................................... 35
  - Assigning Major Numbers ......................................................................... 35
  - Using Minor Numbers ............................................................................... 35
  - Device-Special Files .................................................................................. 35
  - Device Switch Tables .................................................................................. 36
- System Calls .................................................................................................. 37
  - Two Levels of a Driver .............................................................................. 37
  - Flow of an I/O Request .............................................................................. 37
  - Sample I/O Read Request ......................................................................... 39
- Kernel Data Structures Used for I/O ............................................................ 40
  - buf Structure ............................................................................................ 40
  - isc_table_type Structure .......................................................................... 40
  - I/O Switch Tables ....................................................................................... 40
  - uio Structure ............................................................................................. 41
  - iovec Structure .......................................................................................... 41
- Timeout Mechanisms ..................................................................................... 42
  - How Timeouts Work .................................................................................. 42
  - timeout() Routine ...................................................................................... 42
  - timeout() Routine ...................................................................................... 43
- Interrupt Handling ........................................................................................ 44
  - Software Trigger Mechanism ..................................................................... 45
- Memory Allocation and Mapping I/O Space .................................................. 47
  - Memory Allocation Services ...................................................................... 47
  - Memory Mapped Registers Services .......................................................... 48
  - Configuration Space Accesses .................................................................... 48
  - I/O Port Access Services .......................................................................... 49
- Cache Coherence .......................................................................................... 50
  - Driver Requirements for Coherency ........................................................... 51
  - Rules For Using dma_sync_IO ................................................................. 51
- DMA Mapping ............................................................................................... 52
- I/O Adapters ................................................................................................. 53
- WSIO Mapping Services .............................................................................. 54
- Programming Considerations ....................................................................... 55
- WSIO Interface Registration Services .......................................................... 56
4. Writing a Driver

Suggested Driver Writing Methodology ................................................................. 82
Writing a Driver: Step-by-Step ................................................................. 82
Step 1: Choosing a Driver Name ................................................................. 83
Step 2: Choosing System Header Files ................................................................. 84
System-Defined Header Files ................................................................. 84
Header Files for Tape Drivers ................................................................. 85
Step 3: Defining Installation Structures ................................................................. 86
The drv_ops_t Structure Type ................................................................. 86
The drv_info_t Structure Type ................................................................. 88
The wsio_drv_data_t Structure Type ................................................................. 89
The wsio_drv_info_t Structure Type ................................................................. 90
Sample Header for a Device Driver ................................................................. 90
Sample Header for an Interface Driver ................................................................. 91
Sample Header for a Pseudo Driver ................................................................. 92
Step 4: Identifying Routines for Your Driver ................................................................. 94
Choosing Needed Routines ................................................................. 95
Step 5: Writing Configuration Routines ................................................................. 97
5. Installing Your Driver

Step 1: Compile Your Driver .......................................................... 150
  Compiling a Driver ................................................................. 150
Step 2: Create a Master File ......................................................... 152
  Creating a Master File for Your Driver .................................... 152
Step 3: Modify the System File ...................................................... 156

Support for PCI OLAR .......................................................... 144
  Registering a Driver Handler .................................................. 144
  Registering a Driver Event Mask .......................................... 145
  Setting Event Timeout Values ............................................... 146
  Driver Event Handler .......................................................... 147

New WSIO Services for 11i .................................................... 136
  Register Services ................................................................. 137
  Configuration Space Services ............................................... 138
  I/O Port Space Services ...................................................... 138
  Endian Services ................................................................. 138
  DMA Services ................................................................. 139
  Interrupt Services ............................................................. 140
  Description Service ........................................................... 140
  Ordered Interrupts Service .................................................. 141
  I/O Synchronization Service ................................................. 141
  System Attribute Services ................................................... 141
  Driver Event Handling Services ........................................... 142
  Memory Allocation Services ................................................ 142

Interrupt-Driven Routines for Device Drivers ................................ 135
  Writing a driver_isr() Routine ............................................... 132
  Writing a driver_psizet() Routine .......................................... 134
  Writing a driver_close() Routine .......................................... 114
  Writing a driver_open() Routine ........................................... 112
  Writing a driver_read() or driver_write() Routine .................... 117
  Writing a driver_ioctl() Routine .......................................... 120
  Writing a driver_minor_build() routine .................................. 111
  Writing a driver_select() Routine ....................................... 126
  Writing a driver_strategy() Routine for a Block Device .............. 128
  Writing a driver_ioctl() Routine .......................................... 120
  Writing a driver_read() or driver_write() Routine .................... 117
  Writing a driver_close() Routine .......................................... 114
  Writing a driver_open() Routine ........................................... 112

Configuration Overview ....................................................... 97
  Writing a driver_install() Routine ........................................ 98
  Writing a driver_attach() Routine ........................................ 100
  Writing a driver_dev_init() Routine ...................................... 102
  Writing a driver_if_init() Routine ........................................ 102
  Writing Driver Probe Routines ............................................. 103
  Writing a driver_minor_build() routine .................................. 111
  Writing Entry Point Routines ............................................... 112
  Writing a driver_open() Routine ........................................... 112
  Writing a driver_close() Routine ........................................ 114
  Writing a driver_read() or driver_write() Routine .................... 117
  Writing a driver_ioctl() Routine ........................................... 120
  Writing a driver_minphys() Routine ....................................... 125
  Writing a driver_select() Routine ....................................... 126
  Writing a driver_strategy() Routine for a Block Device .............. 128
  Writing Other Driver Routines ............................................. 132
  Writing a driver_isr() Routine ............................................... 132
  Writing a driver_psizet() Routine .......................................... 134
  Safeguarded-Page Area Service ........................................... 133
  Writing a driver_get_page_as() Routine ................................... 133
  Writing a driver_set_page_as() Routine ................................... 133
  Writing a driver_get_page() Routine ..................................... 133
  Writing a driver_set_page() Routine ..................................... 133
  Writing a driver_ensure_page() Routine .................................. 133

Contents
Contents

6. Creating Networking Device Drivers
   HP-UX Networking Interface Architecture ................................................................. 162
   Data Link Interface Layer ........................................................................................... 163
   Network Protocol Layer ............................................................................................. 163
   Protocol Interface Layer ............................................................................................. 163
   STREAMS Environment ............................................................................................... 164
   Networking Driver Structure ..................................................................................... 165
   STREAMS Device Drivers ............................................................................................ 167
   Data Structures and Interfaces.
     hw_if_type Structure .................................................................................................. 168
     hw_dlpi Structure ....................................................................................................... 170
   Protection and Synchronization
     hw_if_type Structure Protection ............................................................................. 171
     Driver Structure Protection ....................................................................................... 171
   Initializing Networking Device Drivers
     Calling driver_install() ............................................................................................ 174
     Calling driver_attach() ............................................................................................. 175
   Protocol Binding and Demultiplexing.
     Protocol Binding ........................................................................................................ 179
     Protocol Demultiplexing ............................................................................................. 179
     Packet Protocol Format ............................................................................................. 180
   Promiscuous Inbound and Outbound .......................................................................... 181
   Message Block and Queue Functions ........................................................................... 182
   DLPI Interface ............................................................................................................... 183
   Device Information ........................................................................................................ 183
   IP and ARP Configuration ........................................................................................... 184
   STREAMS DLPI Overview ............................................................................................ 185
   Device/DLPI Driver Synchronization ............................................................................ 186
   STREAMS Synchronization ........................................................................................... 187
   Entering STREAMS from ICS ....................................................................................... 187
   Driver Support for DLPI ............................................................................................... 187
   Major Data Structures ................................................................................................. 187
   Opening and Closing a Driver ..................................................................................... 189
   Control Functions ........................................................................................................ 190
   IOCTL Processing ......................................................................................................... 191
   Transmission of Message Blocks .................................................................................. 192
   Reception of Message Blocks ....................................................................................... 192
   DLPI Primitives and IOCTLs ....................................................................................... 192
   Driving the NIC .............................................................................................................. 194
   Data Structures ............................................................................................................ 194
   Control Functions ......................................................................................................... 197
   Datapath ....................................................................................................................... 199
   Interrupt Service Routine — enet_isr() ................................................................. 204
Contents

10. CKO and Transport IOCTLs
    DLPI IOCTL .......................................................... 262
    DLPI_IOC_DRIVER_OPTIONS ........................................ 262
    DL_IOC_HDR_INFO .................................................. 264
    Checksum Offload .................................................. 265
    Adding Checksum Offload Support ............................... 265
    Packet Flow Diagram .............................................. 269

11. Mass Storage Stack Architecture
    Overview of HP-UX Mass Storage I/O Stack ...................... 280
    SCSI Addressing Paradigm ....................................... 281

12. Writing a SCSI Interface Driver
    External Interfaces to a Transport Driver ...................... 284
    Entry Points ...................................................... 284
    Request Entry Point Details .................................. 285
    Service Functions ............................................... 286
    External Data Structures ....................................... 303
    SCSI Control Block .............................................. 303
    ISC .............................................................. 307
    Transport Driver Switch ....................................... 309
    struct buf Structure ............................................ 310
    Logical Unit Structure ......................................... 312
    Target Structure ................................................ 313
    Bus Structure .................................................... 315
    select_q Structure .............................................. 316
    Data Structure Diagram ......................................... 316
    I/O Path .......................................................... 318
    I/O Within SCSI Device Driver ................................ 319
    I/O Within SCSI Service Layer ................................ 320
    I/O Within SCSI Interface Layer ............................... 320
    Transport Driver Development .................................. 321
    Driver Installation and Initialization ......................... 321
    Setting SCSI Parameters ....................................... 325
    SDTR/WDTR Negotiation ......................................... 326
    Register Mapping and DMA ..................................... 327
    Protection and Synchronization ................................ 328
    Sending SCSI I/O Requests to the Target ..................... 329
    Processing Completed I/O Requests ............................ 330
    Interrupt Processing ............................................ 332
    Special Routines ................................................ 332
    Sample Driver .................................................... 335
    Description of the Device ..................................... 335
    Driver Architecture ............................................. 335
    Data Structures .................................................. 336
    qlisp_multi Driver ............................................... 336
Contents

Routines ................................................................. 338
qlisp_multi_dev_probe .............................................. 338
qlisp Driver ............................................................ 340
qlisp_init_pci_cfg .................................................... 340
qlisp_init_reset_card ................................................ 340
qlisp_init_attributes ............................................... 341
qlisp_init_alloc_queues ............................................. 341
qlisp_init_interrupts ............................................... 341
qlisp_init_firmware ................................................. 341
qlisp_pdk_load_nvram_values() .................................... 341
qlisp_pdk_nvram_read() ............................................. 341
qlisp_pdk_nvram_send_bits() ...................................... 341
qlisp_pdk_init_common() ............................................ 341
qlisp_init_common .................................................. 341
qlisp_init_ifsw ...................................................... 342
qlisp_init_start_queues ............................................ 342
qlisp_init_master .................................................... 342
qlisp_init_slave ..................................................... 342
qlisp_pci_attach .................................................... 342
qlisp_mailbox_cmd .................................................. 342
qlisp_build_cmd (and cmd 64 for __LP64__) ...................... 342
qlisp_iocb_cnt ....................................................... 343
qlisp_get_req_iocb .................................................. 343
qlisp_if_start ....................................................... 343
qlisp_fast_complete ................................................ 344
qlisp_post_reset_delay ............................................. 344
qlisp_call_cbfns .................................................... 344
qlisp_process_rsp ................................................... 345
qlisp_isr ............................................................. 345
qlisp_if_lun_open ................................................... 346
qlisp_if_abort ....................................................... 346
qlisp_if_bdr ........................................................ 346
qlisp_reset_bus ..................................................... 346
qlisp_if_reset_bus .................................................. 346
qlisp_send_marker .................................................. 346

13. Writing SCSI Device Drivers
   Step 1: Include Header Files .................................... 348
      General Header Files ......................................... 348
      Header Files for SCSI Drivers ............................... 348
      Header Files for Device Classes ............................ 348
   Step 2: Set Up Structures ...................................... 349
   Step 3: Create the driver_install Routine .................... 351
   Step 4: Create the driver_dev_init() Routine ............... 352
      Setting up the Device Switch Table (scsi_ddsw) .......... 352
   Step 5: Analyze Multiprocessor Implications ................ 355
Contents

Step 6: Create the Entry Point Routines .................................................. 356
   driver_open() ................................................................. 356
   driver_close() ............................................................. 356
   driver_read() ............................................................... 356
   driver_write() .............................................................. 356
   driver_strategy() .......................................................... 357
   driver_psize() ............................................................... 357
   driver_ioctl() ............................................................... 357
Step 7: Error Handling ................................................................. 358
Step 8: Underlying Routines .......................................................... 360
   dd_close ................................................................. 361
   dd_ioctl ................................................................. 362
   dd_ioctl_okay ............................................................ 364
   dd_open ................................................................. 365
   dd_pass_thru_done ...................................................... 367
   dd_pass_thru_okay ....................................................... 368
   dd_read ................................................................. 368
   dd_start ................................................................. 369
   dd_strategy .............................................................. 372
   dd_write ................................................................. 374
Synchronization in SCSI Drivers ................................................. 375
Rules for Ordering Locks ......................................................... 375
Handling Concurrency for Tape Devices .................................. 376
SCSI Services Summary ........................................................ 377

14. Writing PCI Device Drivers
   PCI Overview ....................................................................... 380
   PCI Register Spaces ....................................................... 380
   PCI Transaction Ordering .................................................. 381
   PCI Endian Issues ............................................................ 383
   PCI Device Setup ............................................................ 385
      Mapping Base Address Registers into PCI Memory and I/O Space. ... 385
      Using the Base Address Registers ........................................ 385
      Automatic IRQ Determination ............................................ 386
      Mapping the Memory Base Register ................................... 386
      PCI Configuration Space Restrictions ................................ 387
   PCI Device Operation ........................................................ 389
      The PCI_LITTLE_ENDIAN_ONLY Flag .................................. 389
      Direct Memory Access (DMA) ............................................. 390
   PCI Masters and Coherence .................................................. 391
   Leveraging Existing Drivers ................................................. 392
      Multibus Drivers ........................................................... 392
   PCI Services Summary ....................................................... 393
   Unsupported WSIO Functions ............................................... 394
   Multiprocessor (MP) Safety .................................................. 395
   Constants and Data Structures ............................................ 396
## Contents

User Visible PCI-Specific Data Structures ............................................. 396
Defined Constants .................................................................................... 396
A Sample PCI Driver ................................................................................. 397
Sample WSIO Setup and Structures .......................................................... 397
Sample WSIO Routines ............................................................................ 399
Other Driver Entry Point Routines ............................................................ 402

### 15. On-Line Addition/Replacement
- Introduction ............................................................................................. 404
- Supported Hardware and Software ......................................................... 404
- Overview of Driver Modifications Required for OLA/R .......................... 405
- WSIO Interfaces ...................................................................................... 406
  - `wsio_drv_info_t` .................................................................................. 406
  - WSIO Event Handling Structures ......................................................... 406
- Driver Modifications for OLA/R Support ................................................ 408
  - Driver Registering Its Event Handler .................................................. 408
  - Driver Registering Its Event Mask ....................................................... 408
  - Driver Event Timeout Values .............................................................. 409
  - Event Handling Function ...................................................................... 411
- Handling OLA/R ..................................................................................... 412
  - OLA ..................................................................................................... 412
  - OLR ..................................................................................................... 413
- Miscellaneous Changes Required at the Driver ....................................... 418
- Performing OLA/R of PCI I/O Cards ....................................................... 419

### 16. Writing a DLKM Driver
- Managing Dynamically Loadable Kernel Modules ................................... 422
  - DLKM Concepts .................................................................................... 422
  - DLKM Tools ......................................................................................... 429
  - DLKM Procedures for Dynamically Configured Loadable Modules ......... 433
  - DLKM Procedures for Statically Configured Loadable Modules ............ 440
- Developing Dynamically Loadable Kernel Modules .................................. 443
  - Module Component Files ..................................................................... 443
  - Initializing and Terminating DLKM Modules ....................................... 448
  - WSIO Interface Drivers ....................................................................... 451
  - DLKM Module Development Process .................................................. 456
- Sample DLKM WSIO Class Driver .......................................................... 460
- Sample Component File .......................................................................... 462

### 17. How to Make Pre 11.0 Drivers 64-Bit Safe
- The 64-bit Data Model ............................................................................ 472
- LP64 Considerations ................................................................................. 472
- General Guidelines .................................................................................. 474
- 64-bit Clean Headers ............................................................................... 474
- Useful Data Types .................................................................................... 474
- Hardware Considerations ........................................................................ 476
Step 3: Writing a PSF .............................................................. 535
Step 4: Writing Control Scripts ............................................. 541
Step 5: Packaging the Components ...................................... 550
Step 6: Registering the Depot ............................................. 552
Installing a Depot .............................................................. 553
Command Usage ............................................................... 553
Select Source ................................................................. 553
Select Software ............................................................... 553
Analysis ........................................................................... 554
Installation ...................................................................... 554
Basic Operation .............................................................. 554
Target Selection .............................................................. 554
command_options ............................................................. 555
Managing the Depot Software ............................................. 556
swlist .............................................................................. 556
swcopy ......................................................................... 556
Removing the Installed Software ......................................... 557
Command Messages Logging .............................................. 557
Examples ........................................................................ 558
Creating a Package .......................................................... 558
Control Scripts Files ......................................................... 559
Postinstall ...................................................................... 559
Commands ................................................................. 560
Glossary ............................................................................. 563
Index .................................................................................. 571
Table 2-1. New WSIO Interrupt Services ......................................................... 44
Table 2-2. Memory Allocation Services .......................................................... 47
Table 2-3. Memory Mapped Register Services .................................................. 48
Table 2-4. Configuration Space Accesses .......................................................... 48
Table 2-5. I/O Port Access Services ................................................................. 49
Table 2-6. WSIO DMA Mapping Services .......................................................... 54
Table 2-7. Creating Attributes ....................................................................... 57
Table 2-8. Modifying Attributes ..................................................................... 58
Table 4-1. Device Driver Fields, drv_ops_t Structure ........................................ 87
Table 4-2. Configuration and Entry Point Routines ......................................... 94
Table 4-3. Block Driver ................................................................................. 95
Table 4-4. Character Driver ........................................................................... 96
Table 4-5. Driver Probe Parameters ............................................................... 104
Table 4-6. Address Probe Parameters ............................................................. 105
Table 4-7. Register Services ........................................................................... 137
Table 4-8. Configuration Space Services ........................................................ 138
Table 4-9. I/O Port Space Services .................................................................. 138
Table 4-10. Endian Services ........................................................................... 139
Table 4-11. DMA Services ............................................................................. 139
Table 4-12. Interrupt Services ......................................................................... 140
Table 4-13. Description Service ...................................................................... 140
Table 4-14. Ordered Interrupts Service ........................................................... 141
Table 4-15. I/O Synchronization Service ........................................................ 141
Table 4-16. System Attribute Services ............................................................ 141
Table 4-17. Driver Event Handling Services ................................................... 142
Table 4-18. Memory Allocation Services ........................................................ 143
Table 6-1. Protocol Kind and Value ................................................................. 179
Table 6-2. Promiscuous Mode Matrix .............................................................. 181
Table 6-3. lanscan Output ............................................................................. 184
Table 6-4. enet_dlpi_data_t Data Fields .......................................................... 188
Table 6-5. Message Service Functions ............................................................. 190
Table 6-6. DLPI Primitives and IOCTLs ......................................................... 193
Table 6-7. enet_ift Data Fields ...................................................................... 195
Table 6-8. enlan_ift Data Fields .................................................................... 196
Table 6-9. Bound SAP Data Fields ................................................................. 197
Table 8-1. Utility Routines ............................................................................. 240
Table 8-2. Trace Formatting Routines .............................................................. 241
Table 9-1. Network Configuration Parameters ............................................. 256
Table 9-2. Keywords .................................................................................... 257
Table 10-1. driver_ops_t Data Structure ......................................................... 262
Table 10-2. Capabilities Understood by IP ..................................................... 263
Table 13-1. IOCTL Usage by LVM Summary ................................................ 362
Table 13-2. IOCTL Usage by Tape Driver ....................................................... 363
Tables

Table 13-3. SCSI Services ................................................................. 377
Table 16-1. DLKM Topical Sections .................................................. 422
Table 16-2. Important Terms and Concepts ........................................ 423
Table 16-3. Kernel Configuration Tool Set ......................................... 431
Table 16-4. Dynamically-Configured Loadable Module Procedures .......... 435
Table 16-5. Statically-Configured Loadable Modules Procedures ............ 436
Table 17-1. Summary of LP64 and ILP32 Data Models. ......................... 472
Table 19-1. System Commands ........................................................ 518
Table 19-2. Posix.2 Commands for All Control Scripts ......................... 519
Table 19-3. Additional Preserved Commands. .................................... 519
Figures

Figure 1-1. Same Driver Can Operate in Many Configurations ........................................... 28
Figure 1-2. WSIO CDIO as a Buffer Zone ................................................................. 29
Figure 2-1. Noncoherent System ................................................................. 50
Figure 2-2. Physical and I/O Virtual Addressing ...................................................... 53
Figure 3-1. Synchronization Areas ................................................................. 73
Figure 3-2. Synchronization Mechanism ............................................................... 74
Figure 6-1. Three Layered HP-UX Interface to the PCI Bus ........................................ 162
Figure 6-2. Steps to Develop a Networking Driver ................................................... 165
Figure 6-3. Networking Driver Control Block and Structures ...................................... 168
Figure 6-4. STREAMS DLPI Network Driver Sequence ............................................. 185
Figure 6-5. Control Flowchart for Outbound Path ...................................................... 200
Figure 6-6. Control Flowchart for Inbound Path ....................................................... 202
Figure 6-7. Master Agent/Subagents Relationship .................................................... 207
Figure 7-1. HP LANSCAN Command ................................................................. 213
Figure 7-2. HP and IHV LANADMIN Commands .................................................... 214
Figure 7-3. HP and IHV LINKLOOP Commands .................................................... 222
Figure 8-1. Network Tracing, Logging Elements and Data Flow I ............................... 227
Figure 9-1. SAM and Network Driver Interface ....................................................... 254
Figure 10-1. Fastpath ................................................................. 264
Figure 10-2. Fragments ................................................................. 268
Figure 10-3. Packet Flow Diagram ................................................................. 270
Figure 11-1. Mass Storage Stack ................................................................. 280
Figure 12-1. SCSI Data Structures ................................................................. 317
Figure 12-2. I/O Flow Between Various Queues ...................................................... 319
Figure 12-3. Driver Data Structure Inter-Relationship ............................................. 336
Figure 13-1. Call Graph of SCSI Routines and Services ......................................... 360
Figure 15-1. OLA/R State Diagram ................................................................. 414
Figure 16-1. DLKM Procedural Flowchart ............................................................. 434
Figure 18-1. LBI Interrupt Migration ................................................................. 483
Figure 18-2. TBI Interrupt Migration ................................................................. 486
Figure 19-1. SD Structure ................................................................. 494
Figure 19-2. SD Package ................................................................. 497
About This Document

This manual describes how to write I/O driver routines for the WSIO interface on HP 9000 workstations (Series 700 and others) and servers (Series 800 and others). Except where noted, this manual is applicable to all workstations and servers using Release 11i v1. The manual is organized as follows:

**Chapter 1, “Overview of the Driver Environment.”** — The I/O subsystem’s structure and how drivers fit into this environment.

**Chapter 2, “HP-UX I/O Subsystem Features.”** — Features of the I/O subsystem, such as types of drivers, memory mapping, flow of I/O requests, data structures, and interrupt handling.

**Chapter 3, “Multiprocessing.”** — Covers the kernel services that handle synchronization used by drivers on multiprocessor systems.

**Chapter 4, “Writing a Driver.”** — A step-by-step strategy for writing drivers. It includes descriptions of routines used by device drivers, interface drivers and combined drivers.

**Chapter 5, “Installing Your Driver.”** — Installing the driver in the kernel and configuring it to communicate with the hardware.

**Chapter 6, “Creating Networking Device Drivers.”** — Designing and writing networking device drivers.

**Chapter 7, “LAN Commands.”** — Provides the user with the ability to scan and administer LAN interfaces on an HP-UX system.

**Chapter 8, “Tracing and Logging in LAN Drivers.”** — Aids in troubleshooting network problems.

**Chapter 9, “SAM Support for LAN Drivers.”** — HP-UX system administration tool, that provides both GUI and TUI based interface to configure system resources.

**Chapter 10, “CKO and Transport IOCTLs.”** — Explains the interaction between the HP-UX transport layers and the DLS providers to create a data transfer mechanism between the layers in a networking stack.


**Chapter 12, “Writing a SCSI Interface Driver.”** — Provides information on designing and developing SCSI transport drivers, also known as a Host Bus Adapter (HBA).

**Chapter 13, “Writing SCSI Device Drivers.”** — How to write SCSI bus device driver routines.

**Chapter 14, “Writing PCI Device Drivers.”** — How to write PCI bus driver routines.

**Chapter 15, “On-Line Addition / Replacement.”** — Driver requirements when adding or removing a PCI card with power on.

**Chapter 16, “Writing a DLKM Driver.”** — Adding a kernel module to a running UNIX system without rebooting the system or rebuilding the kernel.

**Chapter 17, “How to Make Pre 11.0 Drivers 64-Bit Safe.”** — How to modify a Release 10.20 32-bit driver to run in a Release 11.0 32-bit or 64-bit environment.

**Chapter 18, “Interrupt Migration.”** — How to use this mechanism for managing interrupt assignments.

**Chapter 19, “Creating a Software Depot.”** — Describes the Software Depot (SD) creation techniques.
Intended Audience

This document is intended for system administrators or developers responsible for porting or writing drivers. Developers are expected to have:

- Experience writing programs in the C language.
- Working knowledge of the basic concepts of writing a driver.
- An understanding of the functionality of the hardware for which the driver is being written.
- Read the HP-UX System Administration Tasks manual and performed system administration.
- Working knowledge of the virtual memory, I/O, and file system areas in the HP-UX and/or UNIX operating systems.

This document is not a tutorial.

Using this Manual

Reading this manual provides information on the tasks that need to be performed to write a new driver and port an existing driver. Various steps will differ depending on the task being performed.

NOTE

This book contains many examples of C programs to help design device drivers. Because of page width restrictions, some long lines of code exceed the space available and break in unintended places. Please treat these “broken” lines as one line. We recommend that you use the sample files included with this manual when possible, rather than retyping the examples.
Typographical Conventions

This document uses the following conventions.

audit (5) An HP-UX manpage. In this example, audit is the name and 5 is the section in the HP-UX Reference. On the web and on the Instant Information CD, it may be a hot link to the manpage itself. From the HP-UX command line, you can enter “man audit” or “man 5 audit” to view the manpage. See man (1).

Book Title The title of a book. On the web and on the Instant Information CD, it may be a hot link to the book itself.

KeyCap The name of a keyboard key. Note that Return and Enter both refer to the same key.

Emphasis Text that is emphasized.

Bold Text that is strongly emphasized.

Bold The defined use of an important word or phrase.

ComputerOut Text displayed by the computer.

UserInput Commands and other text that you type.

Command A command name or qualified command phrase.

Variable The name of a variable that you may replace in a command or function or information in a display that represents several possible values.

[ ] The contents are optional in formats and command descriptions. If the contents are a list separated by |, you must choose one of the items.

{ } The contents are required in formats and command descriptions. If the contents are a list separated by |, you must choose one of the items.

... The preceding element may be repeated an arbitrary number of times.

| Separates items in a list of choices.

HP Encourages Comments

HP encourages your comments concerning this document. We are truly committed to providing documentation that meets your needs.

Please send comments to: netinfo_feedback@cup.hp.com

Please include document title, manufacturing part number, and any comment, error found, or suggestion for improvement you have concerning this document. Also, please include what we did right so we can incorporate it into other documents.

Email & Internet Resources

Interface program and developer resource materials are available at the following locations:

- Hardware Provider Program at http://www.hp.com/dspp/hphp
- Interface Program E-mail at interface@fc.hp.com
- Developer Resource at http://www.hp.com/dspp

Support/Compatibility Disclaimers

Since drivers function at the level of the kernel, Hewlett-Packard Company (HP) reminds you of the following:
Adding your own driver to HP-UX requires relinking the driver into HP-UX. With each new release you should plan on recompiling your driver in order to reinstall it into the new HP-UX kernel. Many header files do not change. However, drivers typically use some header files that could change across releases (i.e., you can have some system dependencies).

HP provides support services for HP products, including HP-UX. Products, including drivers, from non-HP parties receive no support, other than the support of those parts of a driver that rely on the documented behavior of supported HP products.

Should difficulties arise during the development and test phases of writing a driver, HP may provide assistance in isolating problems to determine if:

- HP hardware is not at fault; and
- HP software (firmware) is not at fault by removing user-written kernel drivers.

When HP hardware, software, and firmware are not at fault, you should seek help from the third party from whom you obtained software or hardware.
Reference Documentation

Hewlett-Packard Company

- Dealer Configuration File Creation Guide, HP Part No. D2230-90001
- HP-UX Managing Systems and Workgroups, HP Part No. B2355-90664
- HP-UX Reference, HP Part No. B2355-90052
- HP-UX System Administration Tasks, HP Part No. B2355-90079
- HP C Programmer’s Guide, HP Part No. 92434-90002
- Configuring HP-UX for Peripherals, HP Part No. B2355-90053
- Installing and Updating HP-UX, HP Part No. B2355-90078
- PA-RISC 1.1 Architecture and Instruction Set Reference Manual, HP Part No. 09740-90039
- PA-RISC Procedure Calling Conventions Reference Manual, HP Part No. 09740-90015
- Managing HP-UX Software with SD-UX, HP Part No. B2355-90044

Other References

- PCI Local Bus Specification, Revision 2.1, PCI Special Interest Group
- PCI System Design Guide, Revision 1.0, PCI Special Interest Group
- Data Link Provider Interface Specifications, Unix International
1 Overview of the Driver Environment

This book is intended for individuals writing interface and device drivers for HP-UX Workstations and servers.

This chapter describes how the HP-UX I/O subsystem is structured, and how the driver fits into this environment.
How the I/O Subsystem is Structured

The I/O subsystem provides a uniform interface for user processes when reading information from and writing information to devices, providing the System Administrator with information about the system’s I/O hardware and software. It also provides an environment within which drivers can control I/O devices. This environment provides drivers with tools used by developers to do their work and contexts in which driver code can run.

Much of this environment is provided by kernel code; but configuring the system (associating drivers with devices) and setting the values of system parameters (tuning the system) also play their parts.

The HP-UX I/O system has two sections: a **General I/O System** (GIO), and one or more **Context-Dependent I/O** (CDIO) modules.

The GIO, which is always present, provides all the functionality that is global to the I/O system, and provides services the CDIOs can use.

CDIOs contain all bus-specific and device-specific functionality. A system administrator configures CDIOs into a system only as necessary.

**General I/O System (GIO)**

The GIO manages the system’s I/O resources and data structures, drives the system-configuration process, providing an interface to the system-administration utilities. GIO functionality includes:

- Management of data structures used for I/O configuration.
  - Data structures that can be manipulated by system-administration utilities or that are global to the system must be maintained by the GIO. These include:
    - I/O tree node
    - Block and character switch tables
    - **Kernel Device Table** (KDT)
- The algorithms driving system configuration.
  - System configuration is driven by the GIO, although all interaction with interface cards and devices is handled by CDIOs.
- The system administration interface
  - System administration utilities must see a consistent view of the system that are independent of individual drivers' views.
Context-Dependent I/O Modules (CDIOs)

CDIOs contain functionality specific to a particular bus or driver environment (the context is defined by the bus or driver environment). Individual, independent CDIOs provide a way to hide any driver-environment specifics from other environments. This, in turn, allows individual component CDIOs to be configured in or out of a system as needed.

There are two categories of CDIOs:

1. Bus-nexus CDIOs (CDIOs that communicate directly with a bus), which provide bus-dependent services to other CDIOs. They may have bus-nexus drivers to control bus adapters or bus converters. A kernel can contain the following bus-nexus CDIOs:
   - CORE CDIO (Core Context Dependent I/O Module)
   - PA CDIO (Precision Architecture Context Dependent I/O Module)
   - EISA CDIO — optional (EISA Context Dependent I/O Module)
   - PCI CDIO — optional (PCI Context Dependent I/O Module)

2. Driver Environment CDIOs, which provide drivers with a defined environment. Drivers within a CDIO’s environment share a common set of services and entry points. A kernel can contain the Workstation I/O (WSIO) Context-Dependent I/O module CDIO.

Basic Components of a CDIO

- **GIO Interface**
  The GIO interface contains entry points invoked by the GIO to access the CDIOs. Generic configuration requests are converted by the CDIO into the appropriate context-dependent functions.

- **Inter-CDIO Communication Interface**
  Inter-CDIO communication is provided by services that allow one CDIO to claim hardware modules found by another CDIO, or to gain access to hardware resources maintained by another CDIO.

- **Driver Services**
  Driver services that define a driver environment are part of the CDIO. There may be cases where a service in one CDIO is called by a driver in another CDIO. This happens, for instance, with some EISA card drivers that are part of the WSIO CDIO, but that call bus-dependent functions from the EISA CDIO. However, bus-dependent functions are mostly hidden by services in the WSIO CDIO driver environment to reduce dependencies. Therefore, drivers that control CORE functions and EISA cards do not always require the EISA CDIO (if there is no EISA hardware in the system).

  For example, some drivers can control cards in either a CORE CDIO or EISA CDIO environment. The WSIO CDIO driver environment helps to hide the bus-specific services, so that both EISA and CORE do not need to be pulled in every time the driver is configured.

- **Drivers**
  In most cases, a CDIO contains drivers. In a bus-nexus CDIO like EISA, the driver is the EISA bus-nexus manager that configures the EISA adaptor and that provides services specific to EISA. A driver-environment CDIO like WSIO can support many drivers. In most cases, the drivers you write will interact with the WSIO CDIO.

- **Management of I/O resources**
  A CDIO controls resources specific to a device. For example, the PA CDIO controls resources specific to Precision Architecture, such as interrupt bits. The WSIO CDIO manages common structures like the Interface Select Code (ISC) table.
How the Driver Environment Works

The WSIO CDIO was originally designed for the workstation single-processor environment. With HP-UX Release 10.20, its functionality was expanded to encompass the server multiprocessor environment. Because the WSIO CDIO is a driver-environment CDIO, it provides a consistent environment regardless of the configuration with bus-nexus CDIOs. Drivers residing within the WSIO CDIO continue to operate smoothly without knowing the underlying configuration.

For example, a driver in the WSIO CDIO (such as a SCSI disk driver) can make the same service calls whether it is configured to work with a CORE CDIO, an EISA CDIO, or with the system’s I/O bus (see Figure 1-1, “Same Driver Can Operate in Many Configurations.” To the driver, all configurations look the same. It is the task of the WSIO CDIO to interpret the service calls and to take the appropriate actions for the given configuration.

Figure 1-1  Same Driver Can Operate in Many Configurations

Another way to view the role of the WSIO CDIO is to see it as a buffer zone that protects its drivers from the peculiarities of the bus-nexus CDIO that it is configured with, see Figure 1-2, “WSIO CDIO as a Buffer Zone.” The WSIO CDIO masks and hides all interface differences from the driver, handles configuration issues, and monitors resources.
Figure 1-2  WSIO CDIO as a Buffer Zone

Refer to the *HP-UX 11i v1 Driver Development Reference Guide* for the reference pages for WSIO CDIO routines, services, and data structures.
This chapter explains software concepts associated with the HP-UX I/O subsystem. It also provides information and models to help when writing a driver.
General Driver Topics

The topics discussed in this chapter form the foundation of most kernel drivers.

- The “Overview of Driver Types” section discusses pseudo, interface, and monolithic drivers and block and character devices.
- The “Major and Minor Numbers” section describes how these numbers are assigned and how drivers use them to communicate with various aspects of software and hardware.
- The “System Calls” section discusses the stages of I/O, from a user process issuing a system call requesting I/O, to the device making the data transfer.
- The “Kernel Data Structures Used for I/O” section describes most of the kernel data structures used by the I/O subsystem.
- The “Timeout Mechanisms” section shows how a driver controls the wait for an event.
- The “Interrupt Handling” section discusses how to handle interrupts and software triggers.
- The “Memory Allocation and Mapping I/O Space” section discusses general kernel memory allocation and gives pointers to accessing card registers.
- The “Cache Coherence” section discusses maintenance of cache coherence.
- The “DMA Mapping” section describes I/O virtual address mapping and WSIO mapping services.
Overview of Driver Types

WSIO supports several types of drivers:

- Device
- Pseudo
- Interface
- Monolithic

It also supports device access types:

- Block
- Character

Device, Pseudo, and Interface Drivers

- **Device Drivers** — Manage peripheral devices such as disk drives, tape drives, terminals, and printers. Device drivers are accessed from user and system programs and pseudo drivers. A device driver is needed for each type of device (SCSI disk drives, for example).

- **Pseudo Drivers** — Are drivers that are not associated with a particular device. In many cases, they are preprocessors for a device driver. Examples include the Logical Volume Manager (LVM), and the `pty` driver for logical terminals, or certain kinds of pipes. Pseudo drivers are accessed from user and system programs.

- **Interface Drivers** — Manage interface cards (and built-in interface devices) that attach to a bus such as PCI. These include SCSI, RS-232-C, and MUX cards (several RS-232-C ports on one card). Device drivers may be layered on top of interface drivers. An interface driver is needed for each type of interface card.

- **Monolithic Drivers** — Are combined device and interface drivers. And can have a monolithic driver:
  - When a particular type of device is always connected to a particular type of interface card. For example, there is only one interface card and one device on the card.
  - If the card itself acts like a device and is directly addressable (for example, LAN and audio drivers).

Block and Character Devices

HP-UX supports two types of access to devices; block and character.

1. The driver of a **block device** accesses a structured device, such as a disk or magnetic tape that supports a file system. These drivers generally allow buffered, random access. Block devices are accessed through the file system buffer cache.

2. The driver of a **character device** accesses a device as a character device (everything not accessed in blocks). They are also called raw devices.
Features of Block Devices

Block devices have file system support and are random access devices.

File System Support

HP-UX transfers data between a user process and a block device in blocks of size BLKDEV_IOSIZE (currently 2048 bytes). Use block device drivers for disks and any other devices on which a file system can reside.

Data Caching

Disk drives should use data caching because the system benefits from caching file-system data. The read() and write() system calls provide more information about this:

read()  
In data caching, the kernel first checks the file system’s buffer cache for the requested data. If the data is in the buffer cache, the kernel can return the data to the calling process without invoking the block-device driver.

write()  
In data caching, the kernel copies the data to the file system’s buffer cache, and then returns control to the user process. The kernel uses buffer-cache algorithms to determine when to copy data from buffers in the cache to the device. On determining that the data in a buffer should be copied, the kernel calls the appropriate block-device driver, allowing the driver to perform asynchronous, buffered I/O.

Features of Character Device Drivers

Character device drivers have no file system support, and are typically sequential-access devices which do raw I/O.

No File System Support

The kernel does not cache data for character devices. On accessing a device as a character device, data is transmitted in units of one or more bytes.

Raw I/O: Buffered or Unbuffered

There are two cases for raw I/O with character devices. First, use physio() for direct (unbuffered) data transfer by the character-device driver. Second, have a character device driver set up its own buffer using copyin() or uiomove(). This method is useful for a small amount of data or if you need to control the data rate.
The kernel recognizes device drivers by major and minor numbers encoded in the device-special files. Drivers that support both block and character I/O (such as a SCSI disk driver and an optical autochanger) have both a block-major number and a character-major number. Devices that support only character-mode access have only a character-major number.

The kernel uses the major number of a device-special file to index into a device switch table. A device switch table contains an entry for each driver in the system. Each entry contains addresses that map I/O system calls to the device’s entry points. The kernel calls a device driver by making an indirect call through an entry point in a device switch table.

The kernel maintains two device switch tables, one for block devices (bdevsw), which uses the file system buffer cache, and one for character devices (cdevsw), which uses character queues or no buffering scheme. The kernel constructs these device switch tables during system configuration.

The driver uses minor numbers to locate the device and for driver-specific information.

Assigning Major Numbers

The kernel can assign major numbers dynamically. To have the system dynamically assign a major number to the driver:

- Specify -1 in both the b_major and c_major fields of the drv_info_t structure in your driver's header. See “Step 3: Defining Installation Structures” in Chapter 4, “Writing a Driver,” for more information about the drv_info_t structure.

- Also in the drv_info_t structure, set the following bit values in the flags field. If a block driver, set the DRV_BLOCK value. If a character driver, set the DRV_CHAR value. If driver is both a block and a character driver, set both values.

After having built and booted a kernel containing the driver, find out what major number has been dynamically assigned by using the lsdev command (see lsdev (1M). The lsdev reads the information provided by the driver header and retrieves the major number. Major numbers are displayed in decimal form. lsdev (1M) has an example of a dynamic way of extracting the major number from a standard HP-UX driver.

Using Minor Numbers

Minor numbers contain two kinds of information: the location of the interface to which a device is attached, and driver-dependent characteristics. This information is organized by specific bit assignments.

The minor number information is encoded in the device-special file.

For more information, consult the device driver manpages in the HP-UX 11i vi Driver Development Reference Guide.

Device-Special Files

To create a device-special file for the driver, see mknod (1M). A long listing (ls -l) of a typical device-special file might look like this:

crw------- 2 bin 193 0x00080 Jul 12 02:19 mux0

The two fields of importance are the major number (193) and the minor number (0x00080). The major and minor numbers are combined to form a numerical designation for the device driver, in what is called the dev_t format. This format consists of:
Bits 0–7  The major number, which can range from 0 to 255. Character and block major numbers are separate ranges.

Bits 8–31  The minor number. The conventional notation for the minor number follows the format 0xhhhhhh, where h is a four-bit hexadecimal digit. As a general guideline, bits 8–15 encode the instance number of the interface card. The number represents the order in which HP-UX encounters the interface card within a class when binding it into the system. This number is displayed (in decimal notation) in the I column of ioscan -i output. Bits 16–31 encode device and driver dependent characteristics. These can include special rules, such as, for tapes; rewind-on-close, density, for printers; all caps, for disks; section number, unit number.

Device Switch Tables

The device switch tables contain entry points for all device drivers in the system. When a process makes a system call for I/O, the kernel uses the device’s major number as an index into a switch table.

The kernel maintains two switch tables for devices:

1. bdevsw for block devices.
2. cdevsw for character devices.

Each driver of a block device has an entry in the bdevsw table. Each driver of a character device has an entry in the cdevsw table.

The kernel automatically uses the driver’s installation routine, entries in the driver’s dev_ops_t and dev_info_t structures, and information from the driver’s master file in /usr/conf/master.d, to construct the switch tables. See “Step 3: Defining Installation Structures” and “Step 5: Writing Configuration Routines” in Chapter 4, “Writing a Driver,” and “Step 2: Create a Master File” in Chapter 5, “Installing Your Driver.”
System Calls

A user process performs I/O by making system calls. The kernel executes the system calls on behalf of the user process. The processing performed by the kernel depends on:

- Which system call is executed.
- The type of the file specified in the I/O request.

The system calls that perform I/O include `open()`, `close()`, `read()`, `write()`, `ioctl()`, and `select()`, (described in `close(2)`, `ioctl(2)`, `open(2)`, `read(2)`, `select(2)`, and `write(2)`).

A user process performs I/O by making system calls on device special files. When a process issues an I/O system call on a device special file, the corresponding driver routine is called. Before the appropriate driver routine is called, the kernel takes actions that include, for example:

- Checking that the user has permission to access the device.
- Obtaining system buffers to use, if necessary.
- Using the major number to index into a device switch table.
- Calling the driver associated with the device file with the appropriate parameters.

Two Levels of a Driver

The device driver's entry-point routines constitute the upper half of the driver, the user context. A system call from a user program activates the upper half of the driver. The lower half of the driver, the interrupt context, processes interrupts from the device. The halves work as follows:

1. The upper half initiates activity on the device, then waits.
2. The device completes the activity and interrupts, causing the lower half of the driver to tell the upper half that it can continue.

Interrupts are handled by an Interrupt Service Routine (ISR) and supporting routines in the interface driver. This chapter describes interrupts in “Interrupt Handling”, Chapter 4, “Writing a Driver,” describes the interface driver routines.

Flow of an I/O Request

To write a driver, there is a need to understand the interactions that take place between the kernel and the driver as they process an I/O request. This section describes how I/O is accomplished, starting from the user process that issues a system call requesting I/O, continuing with the device transferring the data, and ending with the driver completing the request.

The following items describe how a user request results in driver execution:

1. In the user process:
   - Invoke the kernel.
   - A user process makes an I/O system call and invokes the kernel through the system-call interface. The kernel does some processing related to the management of the process and of resources for the request.
2. In the kernel:
   - Get major and minor numbers.
If the file specified in the I/O system call is a device-special file, the kernel gets the major and minor numbers from the file's inode. The kernel uses the major number to index into a device switch table and sets up parameters, if any, to be passed to the driver.

Call the device driver.

The I/O system call obtains the device driver's entry point from the device switch table's entry, and passes control to the driver, passing parameters to the device driver that provide the driver with information about the request.

3. In the device driver:
   Initiate request and coordinate tasks.

   The device driver's routine does the necessary setup and begins processing of the I/O request. This involves initializing data structures and setting up a request to process the I/O.

   After the driver routine sets up a request for I/O, the driver either waits for the I/O to be completed, or immediately returns control to the routine that invoked it. Whether the driver returns or waits for the I/O to be completed depends on the characteristics of the device and the needs of the driver.

   A device driver routine waits by calling `sleep()`, in which case the user process is put to sleep until a corresponding call to `wakeup()` is issued by another routine.

4. In the interface driver:
   Process request.

   The request is processed by the interface driver when the device is available. After the hardware completes the I/O request, an interrupt is sent back to the device driver, signaling I/O completion.

5. In the device driver:
   Complete request.

   If the device driver has called `sleep()` and is waiting for the device to complete the transfer, the interrupt routine calls `wakeup()` to awaken the sleeping process. When the process awakens, it continues to execute from where it put itself to sleep, doing processing appropriate to completing the system call. Then it returns an integer value (indicating the success or failure of the request) to the kernel routine that invoked it, completing the original request.

6. In the kernel:
   Return control to user process.

   The kernel interprets the return value from the device driver, and sets the return value of the system call accordingly. It then returns control to the user process.
Sample I/O Read Request

The following steps are the sequence of actions taken to process an I/O request on a character device file. In porting or writing drivers, use similar steps, according to the needs of the device:

1. A user runs a program that executes a `read()` system call on a character device file.
2. The `read()` system call:
   a. Performs necessary preprocessing of the request, such as verifying that the file is a character device, checking permissions.
   b. Extracts the major and minor numbers from the inode for the special file.
   c. Uses the major number to index into the `cdevsw` table to obtain the name of the device driver's `driver_read()` routine.
   d. Invokes the `driver_read()` routine.
3. The `driver_read()` routine calls `physio()` to invoke the `driver_strategy()` routine.
4. The `driver_strategy()` routine gets the minor number to access the correct device and decodes the device options.
5. The `driver_strategy()` routine queues a request on the interface driver for a read from the device.
6. The interface driver is currently processing another request, so the `driver_strategy()` routine returns and `physio()` sleeps while waiting for the I/O request to complete.
7. The device interrupts when it completes the previous I/O request.
8. The interface driver processes the interrupt and starts the next request waiting on the queue (our request).
9. The interface driver tells the hardware to perform our read request.
10. The hardware completes our read request and interrupts the system.
11. The interface driver processes the interrupt, waking up the device driver's upper half.
12. Control returns to the `driver_read()` routine, which completes any final device-specific processing of the request.
13. Control returns to the `read()` system call, which completes the request and returns control to the user process.
Kernel Data Structures Used for I/O

This section describes the kernel data structures commonly used by the I/O subsystem. These data structures are:

- The buf structure.
- The interface select code table (isc_table_type structure).
- The I/O switch tables (ifsw and gfsw structures).
- The uio structure.
- The iovec structure.

buf Structure

The buf structure is a buffer header for block I/O for a file system buffer. It is the central data structure used by the File System to buffer data that is passed back and forth between the File System and the I/O system. The information in this structure specifies the buffer and the operations that can be performed on it.

When the kernel invokes the driver_strategy routine, it passes a pointer to a buf structure as a parameter. The driver_strategy routine should schedule the transfer of data into or out of the buffer allocated to the buf structure.

See buf (KER4) for details about fields in the buf structure.

isc_table_type Structure

Each instance of an interface card has an Interface Select Code (ISC) entry that the system maintains in an internal table. Each ISC entry, defined as an isc_table_type structure, is used by WSIO to maintain interface driver information.

An interface driver can get information specific to each instance of its cards by referencing the appropriate ISC entry.

A device driver can call the wsio_get_isc() service to obtain a pointer to the ISC entry for its corresponding interface driver by giving its dev_t number. It can pass that isc pointer to the interface driver in a ifsw function call.

See isc_table_type (KER4) for details about fields in the isc_table_type structure.

I/O Switch Tables

The I/O system supports two I/O interface switch tables through fields in the ISC structure. isc->gfsw is intended for use by the system. isc->ifsw is available for communication between one or more device drivers and one or more interface drivers. The fields in both are filled in by interface drivers, usually during the boot process. The field values default to NULL.

Generic Function Switch

The generic function switch, isc->gfsw, defined in <sys/io.h>, is intended for system-to-interface driver communication, not device driver-to-interface driver communication. It consists of pointers to two routines:

- isc->gfsw.init points to a driver-defined interface initialization routine that is run by the system during the boot process. See Chapter 4, “Writing a Driver.”
isc->gfsw.diag points to a driver-defined interface diagnostic routine whose usage is currently undefined.

**Interface Function Switch**

The interface function switch, isc->ifsw, is intended for device driver-to-interface driver communication. This is where a device driver can call the functions of an interface driver. It consists of an address pointer that is set, presumably to a structure that is defined as interface functions and other relevant flags and data that make up the interface driver.

A principal use of the interface function switch is where there is one or more device drivers working with two or more interface drivers. For example, a device driver working with two interface drivers that support the same disk protocol.

The interface drivers use identical structures to specify their operations and put the addresses of the structures in their respective isc->ifsw fields. The device driver obtains the ISC structure for the appropriate interface driver see `wsio_get_isc` (WSIO3) and then calls the interface driver through the switch table to perform the operation.

This allows the device driver to trigger interface-specific routines without actually having to know which interface driver is configured with it. Multiple interface drivers could be configured interchangeably with this device driver, as long as they share the same ifsw type definition.

In a simple one device driver, one interface driver setup, the device driver can call the interface routines directly, ignoring the isc->ifsw switch table.

**uio Structure**

For each read() and write() system call for a character device, the kernel allocates and fills out a uio structure. Then it passes this structure to the driver's driver_read or driver_write routine.

The uio structure contains a pointer to the user's data area. The driver_read and driver_write routines can either transfer the data between the user's buffer and the driver by calling physio() for a block transfer, or by calling uiomove() to move the data byte by byte.

Drivers of character devices seldom access individual fields in the uio structure. The uiomove() and physio() routines take care of many details for you.

See uio (KER4) for details about fields in the uio structure.

**iovec Structure**

The data buffer descriptor is passed in as part of the uio structure for character I/O and also used by the CDIO mapping services. See iovec (KER4) for details about fields in the iovec structure.
Timeout Mechanisms

A driver uses timeout mechanisms when it needs to wait for a response from a device. Timeouts ensure the return of control to the driver if the device fails to respond within an allotted time. They can also be used by the driver to poll the status of device registers at regular intervals. This section describes the following HP-UX timeout routines:

- `timeout()`
- `Ktimeout()`
- `untimeout()`

How Timeouts Work

The `timeout()` routine causes a timeout to occur a specified number of clock ticks later. Execution occurs in the interrupt context of the current processor at priority level 2. The `Ktimeout()` routine differs by causing a timeout to occur at priority level 5.

The recommended timeout routine is `timeout`. At priority level 2, external interrupts are still enabled; whereas, at priority level 5, external interrupts are disabled. When processing timeouts at priority level 5, as is done via `Ktimeout()`, the driver may unnecessarily cause interrupt servicing to be delayed. `Ktimeout()` is provided for legacy uniprocessor drivers that must synchronize execution with their interrupt service routines.

The `timeout()` routine works as follows:

1. The driver calls `timeout()` to set a timeout.
2. The driver continues processing device I/O requests.
3. If the driver does not call `untimeout()` before the time specified in the `timeout()` call elapses, the timeout occurs and the kernel executes the routine the driver specified in the call to `timeout()`.

The `timeout()` routine is typically used where a device might hang while processing a request. By setting a timeout, the driver has an opportunity to recover if the device hangs.

timeout() Routine

```c
struct callout_t * timeout(int(*func)(),
    caddr_t arg, int t);
```

- `func` Address of the function to be called when the timeout occurs.
- `arg` Argument passed to `func` when it is called.
- `t` Number of clock ticks to wait before the timeout occurs. Express `t` in terms of multiples of the system variable `Hz`, which is defined as the number of ticks in a second. For example, `Hz/2` is half a second.

See `timeout` (KER2) for details.
untimout() Routine

```c
int untimeout (int (* func)(),caddr_t arg);
```

`func` Address of the function passed to `timeout()`.

`arg` Argument passed to the `timeout()`.

See `untimout (KER2)` for details.
Interrupt Handling

In 11i there is a set of new interrupt services that allow drivers to allocate multiple interrupt resources. Drivers can specify whether an interrupt is to be line based or transaction based. The new WSIO interrupt services are shown in Table 2-1, “New WSIO Interrupt Services.”

### Table 2-1 New WSIO Interrupt Services

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wsio_intr_alloc()</td>
<td>Allocates an interrupt object.</td>
</tr>
<tr>
<td>wsio_intr_free()</td>
<td>Frees an interrupt object.</td>
</tr>
<tr>
<td>wsio_intr_activate()</td>
<td>Enables an interrupt object.</td>
</tr>
<tr>
<td>wsio_intr_deactivate()</td>
<td>Disables an interrupt object.</td>
</tr>
<tr>
<td>wsio_intr_set_cpu_spec()</td>
<td>Sets up transaction based interrupts.</td>
</tr>
<tr>
<td>wsio_intr_set_irq_line()</td>
<td>Sets up line based interrupts.</td>
</tr>
<tr>
<td>wsio_intr_get_irq_line()</td>
<td>Gets IRQ line for line based interrupt.</td>
</tr>
<tr>
<td>wsio_intr_get_txn_info()</td>
<td>Get transaction address and data values.</td>
</tr>
</tbody>
</table>

Each interrupt object represents a different interrupt resource. A driver can allocate one or more interrupt objects by calling `wsio_intr_alloc()` to allocate each new one. Each object is then passed as a parameter to the other interrupt services to specify which interrupt resource.

Drivers can configure the type of interrupt to be line or transaction based. To specify a transaction based interrupt they would call `wsio_intr_set_cpu_spec()` passing in the specific interrupt object. For line based interrupts they would call `wsio_intr_set_irq_line()`. Some platforms may not support both types of interrupts due to the underlying hardware so the driver may be restricted to using only one type. The services will return an error condition when a particular type of interrupt is not supported by the underlying hardware.

To get the **Interrupt ReQuest** (IRQ) line for a line based interrupt a driver would call `wsio_intr_set_irq_line()`. To get the interrupt address and vector for transaction based interrupts they would call `wsio_intr_get_txn_info()`. The WSIO services `wsio_intr_activate()` and `wsio_intr_deactivate()` are used to enable and disable interrupts.

There are two new event types that can be passed to a driver's event handler. Drivers that use transaction based interrupts must support them. They are `CPU_ONLINE` and `CPU_OFFLINE`. The first indicates that a new CPU is available and can be used by the driver for transaction based interrupts. The second indicates that a CPU is going away and the driver must migrate its interrupts to another CPU. For more on the driver event handler see the man pages for the `wsio_install_drv_event_handler` and `wsio_reg_drv_capabilities`. Also, see the sections on these services in Chapter 4, “Writing a Driver,” in this manual.
Software Trigger Mechanism

The software trigger mechanism provides software triggering of interrupt service routines. The `sw_trigger()` routine arranges the calling of a routine in interrupt context at a given priority level.

- Use a software trigger when your driver needs to acknowledge a device's interrupt quickly, at a high level, but can do the rest of the interrupt processing less urgently, at a lower level.
- Software triggers provide a way for the top half of a driver to trigger the lower half to perform a specific function.

The kernel uses a linked list of structures to represent software triggers waiting to be serviced. The kernel checks this list each time it finishes servicing an interrupt. Elements of the list are `sw_intloc` structures, defined in `/usr/include/sys/timeout.h`.

Software Trigger Routine

Drivers call the `sw_trigger()` function to request a software trigger.

```c
void sw_trigger(struct sw_intloc *intloc, int (*proc)(),
                caddr_t arg, int level, int sublevel);
```

- `intloc`: A pointer to a `sw_intloc` structure to be added to the queue of software triggers. The driver allocates the structure, zero-filled. The `sw_trigger()` routine initializes its fields.
- `proc`: The address of a routine to be called when the software trigger is executed.
- `arg`: The argument to be passed to `proc`.
- `level`: The priority level of the software trigger.
- `sublevel`: The priority level of the software trigger. Currently, sublevels are not implemented. Drivers should use 0 as the last argument.

When it checks the list, the kernel processes all requests for software triggers whose `level` is greater than the current interrupt level. The kernel processes pending requests in decreasing order of priority.

The `sw_trigger()` routine checks to see if the structure to which `intloc` points is already on the trigger queue. If it is, the kernel throws this request away, thus permitting only one pending request per `sw_intloc` structure. If the driver needs to have more than one software trigger pending, it must use separate `sw_intloc` structures.

The `level` value has the following restrictions:

- The driver cannot set a software trigger higher than the current processor priority level.
- Cannot call `sw_trigger()` with `level` set to 7.
Skeletal Driver Fragment

The following fragment of a skeleton driver acknowledges an interrupt from a card at a high priority, and then uses a software trigger to defer the bulk of the interrupt processing to a lower priority.

```c
#include <sys/types.h>
#include <sys/timeout.h>

static struct sw_intloc mycard_intloc;

static void
mycard_isr(void)
{
    caddr_t reason;
    /* stop card from interrupting */
    mycard->control = .....;

    /* determine reason for interrupt and do any immediate interrupt processing */
    reason = ...; /* values from card regs */

    /* set up swtrigger() request to perform */
    /* remainder of interrupt processing at */
    /* a lower priority level. */
    sw_trigger(&mycard_intloc, mycard_isr2, reason, 3, 0);

    return;
}

static int
mycard_isr2(caddr_t reason)
{
    /* complete secondary interrupt processing */
    switch ((intptr_t)reason) {
        case IOCOMPLETE:
            /* process I/O complete condition */
            case IOERROR:
                /* processing for I/O error */
                .
                .
                .
            return 0;
    }
}
Memory Allocation and Mapping I/O Space

The WSIO CDIO has introduced several new memory allocation services. Drivers should use these services rather than the older kmalloc() and kfree() services. The new services take advantage of new kernel VM features. They are also sensitive to the I/O devices locality on ccNUMA platforms.

Memory Allocation Services

The first two services, wsio_alloc_mem_handle() and wsio_free_mem_handle() are used to create and destroy memory allocation handles. The driver can use these services to create and destroy handles that specify the type of memory it wants to allocate.

The second two services, wsio_alloc_mem() and wsio_free_mem(), are used to allocate and free memory using the memory allocation handle created by the service wsio_alloc_mem_handle(). Typically, drivers would call wsio_alloc_mem_handle() in their init routines to specify the type of memory they want to allocate. The service will return a handle that the driver would then pass into the services wsio_alloc_mem() and wsio_free_mem() when allocating and freeing buffers.

Drivers can allocate multiple memory handles for the different types of memory buffers they use. For example, a driver could specify a memory handle that only allocates physical memory below four Gbytes. It could then specify another handle for allocating memory that is always physically contiguous. The services are listed in Table 2-2, “Memory Allocation Services.”

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wsio_alloc_mem_handle()</td>
<td>This service is used to specify the type of memory a driver will allocate.</td>
</tr>
<tr>
<td></td>
<td>It returns a memory handle.</td>
</tr>
<tr>
<td>wsio_free_mem_handle()</td>
<td>This service frees a memory handle allocated by</td>
</tr>
<tr>
<td></td>
<td>wsio_alloc_mem_handle().</td>
</tr>
<tr>
<td>wsio_alloc_mem()</td>
<td>This service is called to allocate memory.</td>
</tr>
<tr>
<td>wsio_free_mem()</td>
<td>This service is called to free memory.</td>
</tr>
</tbody>
</table>

For more specific information on how to call these services and the parameters passed to them, refer to the manpages in the HP-UX 11i v1 Driver Development Reference Guide.
Memory Mapped Registers Services

The WSIO CDIO has introduced three new sets of services that drivers can use to discover, map, and access a card's memory mapped registers, configuration space, and I/O port space.

Drivers can call the service `wsio_get_all_registers` in their attach routine to get the memory mapped register sets of an interface card. They would then call the service `wsio_map_reg()` repeatedly to map each set in. Finally, the driver can call `wsio_readXX()` to read the register or `wsio_writeXX()` to write to the register. There are separate services for 8, 16, 32 and 64 bit registers (XX == 8, 16, 32, or 64). The new set of services to get, map, and access memory mapped registers are shown in Table 2-3, "Memory Mapped Register Services."

For more specific information on how to call these services and the parameters passed to them, refer to the manpages in the HP-UX 11i v1 Driver Development Reference Guide.

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wsio_get_all_registers()</code></td>
<td>Returns an array of all device registers.</td>
</tr>
<tr>
<td><code>wsio_map_reg()</code></td>
<td>Maps a device register to host memory.</td>
</tr>
<tr>
<td><code>wsio_unmap_reg()</code></td>
<td>Unmaps a device register.</td>
</tr>
<tr>
<td><code>wsio_readXX()</code></td>
<td>Reads from a device register where XX is either 8, 16, 32, 64.</td>
</tr>
<tr>
<td><code>wsio_writeXX()</code></td>
<td>Writes from a device register where XX is either 8, 16, 32 or 64.</td>
</tr>
</tbody>
</table>

Configuration Space Accesses

This set of services is used by device drivers to discover and access configuration space resources. The first service, `wsio_map_cfg_handle()` is used to obtain a configuration space handle. The handle is then passed to the services `wsio_cfg_inXX()` and `wsio_cfg_outXX()`. There are separate versions of these services for 8, 16, 32, and 64 bit accesses (XX == 8, 16, 32, 64). The new set of services for configuration space accesses are shown in Table 2-4, "Configuration Space Accesses."

For more specific information on how to call these services and the parameters passed to them, refer to the manpages in the HP-UX 11i v1 Driver Development Reference Guide.

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wsio_map_cfg_handle()</code></td>
<td>Obtains a configuration space handle.</td>
</tr>
<tr>
<td><code>wsio_unmap_cfg_handle()</code></td>
<td>Release a configuration space handle.</td>
</tr>
<tr>
<td><code>wsio_cfg_inXX()</code></td>
<td>Reads from configuration space.</td>
</tr>
<tr>
<td><code>wsio_cfg_outXX()</code></td>
<td>Writes to configuration space.</td>
</tr>
</tbody>
</table>
I/O Port Access Services

Drivers that control devices that have I/O ports can use these services to access them. The Drivers would first call `wsio_get_ioports()` to obtain an array of I/O port addresses for an interface card. Typically this is done in the driver attach or init routine when claiming or configuring the card. Next it would call `wsio_map_port()` to map each set of I/O port addresses in. `wsio_map_port()` returns a handle that the driver can then pass to `wsio_port_inXX()` or `wsio_port_outXX()` to write or read the port. There are separate versions of `wsio_port_inXX()` and `wsio_port_outXX()` for reading and writing 8, 16, 31 and 64 bit values.

The I/O port access services are shown in Table 2-5, “I/O Port Access Services.”

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wsio_get_ioports()</code></td>
<td>Obtains the addresses and sizes of I/O ports.</td>
</tr>
<tr>
<td><code>wsio_map_port()</code></td>
<td>Obtains a port handle.</td>
</tr>
<tr>
<td><code>wsio_unmap_port()</code></td>
<td>Releases a port handle.</td>
</tr>
<tr>
<td><code>wsio_port_inXX()</code></td>
<td>Reads from an I/O port.</td>
</tr>
<tr>
<td><code>wsio_port_outXX()</code></td>
<td>Writes to an I/O port.</td>
</tr>
</tbody>
</table>

For more specific information on how to call these services and the parameters passed to them, refer to the manpages in the HP-UX 11i v1 Driver Development Reference Guide.
Cache Coherence

Cache coherence is the consistency of data in host memory as viewed by processor caches and I/O devices. HP-UX supports hardware platforms with processor caches that are either coherent, semicoherent or noncoherent to I/O devices.

Semicoharent platforms implement DMA similar to coherent systems. However, the data read from an I/O device, software must synchronize the data that’s been read into host memory after the DMA transaction completes.

Noncoherent platforms implement DMA in which accesses to data in host memory by I/O devices are not made consistent with processor caches by hardware. Software must explicitly flush the processor caches prior to starting a DMA transaction by an I/O device; and, in the case of data read from an I/O device, purge the processor caches after the DMA transaction completes. Refer to Figure 2-1, “Noncoherent System.”

Figure 2-1 Noncoherent System
**Driver Requirements for Coherency**

Drivers assume that platforms are noncoherent and must explicitly control flushing and synchronization of the processor caches by calling `dma_sync_IO()`. Drivers written for noncoherent platforms will work correctly on coherent and semicoherent platforms.

The `dma_sync_IO()` routine is sensitive to the underlying coherency of the platform. If the platform is coherent, `dma_sync_IO` does nothing, the hardware provides the coherency functionality. If the platform is semicoherent, `dma_sync_IO()` handles the special case where the processor caches must be synchronized with data that have been read into host memory.

PCI buses have special coherency exceptions which are discussed in Chapter 14, “Writing PCI Device Drivers.” They are also discussed in `pci_errata (PCI5)`, `wsio_map_dma_buffer (WSIO3)`, and `wsio_fastmapd_dma_buffer (WSIO3)` in the HP-UX 11i v1 Driver Development Reference Guide.

**Rules For Using dma_sync_IO**

There are three cases to consider when drivers must call `dma_sync_IO()`. These cases are prior to starting a write transaction, prior to starting a read transaction and after completing a read transaction:

1. Prior to starting a write transaction:
   
   For each buffer to be written, the driver must call `dma_sync_IO()` with the `IO_WRITE` hint set. On noncoherent platforms, this will cause the associated processor caches to be flushed. For all but the last buffer, the `IO_NO_SYNC` hint should also be set to reduce the performance penalty of synchronizing the cache flushes on noncoherent platforms.

2. Prior to starting a read transaction:
   
   For each buffer to be read into, the driver must call `dma_sync_IO()` with the `IO_READ_START` hint set. On noncoherent platforms, this will cause the associated processor caches to be purged. For all but the last buffers, the `IO_NO_SYNC` hint should also be set to reduce the performance penalty of synchronizing the cache purges on noncoherent platforms.

3. After completing a read transaction:
   
   For each buffer that’s been read into, the drive must call `dma_sync_IO()` with the `IO_READ` hint set. On noncoherent platforms, this will cause the associated processor caches to be purged of data that may have been prefetched. For all but the last buffer, the `IO_NO_SYNC` hint should also be set to reduce the performance penalty of synchronizing the cache purges on noncoherent platforms. On semicoherent platforms, the processor caches will be made to synchronize with the data read when the `IO_NO_SYNC` hint is not set.
DMA Mapping

There are three address views to host memory; **physical processor**, **virtual** and **I/O virtual**. HP-UX supports platforms that implement all three address views and platforms that implement only the physical and virtual address views:

1. **Physical Address View**
   
   Host memory is accessed through a real address space that is termed the physical address view. HP-UX supports platforms where the physical address width of the processor/memory interconnect is 32 bits wide or wider.

2. **Virtual Address View**
   
   Processors, when executing in virtual mode, access host memory through a virtual address view. Address translation hardware and software convert a virtual address to a physical address before host memory is accessed. Processor caches intercede between the processor and host memory to provide coherent access to host memory. Processor caches access host memory using physical addresses and typically implement their coherency protocol using coherency indices derived from the virtual addresses.

3. **I/O Virtual Address (IOVA) View**
   
   I/O devices access host memory through either a physical address view or an **I/O Virtual Address** (IOVA) view. Platforms where the processor caches are noncoherent with I/O devices and do not have special address translation hardware imbedded in I/O adapters, implement only the physical address view to I/O devices. Platforms that do have address translation adapters for I/O generally implement the IOVA view.

   IOVAs enable devices on a 32-bit wide I/O bus to access host memory physical addresses that are beyond the 32-bit range of the devices. Address translation hardware, embedded in the I/O adapter that connects the I/O bus to the processor/memory interconnect, must be programmed by software to translate I/O bus addresses to host memory physical addresses, refer to Figure 2-2, “Physical and I/O Virtual Addressing.”

   The special address translation hardware often participates in the cache coherence protocols with processors.
I/O Adapters

An I/O adapter provides I/O Virtual Address (IOVA) translation between an I/O bus and the processor/memory interconnect. Devices on the I/O bus issue bus transactions to IOVA memory targets and the I/O adapter translates IOVA memory targets to physical addresses. The I/O adapter also participates in the coherency protocol of the processor caches for platforms that are coherent or semicoherent.

Address translation is assisted by the I/O Page Directory (PDIR) associated with an I/O adapter. The I/O PDIR is analogous to the PDIR used by processors for virtual-to-physical address translations. It is a table maintained by the kernel to provide mappings between IOVAs and physical addresses.

Hardware platforms can be classified as either Coherent or Noncoherent I/O systems. Some hardware platforms supported by HP-UX share the characteristics of both system types. Drivers can be written for these systems by assuming the platform is a Coherent I/O system and using the appropriate WSIO mapping services.

Coherent I/O Systems

Coherent I/O systems are characterized as having I/O adapter hardware with associated I/O PDIR tables. The I/O adapters implement IOVAs and participate in the coherency protocol of the processor caches.

Noncoherent I/O Systems

Noncoherent I/O Systems are characterized as not having I/O adapter hardware and not participating in the coherency protocol of the processor caches.
WSIO Mapping Services

The 11i WSIO provides a new set of DMA mapping services. These services allow drivers to specify multiple DMA objects and configure them for different types of DMA. For example, a driver could specify two DMA objects and configure one for continuous DMA and another for packet DMA. Continuous DMA is where a buffer is allocated and mapped long term for such purposes as control structures, whereas packet DMA implies mapping buffers short term for one DMA and then unmapping them. The new DMA mapping services are shown in Table 2-6, “WSIO DMA Mapping Services.”

Table 2-6  WSIO DMA Mapping Services

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wsio_allocate_dma_handle()</td>
<td>Obtain a DMA handle used to set up DMA.</td>
</tr>
<tr>
<td>wsio_free_dma_handler()</td>
<td>Free a DMA handle.</td>
</tr>
<tr>
<td>wsio_init_map_context()</td>
<td>Initialize a new DMA context.</td>
</tr>
<tr>
<td>wsio_allocate_shared_mem()</td>
<td>Allocate and map a buffer for DMA.</td>
</tr>
<tr>
<td>wsio_free_shared_mem()</td>
<td>Unmap and free a buffer allocated by wsio_allocate_shared_mem().</td>
</tr>
<tr>
<td>wsio_flush_shared_mem()</td>
<td>Flush a buffer allocated by wsio_allocate_shared_mem().</td>
</tr>
<tr>
<td>wsio_map_dma_buffer()</td>
<td>Map an existing memory buffer for DMA.</td>
</tr>
<tr>
<td>wsio_fastmap_dma_buffer()</td>
<td>Map an existing memory buffer for DMA.</td>
</tr>
<tr>
<td>wsio_remap_dma_buffer()</td>
<td>Map an existing memory buffer using a previously allocated range of IOVAs.</td>
</tr>
<tr>
<td>wsio_unmap_dma_buffer()</td>
<td>Unmap a previously mapped memory buffer.</td>
</tr>
<tr>
<td>wsio_iova_to_phys()</td>
<td>Translate an IOVA to the physical address of the buffer.</td>
</tr>
<tr>
<td>wsio_set_dma_callback()</td>
<td>Set a callback function for DMA.</td>
</tr>
<tr>
<td>wsio_dma_pass_thru()</td>
<td>Calls a pass-through function that might not otherwise be directly accessible.</td>
</tr>
<tr>
<td>wsio_dma_set_device_attributes()</td>
<td>Set DMA hints for all DMA objects associated with a device.</td>
</tr>
<tr>
<td>wsio_set_dma_attributes()</td>
<td>Set DMA hints only for the DMA object passed in.</td>
</tr>
</tbody>
</table>
Programming Considerations

The WSIO mapping services are designed to hide the underlying platform hardware from drivers. Drivers that use the new services do not have to know if the platform has an I/O Adapter with an I/O PDIR or whether the platform is coherent or not.

The new 11i WSIO mapping services allow drivers to allocate multiple DMA objects and set attributes in each that favor the different types of DMA the driver/interface does. DMA attributes can take advantage of certain features of the underlying hardware such as prefetch depth or they can specify how a buffer should be mapped in terms of address alignment. Not all attributes are supported by all platforms; an attribute is only a hint by the driver to specify a desired behavior. For a complete list of DMA attributes see the wsio_dma_set_device_attributes() and wsio_set_dma_attributes() manpages in the HP-UX 11i v1 Driver Development Reference Guide.

To use the new WSIO DMA mapping service, a driver must first allocate a DMA object by calling the service wsio_allocate_dma_handle(). It then specifies the desired characteristics for the type of DMA by setting attributes in the object using the service wsio_set_dma_attributes(). Drivers usually do this in their init routines when they claim their interface card.

DMA objects are allocated on a per driver instance. The DMA object is then saved by the driver and used to initiate all DMAs for that driver instance. Drivers can allocate more than one DMA object if they want to specify attributes for different types of DMA. There are no restrictions on the number of DMA objects a driver can create although most drivers will only allocate one or two objects. When a driver has multiple DMA objects for a certain driver instance and wants to set an attribute for all of them it would use the service wsio_dma_set_device_attributes().

To map an existing buffer for packet DMA the driver calls either wsio_map_dma_buffer() or wsio_fastmap_dma_buffer(). The latter is a faster version of the first but has a number of restrictions. When the DMA has completed the driver would unmapping by calling wsio_unmap_dma_buffer(). If wsio_map_dma_buffer() is only able to partially map the buffer, it returns WSIO_MAP_W_PARTIAL instead of WSIO_MAP_OK. Be sure to check for partially mapped buffers when using wsio_map_dma_buffer().

If a driver wants to map a new memory buffer into an existing range of IOVAs it would call the service wsio_remap_dma_buffer(). This service does not guarantee that the new buffer is mapped into exactly the same range of IOVAs as the functionality is dependent upon the underlying platform hardware supporting an IOPDIR.

If a driver wants to allocate and map a buffer for continuous DMA it would call the WSIO service wsio_allocate_shared_mem(). Usually these buffers are allocated in the init routine of the driver when it claims and configures the interface card. These tend to be buffers that are mapped for long term usage.

Any buffers that were allocated by wsio_allocate_shared_mem() must be freed by calling wsio_free_shared_mem(). Since these tend to be long term buffers, this is only done when a driver instance is being removed by a PCI OLD action or the driver is being unloaded via a Dynamically Loadable Kernel Module (DLKM) action.

The service wsio_flush_shared_mem() is used to flush buffers for continuous DMA. Its effect is dependent on the underlying hardware; on some platforms it will have no effect.

The service wsio_iova_to_phys() is used to translate an IOVA to the physical address of the buffer it maps.

One important attribute that can be set is WSIO_DMA_ATTR_INTERLEAVE. Setting this attribute tunes the mapping allocating scheme to favor certain types of DMA. The default value for this attribute is zero (0). This favors drivers that may want to map large buffers for DMA and can stream data to or from many disks concurrently. Setting the attribute to one (1) favors I/O devices that have many DMAs involving small buffers that are processed sequentially. Drivers such as SCSI that control mass storage devices should use the default value. Drivers that control networking devices should set the attribute to one (1). One restriction when setting the attribute to one is the DMA buffers being mapped must reside on the same 4K page.
WSIO Interface Registration Services

The WSIO CDIO provides a number of new driver services to register interface cards with the WSIO and General I/O (GIO) system, assign attributes to them and query hardware paths. An interface can be of type interface or transparent. All interfaces created will have an ISC handle associated with them. The handle can be passed to other WSIO services to reference the interface. Interfaces can be uniquely identified by their hardware paths as no two interfaces can share the same hardware path. There are services to convert an interface to its hardware path and vice versa.

Attributes are properties that are assigned to an interface. An attribute has an ASCII string name to along with a value. The combination of the attribute name and the interface will uniquely identify an attribute. No two attributes assigned to the same interface can have the same name although two attributes assigned to different interfaces can have the same name. When an attribute is created the caller can indicate if it is to be exported to any children underneath.

The list of these services is as follows:

- `wsio_create_interface()`
- `wsio_destroy_interface()`
- `wsio_hwpath_to_isc()`
- `wsio_isc_to_hwpath()`
- `wsio_create_attribute()`
- `wsio_modify_attribute()`
- `wsio_get_attribute()`
- `wsio_sizeof_attribute()`
- `wsio_destroy_attribute()`

Description of Services

The following sections provide a brief description of the services. Refer to the respective manpages in the *HP-UX 11i v1 Driver Development Reference Guide*, for additional details.

**wsio_create_interface Service**

This service is called to register new I/O interfaces. It can be called in a drivers install, probe or scan routine. The types of interfaces that can be created are:

- **WSIO_INTERFACE** — An interface.
- **WSIO_TRANS** — A transparent interface.

The **WSIO_TRANS** is a specialized type of interface. It has no associated hardware and is used to create hardware path elements. Both types of interfaces will have an **isc** handle associated with them. These can then be passed into the other WSIO routines. Both types must be created in the drivers scan or probe routine. The parameters "path" and "parent" are used together to build the hardware path for the new interface. If the parent parameter is not NULL "path" is assumed to be relative to the parent, otherwise it is assumed to be absolute.
The service will first check to see if the interface already exists at the specified hardware path. If it doesn’t it will create it otherwise it will compare the “id”, “name” and “desc” attributes of the existing interface with those passed in as parameters. If they’re different it will update the “id”, “name” and “desc” attributes with the new values and report the difference to the I/O subsystem.

The service returns an “isc” handle for the newly created entry.

**wsio_destroy_interface** Service

This service is called to unregister an interface that was created via a previous call to `wsio_io_create_interface()`. Any resources that the WSIO allocated to the interface will be destroyed.

**wsio_hwpath_to_isc** Service

This service is used to map a hardware path to an I/O interface. If the parameter “ancestor” is not NULL then “path” is assumed to be relative to the hardware path associated with the isc, otherwise “path” is treated as an absolute path. If successful the “isc” handle of the interface is returned. An interface can be of type `WSIO_INTERFACE` or `WSIO_TRANS`.

**wsio_isc_to_hwpath** service

This service returns the hardware path of an interface described by the parameter “isc”. The hardware path is returned in the parameter “path”.

**wsio_create_attribute** Service

The service is called to create a new attribute for an interface. The isc parameter identifies the interface. The second parameter is the name for the new attribute. The parameters “value”, “size” and “flags” identify the initial data for the attribute. The last parameter, “flags” parameter identifies characteristics of the attribute and the data referenced by “value”. This service is safe to call on the **Interrupt Control Stack** (ICS) unless the flag `WSIO_WAIT_OK` is specified in the flags parameters.

The flag `WSIO_ATTR_EXPORT` indicates that this attribute will be visible to any children.

The creating attributes are explained in Table 2-7, “Creating Attributes.”

<table>
<thead>
<tr>
<th>wsio_attrib_flags_t</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSIO_COPYDATA</td>
<td>If set then “value” is assumed to contain an address that references a data buffer and the contents of the buffer is copied, otherwise “value” is assumed to contain the immediate data which is saved.</td>
</tr>
<tr>
<td>WSIO_WAIT_OK</td>
<td>If resources are not available the call will block until they are.</td>
</tr>
<tr>
<td>WSIO_ATTR_EXPORT</td>
<td>The attribute will be exported to any children.</td>
</tr>
</tbody>
</table>

When creating an attribute a reference to a kernel memory data structure can be saved by simply passing in the address and size of the structure as the “value” and “size” parameters. The kernel memory data structure **MUST** then be persistent in memory as long as the attribute exists. If the caller wishes to save a copy of a structure then they **MUST** set the `WSIO_COPYDATA` flag. The service will then copy the contents of the data to an internal buffer.
**wsio_modify_attribute Service**

This service is called to modify the value of an attribute associated with an interface. The \textit{isc} handle of the interface is passed in as the first parameter. The new data for the attribute is defined by the parameters “\textit{value}” and “\textit{size}”. The parameter “\textit{size}” indicates the size of the new data. If size is greater than the original then the service may fail or block if \texttt{WSIO\_WAIT\_OK} is specified in the flags.

The valid attribute flags are explained in Table 2-8, “Modifying Attributes.”

**Table 2-8  Modifying Attributes**

<table>
<thead>
<tr>
<th>\textit{wsio_attrib_flags_t}</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{WSIO_COPYDATA}</td>
<td>If set then “\textit{value}” is assumed to contain an address that references a data buffer and the contents of the buffer is copied otherwise “\textit{value}” is assumed to contain the immediate data which is saved.</td>
</tr>
<tr>
<td>\texttt{WSIO_WAIT_OK}</td>
<td>If resources are not available the call will block until they are.</td>
</tr>
<tr>
<td>\texttt{WSIO_ATTR_EXPORT}</td>
<td>The attribute will be exported to any children.</td>
</tr>
</tbody>
</table>

**wsio_get_attribute Service**

This service is used to retrieve the current value of an attribute associated with the interface identified by the parameter “\textit{isc}”. The value returned depends upon how the attribute was created. If the attribute was created with the flag \texttt{WSIO\_COPYDATA} then \texttt{wsio\_io\_get\_attribute()} will copy all of the saved data to the location referenced by “\textit{value}” hence “\textit{value}” must reference a buffer large enough. If the flag \texttt{WSIO\_COPYDATA} was not set then the immediate data is returned.

The parameter “\textit{size}” indicates how many bytes were transferred.

**wsio_sizeof_attribute Service**

This service returns the size of an attribute identified by the “\textit{name}” and “\textit{isc}” parameters.

**wsio_destroy_attribute Service**

This service is called to destroy an attribute.
Dynamic Kernel Memory Allocation

The Arena Allocator interfaces provide kernel modules a new way of dynamically allocating kernel memory. These interfaces replace the legacy kmalloc() and kfree() kernel memory allocation interfaces. Features of the Arena Allocator are object caching, better fault isolation, reduced memory fragmentation, and improved scaling.

Object caching can be used to improve performance. With object caching, a kernel module can avoid expensive initialization after each allocation. Since object caching is managed by the kernel, it allows the kernel to reclaim memory when there is memory pressure and it removes the burden of managing this cache by each kernel module.

The Arena Allocator provides better diagnostics by isolating memory objects to private arenas (i.e., memory pools). Incorrect use of de-allocated memory through stale pointer references can lead to memory corruption of other memory objects in the kernel. Such memory corruption problems are extremely time consuming and difficult to debug. With the Arena Allocator, stale pointer references can be contained within a single arena, thus making it easier to isolate defects.

In addition, special attributes can be applied to each arena. For example, certain types of memory objects may require stricter alignment or they may require contiguous physical memory. These and other attributes may be specified when an arena is created.
**Arena Object Categories**

All dynamic memory allocations in the kernel can be categorized as fixed size objects and variable size objects. An object is considered a fixed size object, when the size of the object remains the same at every invocation of the call to allocate memory. An object is considered a variable sized object when the size does not remain the same at every invocation of the call to allocate memory.

Each type of object may have its own arena. Arenas that are associated with fixed size objects are called fixed arenas and arenas that are associated with variable size objects are called variable arenas. Before an allocation can be requested, an arena must be created by calling `kmem_arena_create()`. Each arena is identified by a unique handle returned by `kmem_arena_create()`, and all allocations are made using the handle.

Fixed size allocations are requested by calling `kmem_arena_alloc();` variable size allocations are requested by calling `kmem_arena_varalloc()`. The allocated kernel memory is deallocated by calling `kmem_arena_free()`. When the module is done using the arena and has deallocated all associated allocated objects, the arena is removed by calling `kmem_arena_destroy()`.

Objects that can be accommodated in a 4K page are considered “small objects”. Objects that cannot be accommodated within a 4K page, but are less than or equal in size to 16K bytes, are considered “large objects”. For example, an object with a requested size of 1200 bytes but requiring a 3K alignment is considered a large object. Objects greater than 16K are considered “Xlarge objects”. By default, Xlarge objects are not cached and do not have a free list, but this attribute can be overridden when the arena is created.

**Arena Allocator Interfaces**

The interfaces dynamically allocate kernel memory as listed:

- `kmem_arena_attr_init()` - This function initializes the attribute data structure `kmem_arena_attr_t`, which is then passed to `kmem_arena_create()`. The purpose of this function is to allow future extensions to the attributes in a binary compatible manner and to fill in the default attribute values.

- `kmem_arena_create()` - This function creates either a fixed size or variable size arena. It is called once and before any allocations from an arena are made — preferably when the kernel module is loaded or installed. Default and non-default attributes for the arena are passed through the attributes parameter. If only default attributes are needed, `NULL` can be passed as the attributes parameter.

- `kmem_arena_alloc()` - This function returns fixed size allocations. The first parameter is the handle returned from the call to `kmem_arena_create()` that created the fixed size arena.

- `kmem_arena_varalloc()` - This function returns variable size allocations. The first parameter is the handle returned from the call to `kmem_arena_create()` that create the variable size arena.

- `kmem_arena_free()` - This function deallocates the memory allocated by either `kmem_arena_alloc()` or `kmem_arena_varalloc()`. The kernel normally returns the memory to the same arena from which the memory originated.
kmem_arena_destroy()  This function removes the arena. The parameter passed is the handle returned from kmem_arena_create(). The kernel module must ensure that all memory allocated using the handle have been deallocated before removing the arena.

kmem_arena_attr_t  This structure specifies attributes for creating an arena.

kmem_handle_t  This is an opaque type used to identify arenas.

Arena allocator interfaces are declared in the header <sys/vm_arena_iface.h>. Refer to the respective reference pages for these interfaces in the HP-UX 11i v1 Driver Development Reference Guide.

Fixed Size and Variable Size Arenas

Arenas come in two basic types: fixed size and variable size. This refers to the size of the objects which can be allocated from the arena:

1. Fixed Size Arena — Objects allocated from this arena are of the same size.
2. Variable Size Arena — Objects allocated from this arena may vary in size.

The advantages of fixed size arenas are:

1. Allocations are faster than variable arenas.
2. Object Caching can be used to improve performance.
3. Efficient memory utilization since all objects are the same size.

Variable size arenas can be used to allocate small objects, large objects and Xlarge objects. Xlarge objects are not cached by default. Since there is higher memory consumption in using arenas with variable size objects, HP recommends that each kernel module create one arena for all its variable size objects.

Variable size objects have default alignments. Kernel modules cannot specify the alignment attribute kat_align to create variable size arenas; kat_align may optionally be specified to create fixed size arenas. Both variable size and fixed size arenas, however, may be created with KAT_ALIGN_ON_SIZE set in the kat_flags attribute, which is a distinct attribute from kat_align.

The default alignment for allocated objects will be at least double-word (64-bit) aligned to satisfy the alignment required by C data structures. Kernel modules must not assume any other default alignment for allocated objects.

Arena Creation

Arenas are created using kmem_arena_create(), which returns a unique arena handle. This handle is used for all allocations and de-allocations. Arenas should be created during module initialization, when loaded for DLKM drivers and when installed for statically linked drivers.

Arena attributes are initialized to default values by kmem_arena_attr_init(). Among the attributes that can be modified by a kernel module are the following:

Constructor():

Each time the arena is refilled by the kernel, the refill objects are initialized by calling the constructor function. The constructor function is not called for objects cached in the free list. The constructor function is optionally specified in kat_ctor() to implement object caching. The default is no object caching.
Destructor():

The object will be dismantled by calling the destructor function every time an object is reclaims from an arena. The destructor function is optionally specified in `kat_dtor()` to implement object caching.

Alignment:

By default, allocated objects will be at least double-word (64-bit) aligned. The alignment can be changed to other than the default alignment (only for fixed size arenas) by setting the alignment value in `kat_align`.

Number of objects per refill:

For sizes less than a page, the kernel will split a 4K page by default into multiple objects during refill. For object sizes 4K and greater only one object is refilled by default. For heavily used arenas, this behavior can be changed to refill greater quantities to improve performance by setting the refill value in `kat_refillcnt`.

Maximum Allocation:

Limit the number of objects allocated to this arena by setting the max count in `kat_maxcnt`. This attribute can be used to detect memory leaks and prevent bringing down the system.

Minimum number of objects in free list:

Advise the kernel to keep a minimum number of objects in the free list per SPU during garbage collection by setting the value in `kat_minfcnt`. This attribute is an advisory for small objects. The garbage collector will skip the free list if the length is less than this number. If the length of the free list is larger than this number, the garbage collector will attempt to free as many pages as possible to reduce memory fragmentation. Once all complete pages are collected the length of the free list could be less than this number. For large objects, this value will be enforced.

Multi Cache Line Size:

By default, the kernel shares the same cacheline as the object to store the object header. This might be a problem on non-coherent I/O machines when the object is used for DMA. In such instances, this attribute should be used by setting `KAT_MULTICACHE_SIZE` in `kat_flags`.

No Large Page:

Instruct the kernel to use 4K pages rather than a superpage (large pages) by setting `KAT_NO_LGPG` in `kat_flags`. This attribute is required by certain drivers in order to remap a page.

Cache Xlarge Objects:

By default, Xlarge objects are not cached. This attribute is overridden by specifying `KAT_CACHE_XLARGE_OBJECTS` in `kat_flags`. 
Fixed size arena with default attributes:

```c
kmem_handle_t bar_handle;
struct bar { ...};

void
mymodule_init(void)
{
    bar_handle = kmem_arena_create(sizeof(struct bar),
        "MYMODULE_ARENA", NULL, M_WAITOK)
    ...
}

void
mymodule_foo(void)
{
    struct bar *my_obj;

    my_obj = kmem_arena_alloc(bar_handle, M_WAITOK);
    ...
    kmem_arena_free(my_obj, M_WAITOK);
}
```

Overriding the default attributes:

```c
kmem_handle_t bar_handle;
struct bar { ...};

void
mymodule_init(void)
{
    kmem_arena_attr_t attr;

    kmem_arena_attr_init(&attr, sizeof(kmem_arena_attr_t));
    attr.kat_flags |= KAT_ALIGN_ON_SIZE;

    bar_handle = kmem_arena_create(sizeof(struct bar),
        "MYMODULE_ARENA", &attr, M_WAITOK);
    ...
}

void
mymodule_foo(void)
{
    struct bar *my_obj;

    my_obj = kmem_arena_alloc(bar_handle, M_WAITOK);
    ...
    kmem_arena_free(my_obj, M_WAITOK);
}
Variable size arena with default attributes:

```c
kmem_handle_t bar_handle;
struct bar {...};

void
mymodule_init(void)
{
    bar_handle = kmem_arena_create(0, "MYMODULE_ARENA",
                                   NULL, M_WAITOK);
    ...
}

void
mymodule_foo(void)
{
    struct bar *my_obj;

    my_obj = kmem_arena_varalloc(bar_handle,
                                  sizeof(struct bar), M_WAITOK);
    ...
    kmem_arena_free(my_obj, M_WAITOK);
}
```
Object Caching

A vast majority of allocations in the kernel are for fixed size objects. This allows each module the option to cache objects and avoid costly initialization. Some of the benefits of object caching are:

- Avoid costly construction of objects after each allocation.
- Avoid having each kernel module write code to manage the cache.
- Allows the kernel to efficiently reclaim memory when there is memory pressure.
- Allows caching a stack of objects. A cached object can have a linked list of other cached objects.

Consider the situation where a larger number of objects have associated spinlocks. A typical allocation of an object without object caching is as follows:

```c
kmem_handle_t bar_handle;

void mymodule_init(void)
{
    bar_handle = kmem_arena_create(sizeof(struct bar), "BAR_ARENA",
                                   NULL, M_WAITOK); _
}

#define MYMODULE_SPINLOCK_ORDER 13

void
mymodule_foo(void)
{
    struct bar *my_obj;

    my_obj = kmem_arena_alloc(bar_handle, M_WAITOK);
    my_obj->lock = alloc_spinlock(MYMODULE_SPINLOCK_ORDER, "mymodule_spinlock");
    do_useful_work();
    my_obj->lock = dealloc_spinlock(my_obj->lock);
    kmem_arena_free(my_obj, M_WAITOK);
}
```
The call to `alloc_spinlock()` can be expensive since it must allocate memory for the lock and then initialize the lock. The call to `alloc_spinlock()` can be moved to a constructor function, thus avoiding the call to `alloc_spinlock()` after each allocation. The constructor function `ctor()` is called by the kernel only when the cache (i.e., free list) is empty and new objects are created. The code with the new allocator can be written as follows:

```c
void mymodule_init(void)
{
    kmem_arena_attr_t attr;

    kmem_arena_attr_init(&attr, sizeof(kmem_arena_attr_t));
    attr.kat_ctor = bar_ctor;
    attr.kat_dtor = bar_dtor;

    bar_handle = kmem_arena_create(sizeof(struct bar), "BAR_ARENA",
                                    &attr, M_WAITOK);
    ... 
}

int bar_ctor(struct bar *my_obj, size_t size, int flags)
{
    my_obj->lock = alloc_spinlock(MYMODULE_SPINLOCK_ORDER, "mymodule_spinlock);
    return 0; // 0 : success; != 0 : failure
}

void bar_dtor(struct bar *my_obj, size_t size, int flags)
{
    dealloc_spinlock(my_obj->lock);
}

void mymodule_foo(void)
{
    struct bar *my_obj;

    my_obj = kmem_arena_alloc(bar_handle, M_WAITOK);
    do_useful_work();
    kmem_arena_free(my_obj, M_WAITOK);
}
```
Object Constructors and Destructors

To use object caching, the kernel module needs to specify a constructor function and a destructor function as attributes when creating an arena. The constructor function and destructor function are callback functions. The constructor function will be called when the kernel needs to refill the free list of objects. The destructor function is called when the kernel removes objects from the free list during garbage collection and when deallocating noncached objects. The constructor function is required when using object caching, but the destructor function is optional. The prototype for the constructor function and destruction function are:

```c
int (*ctor)(void *addr, size_t size, int flags)
void (*dtor)(void *addr, size_t size, int flags)
```

The constructor function can allocate other objects from other arenas. The constructor function cannot allocate objects from the same arena associated with the constructor function.

The flags parameter in the constructor function and destructor function correspond to the flags specified for the initial allocation. It is the responsibility of the constructor and destructor functions to honor these flags. If the client specifies `M_NOWAIT`, then the constructor function cannot block (i.e., sleep). The constructor function should return a non-zero value if there is an error. If the kernel detects an error returned by the constructor function, NULL is returned to the caller of `kmem_arena_alloc()`.

For small objects, the kernel will refill many objects at once. In this case, the constructor function is called repeatedly for each object. If the constructor function fails for an object, then the destructor function is called for all previously constructed objects.

The constructor function can recursively allocate objects from other arenas that also have constructor functions. This can be used to link together a stack of objects. It is the responsibility of the kernel module to ensure that these recursions are safe. Since there is a hard limit on the kernel stack size, the number of recursions should be kept small.

There is no mechanism to return the error returned from the constructor function to the caller of `kmem_arena_alloc()`. If a complicated error recovery is necessary, then it is best done after allocation. For example, if a constructor function can fail and this error needs to be communicated to the caller, then such initializations should be done after allocation.

The destructor function should deallocate all objects allocated by the constructor, if any. The destructor function should honor all flags passed to the destructor function. No flags are defined at present. The destructor function should not fail, because there is no mechanism to convey such an error since the destructor function is executed asynchronous to the call to `kmem_arena_free()`.

**CAUTION**  
DLKM drivers that can be unloaded should not specify a destructor function. The kernel may asynchronously invoke the destructor function after the driver has destroyed the arena and unloaded itself. If the destructor function has been unloaded with the driver, the kernel will crash when it tries to call the destructor function.
Legacy Kernel Memory Allocation Interfaces

The legacy interfaces kmalloc() and kfree() are supported, although their use is discouraged. The kernel creates a variable size compatibility arena for each of the M-types (e.g., M_DMA, M_IOSYS, M_IHV) of the old interface.

```c
void *kmalloc (size_t size, int type, arena_flags_t flags)
```
Allocates an object from one of the compatibility arenas.

```c
void kfree (caddr_t va, int type)
```
Deallocates object allocated by kmalloc().

Over time, the number of compatibility arenas is expected to decrease as kernel modules migrate over to using the Arena Allocator interfaces. HP’s long-term plan is to make kmalloc() a simple, generic kernel memory allocator. M-type values will be pruned from malloc_type_t over time until the type parameter passed to kmalloc() is eventually ignored in the implementation.

Binary compatibility with future releases of HP-UX will only be guaranteed for the “type” value of 0. For example:

```c
p = kmalloc(size, 0, M_WAITOK);
kfree(p, 0);
```

Owners of kernel modules are advised to use the Arena Allocator interfaces and to stop using the legacy kmalloc() and kfree() interfaces.
Finding the Physical I/O Address

Drivers need a way to find the processor’s view of the physical I/O address obtained from a device’s PCI BAR. By using kernel interfaces that are currently available to drivers, the physical I/O address can be obtained in a manner that is independent of the processor and platform implementation.

The sample code with explanatory comments is shown. Error handling is not shown to keep the code simple to understand.

```c
uint32_t barvalue; /* Value read from a PCI BAR */
caddr_t baraddr; /* PCI BAR's view of I/O address */
void *physaddr; /* Processor's view of I/O address */
void *kvaddr; /* Kernel virtual address of I/O */
void *uvaddr; /* User virtual address of I/O */

/*
 * Get the I/O address to which the PCI device responds.
 * This example assumes a 32-bit I/O address. For devices
 * that support 64-bit I/O addresses, two config registers
 * need to be read and combined to form the 64-bit address.
 */
pci_read_cfg_uint32_isc(isc, MY_PCI_BAR_OFFSET, &barvalue);
baraddr = (caddr_t)(uintptr_t)(barvalue & ~0xF);

/*
 * Map the PCI BAR's view of its I/O address to a kernel
 * virtual address. The map_mem_to_host() function will
 * translate the PCI BAR's address to the processor's view
 * of the I/O address (which may be different), and then map
 * the processor's I/O address to a kernel virtual address.
 * The ltor() function translates the kernel virtual address
 * to the processor's view of the physical I/O address.
 * 
 * Note1: Physical I/O addresses are not equivalent to
 * kernel virtual addresses on IA64 implementations.
 * 
 * Note2: If WSIO has previously mapped the I/O address
 * (isc->if_reg_ptr != NULL), then do not call
 * map_mem_to_host() to map the I/O address again.
 * WSIO maps for the driver the first BAR of the
 * PCI device that contains an I/O memory address.
 */
kvaddr = map_mem_to_host(isc, baraddr, MY_SIZE_OF_IOMEM);
physaddr = ltor(ldsid(kvaddr), kvaddr);

/*
 * Now that we have the processor's view of the physical
 * I/O address, we should unmap the kernel virtual address.
 * If we do not unmap the kernel virtual address, then a
 * call to kernel_iomap() passing PROT_URW will fail, and on
 * IA64 implementations a call to user_iomap() may also fail.
 */
(void)unmap_mem_from_host(isc, kvaddr, MY_SIZE_OF_IOMEM);

/*
 * Map the physical I/O address for user access.
 */
uvaddr = user_iomap(NULL, physaddr, MY_PAGES_OF_IOMEM);

/*
 * We can also let the kernel share access with the user.
 */
kvaddr = kernel_iomap(NULL, physaddr, MY_PAGES_OF_IOMEM,
PROT_URW);
```
Using user_iomap()

The virtual address returned by user_iomap() will be appropriate to the user process; a 32-bit address is returned for 32-bit processes, and a 64-bit address is returned for 64-bit processes. The call to user_iomap() must be made in user context by the driver; the call cannot be made in interrupt context.

NOTE
If 32-bit and 64-bit user processes wish to share access to the same I/O memory range on an HP-PA platform, a 32-bit user process should get the first mapping before any 64-big user process. In addition, no spinlocks may be held when calling user_iomap().

A driver may call user_iomap() for the same I/O memory range multiple times to allow access to the range by multiple user processes. For each call made to user_iomap(), there should be a corresponding call made to user_iounmap() in the context of the user process before the driver unloads.

It is normal for a call to user_iomap() to fail. Possible reasons for failure include:

- The pages are mapped for kernel-only access with PROT_KRW and the user virtual address is equivalent to the kernel virtual address. If the pages are mapped by the WSIO driver environment or mapped using wsio_map_reg(), protection is set to PROT_KRW.
- The pages overlap with an I/O memory range that has already been mapped by a previous call to user_iomap() or kernel_iomap(). To successfully map an I/O memory range multiple times, the mapping must cover exactly the same range (same paddr and count parameters).

A call to user_iomap() may fail if the driver first calls map_mem_to_host() or the WSIO driver environment has mapped the I/O memory on behalf of the driver. If the driver's isc->if_reg_ptr is not NULL, then WSIO has mapped (up to 8K) of the first PCI BAR in the device that specifies a memory range.

Under certain conditions a call to user_iomap() may succeed after the driver calls map_mem_to_host() for the same physical I/O address. Drivers must not rely on this behavior. The call may succeed when the user virtual address is not equivalent to the kernel virtual address. This behavior may exist when:

- Mapping onto a 32-bit user process on a 64-bit PA implementation.
- Mapping onto a 64-bit user process on a 64-bit PA implementation.
3 Multiprocessing

HP-UX servers and workstations are either uniprocessor or multiprocessor systems. Current and new drivers for either servers or workstations must be written to be multiprocessing safe, because they may eventually run on multiprocessor systems. This chapter covers the kernel services that handle synchronization used by drivers on multiprocessor systems.
Comparing Uniprocessing to Multiprocessing

A **uniprocessor** (UP) system is comprised of exactly one processor. A driver in a UP system may be executing in only one thread at any given time. That thread will either be a kernel thread (the upper half of a driver), or on the **Interrupt Control Stack** (ICS) in a processor interrupt context (the lower half of a driver). The UP driver synchronization model coordinates execution between the driver’s upper and lower halves.

A **multiprocessor** (MP) system is comprised of two or more processors. A driver in an MP system may be executing in multiple threads concurrently. For each processor, the driver may be executing in a kernel thread or in the interrupt context of that processor. The MP driver synchronization model coordinates execution among multiple kernel threads as well as between the driver’s upper and lower halves.

HP-UX is a multiprocessing operating system, and as such, drivers must be written for MP systems. The MP synchronization mechanisms provided by HP-UX are spinlocks and beta semaphores. Drivers that use these synchronization mechanisms will work correctly on both MP and UP systems.
Synchronization Mechanisms

Spinlocks are the most heavily used synchronization mechanism in the HP-UX kernel. They are used to protect data accessible from either a kernel thread or an interrupt context. Only one processor is allowed to own a spinlock at any given time. Other processors that attempt to acquire an owned spinlock will spin and wait for the spinlock to be released by the owning processor.

External interrupts for a processor are disabled for the duration when a processor owns a spinlock or when it attempts to acquire a spinlock. Because external interrupts are disabled, spinlocks must not be owned (i.e., locked) for lengthy periods of time. Likewise, spinlocks must not be held across calls to system services that may block (put the thread to sleep).

Semaphores, which are known in the HP-UX kernel as beta semaphores, provide another synchronization mechanism. They are used to protect data that are accessed by a driver’s upper half (executing in a kernel thread that may block). They can not be used to protect data that are accessed by a driver’s lower half (executing in an interrupt context that can not block).

Beta semaphores provide mutual exclusion where only one kernel thread is allowed to own the semaphore at any given time. Other kernel threads attempting to acquire the semaphore will be blocked until the semaphore is released.

Unlike spinlocks, beta semaphores may be held across calls to system services that may block.

The areas for synchronization are shown in Figure 3-1, “Synchronization Areas.”

Figure 3-1    Synchronization Areas

There are two mechanisms in the HP-UX kernel used for synchronization; Spinlocks, and Beta Semaphores. Spinlocks are held for very short durations, generally only a few milliseconds, and are used to protect data accessed in a kernel thread or interrupt context. Beta Semaphores are only held in a kernel thread context, and usually for longer durations than Spinlocks.

The mechanism for synchronization is shown in Figure 3-2, “Synchronization Mechanism.”
The general process flow for a synchronization mechanism is as follows:

1. Allocate the mechanism — Driver allocates and initializes a spinlock resource. Pointers are maintained to the lock and the data area.
2. Acquire the mechanism — Driver takes ownership of, and locks the spinlock.
3. Release the mechanism — Driver releases ownership of, and unlocks the spinlock.
4. Deallocate the mechanism — Driver terminates usage of the spinlock resource.

**Timing Hazards and Idle Time**

Timing hazards, also known as race conditions, can occur on an MP system. Careful regression testing on MP systems is essential to expose timing hazards that may occur in a driver.

Designs where beta semaphores are owned for lengthy periods of time can cause the idle time of the system to increase as kernel threads are forced to block and wait. This situation can be detected with tools such as `top` and `sar` (see `top (1)` and `sar (1)`), and the optional HP products LaserRX and Glance/UX.
Spinlocks

Spinlocks are the basic locking primitive used by the kernel for short-term locks. When a thread acquires a spinlock, the thread’s current processor becomes the effective owner until the spinlock is released. Threads (processors) waiting to acquire an owned spinlock will spin while waiting — they do not block. For the duration that a processor owns a spinlock, external interrupts to the processor are disabled. External interrupts to the processor are disabled to avoid a potential interruption deadlock. Consider the case where driver code is executing on a processor and owns (i.e., has locked) a spinlock. If external interrupts are not disabled, an interrupt from a device may cause the Interrupt Service Routine (ISR) of the driver to be entered on the same processor. If the driver’s ISR attempts to lock the same spinlock, a deadlock will occur because the spinlock is already owned by the processor. The ISR will spin and wait forever.

A spinlock that is owned by a processor makes other processors spin and wait if they attempt to acquire the same spinlock. The other processors burn CPU cycles without doing useful work when this occurs. Therefore, drivers should be designed to hold spinlocks for only short periods of time. A general rule of thumb is if a spinlock is held for longer than a few milliseconds, then it is being held too long.

Spinlock Routines

HP-UX provides the following spinlock routines. Refer to the HP-UX 11i v1 Driver Development Reference Guide for detailed descriptions.

- alloc_spinlock() — allocate and initialize a spinlock resource.
- cspinlock() — conditionally lock a spinlock if the spinlock is not owned.
- dealloc_spinlock() — deallocate a spinlock resource.
- owns_spinlock() — check if the processor owns a spinlock.
- spinlock() — acquire (lock) a spinlock.
- spinunlock() — release (unlock) a spinlock.
Beta Semaphores

Beta semaphores are mutually-exclusive, blocking semaphores. When a thread acquires a beta semaphore, it is the owning thread until the beta semaphore is released. The owning thread may subsequently block (i.e., sleep) and still keep ownership. Threads waiting to acquire an owned beta semaphore are blocked.

Since blocking may occur, beta semaphores must not be acquired by a driver while executing in the interrupt context of a processor.

Beta Semaphore Routines

HP-UX provides the following beta semaphore routines. Refer to the *HP-UX 11i v1 Driver Development Reference Guide* for detailed descriptions.

- `b_cpsema()` — conditionally acquire (lock) a beta semaphore if it is not currently locked.
- `b_initsema()` — initialize a beta semaphore.
- `b_owns_sema()` — test whether a beta semaphore is owned by the current thread.
- `b_psema()` — acquire (lock) a beta semaphore.
- `b_vsema()` — release (unlock) a beta semaphore.
Deadlocks

If a driver acquires beta semaphores or spinlocks in an incorrect order, a deadlock may occur.

The classic illustration of a deadlock is the case of processes A and B which both need resources C and D to complete an activity. If process A locks resource C and process B locks resource D, each will be blocked forever waiting for the resource held by the other process.

To avoid deadlocks, each thread must acquire its locks in the same order. In the above example, processes A and B must acquire resource C before they try to acquire resource D.

Rules for Lock Acquisition

Beta semaphores and spinlocks (and the resources they protect) are assigned a lock order, which is used as follows:

- When a thread of execution acquires a spinlock unconditionally, the order of the requested spinlock must be greater than the order of any spinlock the processor already holds.
- When a kernel thread acquires a semaphore unconditionally, the order of the requested semaphore must be greater than the order of any semaphore the kernel thread already holds.
- If the orders of the acquired and held beta semaphores are equal, both beta semaphores must have the deadlock safe option set. This option is set by ORing the order with the \texttt{SEMA\_DEADLOCK\_SAFE} bit when the semaphore is initialized.
- Spinlocks have the highest order. A thread of execution must acquire all beta semaphores it requires before it acquires a spinlock.

Lock Orders

The header file \texttt{<sys/semglobal.h>} contains the lock orders used by HP supplied kernel services. Drivers typically choose a lock order that is low in value so that the driver can hold its own spinlock (or beta semaphore) while calling a kernel service.
Synchronization Using sleep() and wakeup()

In addition to spinlocks and beta semaphores, HP-UX provides another synchronization mechanism using the system services sleep() and wakeup().

- Typically, the upper half of a driver will start an asynchronous activity and wait for the lower half to complete the activity.
- The system service sleep() is called by the driver's upper half to block the kernel thread and put it on a sleep queue.
- The driver's lower half calls wakeup() to take the kernel thread off the sleep queue and to awaken the thread.

Race Condition Prevention

If the wakeup() routine is called before the kernel thread has been put on a sleep queue, a race condition may exist. To prevent this condition:

- A call to get_sleep_lock is made before calling sleep().
- This routine acquires a sleep queue spinlock that will be released by sleep() after the kernel thread has been put to sleep.
- If the same sleep queue spinlock is not already held, the routine wakeup() acquires the sleep queue spinlock before awakening the sleeping thread. It then awakens the thread, and releases the sleep queue spinlock.

The process is demonstrated in the following code examples:

Drivers typically call get_sleep_lock(), start an asynchronous activity, then call sleep():

```c
(void)get_sleep_lock(wait_chan);
start_async_activity();
(void)sleep(wait_chan, PRIBIO);
```

When the asynchronous activity completes, the driver's async_completion() routine calls wakeup():

```c
static void
async_completion(void)
{
    wakeup(wait_chan);
}
```

Shared Data Protection

The routine get_sleep_lock() may also be used to protect data shared between the kernel thread (that will sleep) and the driver's async_completion() routine. For example, the driver's top half looks like:

```c
(void)get_sleep_lock(wait_chan);
start_async_activity();
activity_count++;
(void)sleep(wait_chan, PRIBIO);
```

Note that the incrementing of activity_count is protected by a sleep queue spinlock.
When the asynchronous activity completes, the driver’s `async_completion()` routine shown, calls `get_sleep_lock()` before it decrements `activity_count` and calls `wakeup()`. The routine `wakeup()` checks if the sleep queue spinlock is already held before it tries to acquire the same sleep queue lock. This allows the caller to hold the sleep queue spinlock across a call to `wakeup()`. After the call to `wakeup()` the sleep queue lock must be unlocked.

```c
static void
async_completion(void)
{
    lock_t * sleep_lock;
    sleep_lock = get_sleep_lock(wait_chan);
    if (activity_count) {
        activity_count--;
        wakeup(wait_chan);
    }
    spinunlock(sleep_lock);
}
```

## Reducing Sleep Lock Contention

By protecting shared data using a sleep lock, a driver will increase the potential for contention on the sleep lock. A driver can reduce the likelihood of contention on the sleep lock by using a spinlock that is private to the driver. The driver private spinlock shall be of order <=SLEEP_Q_LOCK_ORDER. For example:

```c
spinlock(my_lock);
start_async_activity();
activity_count++;
(void)get_sleep_lock(wait_chan);
spinunlock(my_lock);
(void)sleep(wait_chan, PRIBIO);
```

**NOTE** The call to `get_sleep_lock()` must be made before the call to `spinunlock()` to avoid the race condition with `wakeup()`.

The associated completion routine containing the call to `wakeup()` is implemented as follows:

```c
static void
async_completion(void)
{
    spinlock(my_lock);
    if (activity_count) {
        activity_count--;
        spinunlock(my_lock);
        wakeup(wait_chan);
    } else {
        spinunlock(my_lock);
    }
}
```
Multiprocessing
Synchronization Using sleep() and wakeup()
4 Writing a Driver

This chapter describes the code most device drivers need to include, and the things you need to do to write a driver.
Suggested Driver Writing Methodology

Read this chapter, and then read the chapter that provides details about your specific driver. For example, if you are writing PCI Drivers, read Chapter 14, “Writing PCI Device Drivers.” If you are writing a loadable driver module, also read Chapter 16, “Writing a DLKM Driver.”

Writing a Driver: Step-by-Step

The steps you take in writing a driver are described in the following sections:

“Step 1: Choosing a Driver Name”
“Step 2: Choosing System Header Files”
“Step 3: Defining Installation Structures”
“Step 4: Identifying Routines for Your Driver”
“Step 5: Writing Configuration Routines”
“Step 6: Writing Entry Point Routines”
“Step 7: Writing Other Driver Routines”
Step 1: Choosing a Driver Name

The driver needs a name. It must be unique to avoid conflict with kernel routines and global variables. Consider using the developers company's name and something that indicates the driver's purpose.

The name is required in four places:

- In a master file in the /usr/conf/master.d directory, described in “Step 2: Create a Master File” in Chapter 5, “Installing Your Driver.”
- In the system configuration file, /stand/system, described in “Step 3: Modify the System File” in Chapter 5, “Installing Your Driver.”
- In the name field of the drv_info_t structure, described in “The drv_ops_t Structure Type” section.
- As the prefix of the name of the installation routine, driver_install, described in the “Writing a driver_install() Routine” section.

We suggest following the convention in which all installation, entry-point, and other external routines are prefixed with the name of the driver or a distinctive abbreviation. The format is driver_routine(), where driver is the driver's name and routine is the standard part of the routine name, as in driver_open().

For example, if the company is Wonderful Software, and a MUX driver is being written, give the name wondermux. When it installs the driver, the kernel will call a routine named wondermux_install().

In this manual, mydriver and skel are used as sample driver names.
Step 2: Choosing System Header Files

A driver's header contains definitions of data structures, macros, and constants that are used in the compilation of the driver module. Most of these definitions can be created by including other header files in the driver's header.

System-Defined Header Files

This section lists header files that might be needed to include in the driver. To find out which headers the driver requires, see the reference pages in the HP-UX 11i v1 Driver Development Reference and HP-UX Reference for each kernel call and data structure the driver uses.

NOTE Because it redefines some entries from other header files, the <sys/wsio.h> file must be specified in the last #include statement in the driver's header.

Header Files for All Drivers

- /usr/include/sys/buf.h
  The buf I/O buffer structure.
- /usr/include/sys/conf.h
  Device switching tables and the drv_ops_t and drv_info_t structures.
- /usr/include/sys/errno.h
  Error numbers returned to applications.
- /usr/include/sys/file.h
  Flags for the open() system call.
- /usr/include/sys/io.h
  The ISC table.
- /usr/include/sys/malloc.h
  Things needed for acquiring and releasing memory.
- /usr/include/sys/sysmacros.h
  Some commonly used fields in some drivers’ minor numbers.
- /usr/include/sys/uio.h
  The uio structure and its elements.
- /usr/include/sys/user.h
  Things used by some kernel routines.
- /usr/include/sys/wsio.h
  Data and macros used in the WSIO context, including the wsio_drv_info and wsio_drv_data structures. The header of each driver and each WSIO dependent pseudo driver must include this header file.
Header Files for Disk Drivers

- /usr/include/sys/diskio.h
  Flags for the ioctl() system call for use with disks.
- /usr/include/sys/floppy.h
  Flags for the ioctl() system call for use with floppy disks.

Header Files for Tape Drivers

- /usr/include/sys/mtio.h
  Flags for the ioctl() system call for use with magnetic tapes.
Step 3: Defining Installation Structures

Include some data structures in the driver. Which data structures the driver requires depends on what the device or interface-card driver does. They can appear in either the .c file of the driver, or in a header file that is included in the driver.

- **drv_ops_t** — Driver-specific fields that all CDIOs use.
- **drv_info_t** — Driver-specific defined by all CDIOs. CDIOs use these fields to configure the device.
- **wsio_drv_data** — Driver specific fields defined by WSIO drivers.
- **wsio_drv_info_t** — Pointers to the other three structures in the list.

The rest of this section describes these data structures in detail and gives examples of their use in skeleton drivers' headers.

### The drv_ops_t Structure Type

The `drv_ops_t` structure type, defined in `<sys/conf.h>` and shown here, contains pointers to all a driver's entry points. A `drv_ops_t` structure must be statically allocated.

```c
typedef struct drv_ops
{
    int (*d_open)(); /* block and character */
    int (*d_close)(); /* block and character */
    int (*d_strategy)(); /* block */
    int (*d_dump)(); /* NULL {obsolete} */
    int (*d_psize)(); /* block */
    int (*reserved0)(); /* NULL */
    int (*d_read)(); /* character */
    int (*d_write)(); /* character */
    int (*d_ioctl)(); /* character */
    int (*d_select)(); /* character */
    int (*d_option1)(); /* NULL */
    pfilter_t *pfilter; /* block and character */
    int (*reserved1)(); /* NULL */
    int (*reserved2)(); /* NULL */
    int (*reserved3)(); /* NULL */
    int d_flags; /* block and character */
} drv_ops_t;
```

The relevant fields are described in Table 4-1, “Device Driver Fields, `drv_ops_t` Structure.” All other fields in `drv_ops_t` should be NULL. Except as noted, entry points that do not apply to your driver or that your driver does not provide (for example, `d_psize()` has no meaning for a printer) should be NULL.
### Table 4-1  Device Driver Fields, drv_ops_t Structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Device Type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>d_open()</td>
<td>both</td>
<td>Pointer to the driver_open() routine, which enables a device for subsequent operations. If the device is off line, does not exist, or cannot be accessed, an error should be returned. When appropriate, it is permissible for the open routine to do nothing.</td>
</tr>
<tr>
<td>d_close()</td>
<td>both</td>
<td>Pointer to the driver_close() routine, which disables interrupts, resets a device, frees resources, and performs other tasks required when a device is closed.</td>
</tr>
<tr>
<td>d_strategy()</td>
<td>block</td>
<td>Pointer to the driver_strategy() routine, which queues I/O requests for either reading or writing. Drivers of character devices often call physio() from their read and write routines; physio() calls the strategy routine passed in as a parameter, but it is not an entry point into a character driver.</td>
</tr>
<tr>
<td>d_psize()</td>
<td>block</td>
<td>Pointer to the driver_psize() routine. For a swapping device, it should return the size of the swap partition. Consider writing this routine only if your device is used for swapping.</td>
</tr>
<tr>
<td>d_read()</td>
<td>character</td>
<td>Pointer to the driver_read() routine, which should return the requested data transferred from the device.</td>
</tr>
<tr>
<td>d_write()</td>
<td>character</td>
<td>Pointer to the driver_write() routine, which should write the requested data to the device.</td>
</tr>
<tr>
<td>d_ioctl()</td>
<td>character</td>
<td>Pointer to the driver_ioctl() routine, which sends control information to, or gets it from, a device. You can also use it to provide driver-dependent functions that are not implemented by other routines.</td>
</tr>
<tr>
<td>d_select()</td>
<td>character</td>
<td>Pointer to the driver_select() routine, which you can use to test for I/O completion and driver-dependent exception conditions. If your device is always ready for reading or writing, you can put seltrue in the d_select() field. If you do, calls to select() always return true without invoking your driver.</td>
</tr>
<tr>
<td>pfilter</td>
<td>both</td>
<td>Pointer to a pfilter_t structure. Use the &amp;cpd_pfilter pointer. This structure provides backward compatible routines for disk structures with fixed partitions, such as the Series 800 computers before the availability of the Logical Volume Manager (LVM). The &amp;cpd_pfilter pointer is required for such disks; it is ignored under other conditions (or NULL can be used).</td>
</tr>
</tbody>
</table>
The `drv_info_t` Structure Type

All CDIOs use the driver-specific fields in the `drv_info_t` structure type, defined in `<sys/conf.h>` and shown below. A `drv_info_t` structure must be statically allocated.

```
typedef struct drv_info {
    char *name;    /* Name of driver */
    char *class;   /* Device class ('disk', etc.) */
    ubit32 flags;  /* Device flags (see below) */
    int    b_major; /* Block device major number */
    int    c_major; /* Character device major number */
    cio_t  *cdio;   /* Drivers set this to NULL */
    void   *gio_private;    /* Drivers set this to NULL */
    void   *cdio_private;   /* Drivers set this to NULL */
} drv_info_t;
```

The relevant fields are described below. All other fields in a `drv_info_t` should be NULL.

<table>
<thead>
<tr>
<th>Field</th>
<th>Device Type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>both</td>
<td>The bitwise OR of flag values that indicate special features of the device. The flags give information about the device to the kernel. Drivers receive this information, but usually only validate it. Use 0 if no flags are set. The flags used by drivers are:</td>
</tr>
<tr>
<td>class</td>
<td>both</td>
<td>C_ALLCLOSES flag forces a call to <code>driver_close()</code> on every closing of the device. The default action is to call the driver’s close routine only on the last close of the device.</td>
</tr>
<tr>
<td>flags</td>
<td>both</td>
<td>C_NODELAY tells the kernel to not wait for a write request to complete on this device. The default action is to wait for a write request to complete before returning control to the calling process.</td>
</tr>
<tr>
<td>b_major</td>
<td>both</td>
<td>C_MGR_Is_MP identifies the driver as safe for use in a multiprocessing environment.</td>
</tr>
<tr>
<td>c_major</td>
<td>both</td>
<td>C_MAP_BUFFER_TO_KERNEL indicates the device driver needs <code>physio()</code> to remap a user buffer to kernel space prior to calling the <code>driver_strategy()</code> routine. This is the pre 10.0 behavior of <code>physio()</code>.</td>
</tr>
</tbody>
</table>

Table 4-1 Device Driver Fields, `drv_ops_t` Structure (Continued)
Step 3: Defining Installation Structures

DRV_SCAN Driver supports bus scanning.

DRV_MP_SAFE Driver provides its own multiprocessing protection.

DRV_SAVE_CONF Save configuration information to /etc/ioconfig. This file retains potentially volatile information, such as dynamic major numbers and card instance numbers, across reboots.

b_major The major number if this is a block device. Set it to -1 for dynamic assignment or if it is not a block device.

c_major The major number if this is a character device. Set it to -1 for dynamic assignment or if it is not a character device.

NOTE The values specified for b_major and c_major override the values the developer enters in a master file in /usr/conf/master.d see “Step 2: Create a Master File” in Chapter 5, “Installing Your Driver.”

The wsio_drv_data_t Structure Type

The wsio_drv_data_t structure type, defined in <sys/wsio.h> and displayed here, contains driver-specific fields for WSIO drivers:

typedef struct wsio_drv_data {
    char *drv_path;
    /* for matching probes with drivers */
    sbit8 drv_type;
    /* driver type: device or interface */
    ubit32 drv_flags;
    /* pre-10.0 or post-10.0 driver */
    int (*drv_minor_build)();
    /* minor number formatter */
    int (*drv_minor_decode)();
    /* minor number interpreter*/
    int (*drv_get_minors)();
    int (io_path_mgr)();
} wsio_drv_data_t;

The fields are described below.

Field Purpose

drv_path Follow these guidelines:

- For device drivers, drv_path is typically a string that contain the interface card's type and the device's class. For example, scsi_disk.
- For interface drivers, drv_path should match the card's type. For example, scsi instead of ext_bus.
- For pseudo drivers, drv_path should match the card's class. For example, graphics.

drv_type One of the following values:

- TDEVICE — The driver controls a hardware device.
- TINTERFACE — The driver controls an interface card, or is monolithic.
Writing a Driver
Step 3: Defining Installation Structures

drv_flags One of the following values:

- **DRV_CONVERGED** — The driver meets the HP-UX Release 10.0 Converged I/O specifications.
- **NOT_CONVERGED** — The driver conforms to the pre-Release 10.0 unconverged specifications.

drv_minor_build Pointer to your minor number formatter.
drv_minor_decode Pointer to your minor number interpreter.
drv_get_minor, io_path_mgr Drivers should set these fields to NULL.

The wsio_drv_info_t Structure Type

The wsio_drv_info_t structure type, defined in `<sys/wsio.h>` and shown below, contains pointers to the three preceding data structures. The last field is a driver version. Driver writers can use it to indicate different versions of their driver. Drivers can use the WSIO define to set the value of this field. The define is WSIO_DRV_CURRENT_VERSION.

```c
typedef struct wsio_drv_info {
    drv_info_t *drv_info;
    drv_ops_t *drv_ops;
    wsio_drv_data_t *drv_data;
    unsigned int driver_version;
} wsio_drv_info_t;
```

Sample Header for a Device Driver

Here is a sample header for a character/block disk device driver named *skel*.

```c
#include <sys/conf.h>
#include <sys/wsio.h>

int skel_open();
int skel_close();
int skel_strategy();
int skel_psize();
int skel_read();
int skel_write();
int skel_ioctl();

static drv_ops_t skel_ops =
{
    skel_open,
    skel_close,
    skel_strategy,
    NULL,
    skel_psize,
    NULL,
    skel_read,
    skel_write,
    skel_ioctl,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    C_ALLCLOSES | C_MGR_IS_MP
};
```
static drv_info_t skel_info =
{
  "skel",
  "disk",
  DRV_CHAR | DRV_BLOCK | DRV_SAVE_CONF | DRV_MP_SAFE,
  -1,    /* dynamic major number assignment */
  -1,    /* for block and character devices */
  NULL,
  NULL,
  NULL,
};

static wsio_drv_data_t skel_data =
{
  "scsi_disk",
  T_DEVICE,
  DRV_CONVERGED,
  NULL,
  NULL,
  NULL,
};

static wsio_drv_info_t skel_wsio_info =
{
  &skel_info,
  &skel_ops,
  &skel_data,
  WSIO_DRV_CURRENT_VERSION
};

Sample Header for an Interface Driver

An interface-card driver usually has no entry points, except for configuration and interrupt. However, it is
possible to have an interface driver with special entry points for sending requests directly to the card, instead
of to the device that is connected to the card. Also, you can have a driver that is both an interface driver and a
device driver (sometimes referred to as a monolithic driver). In that case, it would have both standard entry
points and an ISC structure, and so forth.

Here is an example of a header for an interface-card driver.

#include <sys/conf.h>
#include <sys/wsio.h>

/*
 * No entry points for an interface driver,
 * so set all the fields to NULL.
 */
static drv_ops_t skel_ops =
{
  NULL, NULL, NULL, NULL, NULL,
  NULL, NULL, NULL, NULL, NULL,
  NULL, NULL, NULL, NULL, NULL,
  0,
};

static drv_info_t skel_info =
{
  "skel",
  "ext_bus",
};
Step 3: Defining Installation Structures

```c

static wsio_drv_data_t skel_data =
{
    "skel",
    T_INTERFACE,
    DRV_CONVERGED,
    NULL,
    NULL,
    NULL,
    NULL,
};

static wsio_drv_info_t skel_wsio_info =
{
    &skel_info,
    &skel_ops,
    &skel_data,
    WSIO_DRV_CURRENT_VERSION
};
```

Sample Header for a Pseudo Driver

Pseudo drivers do not control hardware, but do have character and/or block entry points in a `drv_ops_t` structure. Generally, a pseudo driver preprocesses information that it then passes to the file system or to a device driver that does control hardware. Normally, it is installed in the CDIO where the device driver is installed (generally, WSIO).

If not installed in the WSIO CDIO, the pseudo driver should define the following header and structures and use the `install_driver()` installation function in `driver_install()` (see “Writing a driver_install() Routine” section). Note the use of the `pseudo` class and the `DRV_PSEUDO` flag.

```c
#include <sys/conf.h>

static drv_info_t my_drv_info =
{
    "my",
    "pseudo",
    DRV_PSEUDO|DRV_CHAR,
    -1,
    -1,
    NULL,
    NULL,
    NULL,
};

static drv_ops_t my_ops =
{
    my_open,
    my_close,
    NULL,
    NULL,
    NULL,
    NULL,
    my_read,
    my_write,
};
```
In the WSIO case, the driver adds the following header and structure to the preceding and uses the `wsio_install_driver()` installation function in `driver_install()` see “Writing a driver_install() Routine”.

```c
#include <sys/wsio.h>

static wsio_drv_data_t my_data = {
    "pseudo",
    T_DEVICE,
    DRV_CONVERGED,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    0
};

static wsio_drv_info_t my_wsio_info = {
    &my_info,
    &my_ops,
    &my_data,
    WSIO_DRV_CURRENT_VERSION
};
```
Step 4: Identifying Routines for Your Driver

Table 4-2, “Configuration and Entry Point Routines,” lists the configuration and entry point routines for the various types of drivers.

<table>
<thead>
<tr>
<th>Driver Type</th>
<th>Configuration Routines</th>
<th>Entry Points</th>
<th>Other Routines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block Device</td>
<td>driver_install()</td>
<td>driver_open()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>driver_dev_init()</td>
<td>driver_close()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_strategy()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_psize()</td>
<td></td>
</tr>
<tr>
<td>Block Pseudo</td>
<td>driver_install()</td>
<td>driver_open()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>driver_dev_init()</td>
<td>driver_close()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_strategy()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_psize()</td>
<td></td>
</tr>
<tr>
<td>Character Device</td>
<td>driver_install()</td>
<td>driver_open()</td>
<td>driver_strategy()</td>
</tr>
<tr>
<td></td>
<td>driver_dev_init()</td>
<td>driver_close()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_read()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_write()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_ioctl()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_select()</td>
<td></td>
</tr>
<tr>
<td>Character Pseudo</td>
<td>driver_install()</td>
<td>driver_open()</td>
<td>driver_strategy()</td>
</tr>
<tr>
<td></td>
<td>driver_dev_init()</td>
<td>driver_close()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_read()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_write()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_ioctl()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_select()</td>
<td></td>
</tr>
<tr>
<td>Interface</td>
<td>driver_install()</td>
<td>driver_open()</td>
<td>driver_is()</td>
</tr>
<tr>
<td></td>
<td>driver_attach()</td>
<td>driver_close()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>driver_if_init()</td>
<td>driver_read()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>driver_probe()</td>
<td>driver_write()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_ioctl()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_select()</td>
<td></td>
</tr>
<tr>
<td>Block Monolithic</td>
<td>driver_install()</td>
<td>driver_open()</td>
<td>driver_isr()</td>
</tr>
<tr>
<td></td>
<td>driver_attach()</td>
<td>driver_close()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>driver_dev_init()</td>
<td>driver_strategy()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>driver_if_init()</td>
<td>driver_psize()</td>
<td></td>
</tr>
<tr>
<td>Character Monolithic</td>
<td>driver_install()</td>
<td>driver_open()</td>
<td>driver_isr()</td>
</tr>
<tr>
<td></td>
<td>driver_attach()</td>
<td>driver_close()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>driver_dev_init()</td>
<td>driver_read()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>driver_if_init()</td>
<td>driver_write()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>driver_probe()</td>
<td>driver_ioctl()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_select()</td>
<td></td>
</tr>
</tbody>
</table>

a. A device, pseudo, or monolithic driver can control both block and character data, in which case it can specify the entry points for both types.
b. In a combined block and character driver, the two driver_strategy() routines can often be combined.
c. An interface driver does not have system-called entry points. It can provide a driver-dependent interface through isc->ifsw, as described.
d. A monolithic driver combines a device driver and an interface driver.
Choosing Needed Routines

The driver may not need all the routines shown in Table 4-2, “Configuration and Entry Point Routines.” Choose the ones needed from the following description.

Except for driver_install(), the developer may select any arbitrary name for each of these routines. For convenience in maintenance and debugging, we recommend that the developer use the names shown, substituting the driver’s name for “driver”. The driver_install() routine must be named as shown, with the driver’s name substituted for “driver”.

Driver Type
These are the principal driver types. The block and character types can be combined into the same driver, in which case the driver can specify the entry points for both types. See Table 4-3, “Block Driver,” and Table 4-4, “Character Driver.”

Configuration Routines
These routines are executed when the system boots.

- Every driver requires a driver_install() routine.
- A device or pseudo driver can also have a driver_dev_init() routine.
- An interface driver requires a driver_attach() routine. It can also have a driver_if_init() routine and a driver_probe() routine.
- A monolithic driver uses a combination of device and interface driver routines.
- driver_probe() is established by an interface or device driver to search hardware paths and identify interface cards and device drivers. Most HP-supported buses already have probe routines.

Entry Points
These routines are the interface between system calls and the driver. They are specified in the drv_ops_t header structure and executed by corresponding system calls from a user program. If a device does not perform a certain function, or you don’t ask it to, the corresponding routine is not needed. For example, a printer often has no need for a read routine.

Table 4-3  Block Driver

<table>
<thead>
<tr>
<th>System Call</th>
<th>Executes</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>open()</td>
<td>driver_open()</td>
<td>Open the device</td>
</tr>
<tr>
<td>close()</td>
<td>driver_close()</td>
<td>Close the device</td>
</tr>
<tr>
<td>read()</td>
<td>driver_strategy()</td>
<td>Perform block read</td>
</tr>
<tr>
<td>write()</td>
<td>driver_strategy()</td>
<td>Perform block write</td>
</tr>
<tr>
<td>kernel</td>
<td>driver_psize()</td>
<td>Specify swap partition size</td>
</tr>
</tbody>
</table>
Interface drivers have no entry points since they are called only by device drivers and not from user programs.

Other Routines

These routines are not defined in the header structures. Instead, they are defined within the driver and passed as parameters to other routines. Those listed are a sampling of such routines.

- **driver_isr()** is the interrupt service routine for a device or interface, established by the interface driver.

- **driver_strategy()** for a character driver can be accessed from the **driver_read()** and **driver_write()** routines to transfer data in blocks rather than byte by byte using **physio()**. In a combination block and character driver, the two **driver_strategy()** routines can often be combined.

- **driver_minphys()** is used for a character driver which calls **physio()** and requires that subsequent **driver_strategy()** calls be made with different size buffers than that provided by the standard **minphys()** routine.

### Table 4-4 Character Driver

<table>
<thead>
<tr>
<th>System Call</th>
<th>Executes</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>open()</td>
<td>driver_open()</td>
<td>Open the device</td>
</tr>
<tr>
<td>close()</td>
<td>driver_close()</td>
<td>Close the device</td>
</tr>
<tr>
<td>read()</td>
<td>driver_read()</td>
<td>Perform character read</td>
</tr>
<tr>
<td>write()</td>
<td>driver_write()</td>
<td>Perform character write</td>
</tr>
<tr>
<td>ioctl()</td>
<td>driver_ioctl()</td>
<td>Perform special command</td>
</tr>
<tr>
<td>select()</td>
<td>driver_select()</td>
<td>Test I/O completion</td>
</tr>
</tbody>
</table>

Table 4-4 Character Driver

**System Call**

- open()
- close()
- read()
- write()
- ioctl()
- select()

**Executes**

- driver_open()
- driver_close()
- driver_read()
- driver_write()
- driver_ioctl()
- driver_select()

**Purpose**

- Open the device
- Close the device
- Perform character read
- Perform character write
- Perform special command
- Test I/O completion
Step 5: Writing Configuration Routines

All drivers require one or more routines that install the driver into the system and initialize it. HP-UX defines interfaces for the following configuration routines:

- `driver_install()`
- `driver_attach()`
- `driver_dev_init()`
- `driver_if_init()`
- `driver_addr_probe()`
- `driver_dev_probe()`
- `driver_minor_build()`

**Configuration Overview**

The following steps provide an overview of the order of events followed by the kernel in installing and initializing all drivers at boot time (system start or power-on time). The details are given in the sections that follow for each of the configuration routines.

The following list illustrates configuration routines for four types of drivers: device, interface, pseudo, and monolithic (device and interface combined), which are named `devd`, `ifd`, `pseudod`, and `monod`, respectively.

1. For each driver, the kernel calls the driver's install routine.
   
   The `driver_dev_init()`, `driver_if_init()`, and `driver_probe()` routines are optional for all drivers. A `driver_attach()` routine is required for any interface driver that intends to claim an interface card.
   
   The install routines usually perform the following operations:

   a. All install routines call `wsio_install_driver()` or `install_driver()`, which registers the driver with the system.
   b. If a `devd_dev_init()` routine exists, `devd_install()` puts the routine on the applicable chain of init routines for that type of device.
   c. If a `pseudod_dev_init()` routine exists, `pseudod_install()` puts the routine on the applicable chain of init routines for that type of device.
   d. `ifd_install()` puts `ifd_attach()` on the applicable chain(s) of attach routines for supported interfaces (e.g., the `eisa_attach()` and `pci_attach()` routines for a driver which supports both EISA and PCI bus interface cards.) If it has an `ifd_probe()` routine, it registers the routine with the system.
   e. `monod_install()` puts `monod_attach()` on the applicable chain of attach routines. If it has a `monod_dev_init()` routine, it puts the routine on a chain of init routines. If it has a `monod_probe()` routine, it registers the routine with the system.

2. The kernel has installed all the drivers.
   
   The driver’s static or dynamically assigned major number is known at this point.

3. The kernel identifies the hardware on the system. For each interface card, the kernel calls the first entry on an attach chain that contains `ifd_attach()` or `monod_attach()`. Each entry on the chain is responsible for calling the next entry on the chain.
When called, ifd_attach() or monod_attach() lays claim to the hardware it controls. For each card an interface driver claims, the kernel allocates an ISC structure, and the driver can set up its isc->gfsw->init to point to its card-initialization routine, ifd_if_init() or monod_if_init().

4. When all the attach chains have been run, the kernel examines the ISC table and, for each card that contains a valid function call in its isc->gfsw->init structure (as set up above), it calls that function.

5. Lastly, the init chain is run. For each device, pseudo, or monolithic driver that specified one, its devid_dev_init(), pseudod_dev_init(), or monod_dev_init() routine is called once. Each entry on the chain is responsible for calling the next entry on the chain.

Since dynamic and static major numbers are known, a device, pseudo, or monolithic driver can use this init routine to check that it has an appropriate entry in /dev, or to create such an entry.

Writing a driver_install() Routine

The driver_install() function is provided by the driver writer. The name must be in the format shown, with driver replaced by the name of the driver as specified in the system file (defaults to /stand/system) and in the $DRIVER_INSTALL section of a master file in the /usr/conf/master.d directory.

The driver_install() routine has the following tasks:

- It calls a driver installation service to register the driver with the system and fill out system structures and tables.
  
  Use wsio_install_driver() for WSIO installation. (See wsio_install_driver (WSIO3) in the HP-UX Device Driver Reference). Use install_driver() for installation outside of WSIO, typically with certain pseudo drivers (see install_driver (CDIO3)).

- For an interface or monolithic driver, it places the driver_attach() routine at the head of a global attach chain. Later, when the kernel has probed and found an interface card attached to the system, it initiates the appropriate chain, seeking a driver that recognizes the ID information of that card.

  The kernel maintains several global chains of attach routines. The interface driver’s driver_install() routine must save a pointer to the current head of the appropriate chain and place its driver_attach() routine at the head of the chain. For PCI drivers, the head of the global attach chain is pointed to by pci_attach().

- For an interface or monolithic driver, if a driver_probe() routine is defined, driver_install() calls either wsio_register_dev_probe() or wsio_register_addr_probe() to register the routine with the system. See wsio_register_dev_probe (WSIO3) and wsio_register_addr_probe (WSIO3).

- For a device, pseudo, or monolithic driver, if a driver_dev_init() routine is defined, driver_install() saves a pointer to the current head of the chain and places its driver_dev_init() routine at the head of the chain. The chain is processed once all drivers have been configured. The head of the global init chain is pointed to by dev_init().

**driver_install() Return Values**

driver_install() is expected to return the value returned by wsio_install_driver() or install_driver().

Those values are:

- 0 Failure. The driver was not installed.
- 1 Success.

If it fails, the appropriate message appears on the system console and in the system’s error-log file. driver is the name of the driver.
driver_install() Example for Device and WSIO Pseudo Drivers

Installation routine for a driver named skel. The skel_dev_init() device init routine is added to the head of the dev_init() global chain.

```c
int skel_install(void)
{
    extern int (*dev_init)();
    /* head pointer for init chain */

    skel_saved_dev_init = dev_init;
    /* save head pointer */
    dev_init = skel_dev_init;
    /* make my dev_init the head */

    /* register driver with WSIO and return error, if any */
    return (wsio_install_driver(&skel_wsio_info));
}
```

driver_install() Example for Non-WSIO Pseudo Drivers

Installation routine for a pseudo driver named pseu that must be installed outside the WSIO environment. The pseu_dev_init() device init routine is added to the head of the dev_init() global chain.

```c
int pseu_install(void)
{
    extern int (*dev_init)();
    /* head pointer for init chain */

    pseu_saved_dev_init = dev_init;
    /* save head pointer */
    dev_init = pseu_dev_init;
    /* make my dev_init the head */

    /* register driver and return error, if any */
    return (install_driver(&pseu_drv_info, &pseu_drv_ops));
}
```

driver_install() Example for Interface Drivers

Installation routine for a driver named skel. The skel_attach() interface attach routine is added to the head of the pci_attach() global chain. The skel_probe() probe routine is registered with the system.

```c
int skel_install(void)
{
    extern int (*pci_attach)();
    /* head of pci attach chain */

    skel_saved_attach = pci_attach;
    /* save head pointer */
    pci_attach = skel_attach;
    /* make my attach the head */
/* register probe with WSIO */
wsio_register_dev_probe(drv_name, skel_probe,
&skel_drv_info.name);

/* register driver with WSIO and return error, if any */
return (wsio_install_driver(&skel_wsio_info) );
}

driver_install() Example for Monolithic Drivers

Installation routine for a driver named skel.

The skel_attach() interface attach routine is added to the head of the pci_attach() global chain. The skel_dev_init() device init routine is added to the head of the dev_init() global chain. The skel_probe() probe routine is registered with the system.

```c
int (*skel_saved_attach)();
/* to save previous attach head */
int (*skel_saved_dev_init)();
/* to save previous init head */

int skel_install(void)
{
    /* ptr to pci attach chain */
    extern int (*dev_init)();
    /* head pointer for init chain */
    skel_saved_attach = pci_attach;  /* save head pointer */
    pci_attach = skel_attach;  /* make my attach the head */
    skel_saved_dev_init = dev_init; /* save head pointer */
    dev_init = skel_dev_init;  /* make my dev_init the head */

    /* register probe with WSIO */
    wsio_register_dev_probe(drv_name, skel_probe,
&skel_drv_info.name);

    /* register driver with WSIO and return error, if any */
    return (wsio_install_driver(&skel_wsio_info) );
}
```

Writing a driver_attach() Routine

The driver_attach() function is provided by the driver writer. It can have any unique name. The driver registers its attach routine with the WSIO by inserting it in the appropriate attach chains during driver installation. Commonly, driver is replaced by your driver's name.

The kernel searches the I/O backplane for hardware. When it finds a device, it first does preliminary initialization. Then it calls the driver_attach() routine at the head of the corresponding attach chain, e.g., pci_attach(), that was created by the driver_install() routine.

Each driver_attach() routine in the chain looks at the device product ID. If it recognizes the device as its own, it claims the device with the isc_claim() function and optionally puts a pointer to its driver_if_init() routine in isc->gfsw->init, and performs any other appropriate initialization. Then, whether it claims the device or not, it passes the same parameters to the next driver_attach() routine in the chain, using the function pointer it saved in the driver_install() routine.

This algorithm allows multifunction EISA cards to be claimed by more than one interface driver.
If the `isc->ftn_no` field is not -1 in the received ISC for a multifunction EISA card, the `driver_attach()` routine should call `get_new_isc()` to allocate a new ISC structure for the driver's functions and set the `isc->ftn_no` field of the new ISC structure to the function number for its portion of the card. Then it should pass the new ISC on to the next driver in the attach chain. See `get_new_isc` (WSIO3) in *HP-UX 11i v1 Driver Development Reference*.

Since `isc_claim()` sets the `INITIALIZED` flag in `isc->if_info->flags`, you can also test this flag to see if there was a prior claim.

**driver_attach() Return Value**

Each `driver_attach()` routine is expected to return the value returned by the next `driver_attach()` routine in the chain. The end-of-chain function returns a unique completion code.

**driver_attach() Diagnostics**

The `driver_attach()` routine can signal an error as follows:

- The card is faulty. If the developer sets the `INIT_ERROR` flag in `isc->if_info->flags`, the kernel will display the message:

  `init of hardware not successful`

  The `driver_attach()` routine has access to the card registers via the `isc` pointer passed into it. The routine may verify that data any way it sees fit.

- Bad driver. A `driver_attach()` routine returned without calling the next `driver_attach()` routine in the chain. The system will panic and display the message:

  `bad driver in kernel`

**driver_attach() Examples**

```c
static int
skel_attach(int product_id, struct isc_table_type *isc_ptr)
{
    int MY_PRODUCT_ID = 0x1234; /* the four-byte value for this driver */
    /* Check and make sure the product_id belongs to me. */
    /*
     * if (product_id != MY_PRODUCT_ID) /* not mine */
     * return (*skel_saved_attach)(product_id, isc_ptr);
    */
    /* memory allocation, save pointers, other housekeeping.
     * ...
    */
    /* Claim the driver (sets INITIALIZED in
     * isc_ptr->if_info->flags)
     */
    isc_claim(isc_ptr, &skel_wsio_info);
    /* Save my initialization routine (if I have one).
     */
    isc_ptr->gfsw->init = skel_if_init;
    /* Chain to the next driver's attach routine.
     */
    return (*skel_saved_attach)(product_id, isc_ptr);
}
```
Writing a Driver

Step 5: Writing Configuration Routines

Writing a driver_dev_init() Routine

The `driver_dev_init()` routine is provided by the driver writer. It can have any unique name. You pass the name to WSIO Services by linking it into an `init` chain from your `driver_install()` routine.

```c
int (*sd_saved_dev_init)();

int sdisk_install(void)
{
    sdisk_link();

    /* register driver with WSIO and return any error */
    return wsio_install_driver(&sdisk_wsio_info);
}

static void
sdisk_link(void)
{
    sd_saved_dev_init = dev_init;
    dev_init = sdisk_init;
}

static int
sdisk_init(void)
{
    /*
    * code to initialize (usually) a
    * driver defined switch table
    */
    (*sd_saved_dev_init)();
}
```

Writing a driver_if_init() Routine

The `driver_if_init()` routine is provided by the driver writer. It can have any unique name. You pass the name to WSIO Services by specifying it in the `gfsw` structure during `driver_attach()`.

```c
static int
mydriver_attach(uint32_t card_id, struct isc_table_type *isc)
{
    if (card_id == my_card_id) {

        /* if mydriver has an init routine */
        isc->gfsw->init = mydriver_init;
        ...
    }
}

static int
mydriver_init(struct isc_table_type *isc)
{
    wsio_intr_object_t my_intr_obj);
    int status;

    /* set up the cards line based interrupt */
    status = wsio_intr_alloc(isc, mydriver_isr,
                               (unitptr_t)isc, 0, &my_intr_obj);
    if(status != WSIO_OK)
        return(ERROR);

    status = wsio_intr_set_irq_line(isc, my_intr_obj,}
Writing Driver Probe Routines

Drivers can register probe functions that are used by the WSIO to scan for devices underneath an interface card. When a driver probe function is called it is passed a probe type as a parameter. The probe type can be one of the following three values:

- **PROBE_FIRST**: Find the device at the first address underneath the interface card
- **PROBE_NEXT**: Find the next device after the previous one found.
- **PROBE_ADDRESS**: Look for a device at the specific hardware address.

There are two WSIO services that can be used to register a driver’s probe functions.

Most drivers will only use the service `wsio_register_dev_probe()` to register a probe function based on either the class of devices the driver controls or the driver’s name as specified in the name field of the dev_init structure. The second service `wsio_register_addr_probe()` is used to associate an additional probe function based on the name of the driver as specified in the driver’s `drv_info_t` structure.

Normally a driver will only register a single probe function using the first service. In some cases a stack of drivers that cooperate can register different probe functions using the two different WSIO services. The WSIO will then use these probe functions together. How this is done is covered later.

The following describes each of the WSIO services in more detail.

### wsio_register_dev_probe()

The calling semantics for `wsio_register_dev_probe()` are:

```c
int wsio_register_dev_probe(u_int type, int (*func)(), char *str);
```

Where type can be either `IF_CLASS` or `DRV_NAME`, `func` is a pointer to the driver probe function and `str` is an ascii string.

The service `wsio_register_dev_probe()` is used to register a probe function for a driver based on either the drivers class or name. If the driver specifies `class`, it must pass in an ASCII string that matches the `drv_path` field of its `wsio_drv_data_t` structure and a type of `IF_CLASS`. If the driver specifies name, it must pass in an ASCII string that matches the `name` field of its `drv_info_t` structure along with the type `DRV_NAME`.

A driver registers its probe function by calling `wsio_register_dev_probe()` in its `driver_install` routine. The following is an example of this. In the example the driver specifies a class type probe function.

```c
int sctl_install(void)
{
    /*
Writing a Driver

Step 5: Writing Configuration Routines

```c
* register the scsi probe function
* with the WSIO CDIO.
*/
wso_register_dev_probe(IF_CLASS, scsi_probe, "scsi");

/*
* register driver with WSIO and return any error.
*/
return wsio_install_driver_(&sctl_wsio_info));
}
```

The WSIO will save a pointer to the probe function and later retrieve it when probing for devices underneath an interface card. When the WSIO associates a probe function with an interface card it will first try and match a probe function based on the driver’s name. If one is not found it will look for a probe function based on the driver’s class type.

A probe function registered by `wsio_register_dev_probe` must have the following calling prototype:

```c
int drv_probe(void *node, drv_info_t *drv_info,
               void *probe_id, hw_path_t *hw_path,
               struct isc_table_type *isc, int probe_type,
               char *name, char *desc);
```

For addition information, refer to Table 4-5, “Driver Probe Parameters.”

### Table 4-5 Driver Probe Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>IN</td>
<td>A pointer to the node</td>
</tr>
<tr>
<td>drv_info</td>
<td>IN</td>
<td>A pointer to the <code>drv_info_t</code> structure</td>
</tr>
<tr>
<td>probe_id</td>
<td>OUT</td>
<td>A unique identifier for the device found</td>
</tr>
<tr>
<td>hw_path</td>
<td>IN</td>
<td>The hardware path of the last device found</td>
</tr>
<tr>
<td>isc</td>
<td>IN</td>
<td>A pointer to the <code>isc_table_type</code> structure</td>
</tr>
<tr>
<td>probe_type</td>
<td>IN</td>
<td>The type of hardware probe. There are three types previously described.</td>
</tr>
<tr>
<td>name</td>
<td>OUT</td>
<td>A pointer to a string initialized with the device’s name such as &quot;scsi_disk&quot;.</td>
</tr>
<tr>
<td>desc</td>
<td>OUT</td>
<td>A pointer to a string with the device description. This is driver dependent</td>
</tr>
</tbody>
</table>

`wsio_register_addr_probe()`

Drivers can register an additional address probe function by calling the WSIO service `wsio_register_addr_probe()`. This service is used to associate an additional probe function based on the driver’s name. The name must match the “name” field of the drivers “`drv_info_t`” structure.

A driver registers its address probe function by calling `wsio_register_addr_probe()` in its `driver_install` routine, as shown in the following example.

```c
int c720_install(void)
{
    int ret;
```
/**
 * Register the driver with WSIO.
 * If it succeeds then lets add our attach function
 * to the PCI attach list and register the
 * parallel_scsi_probe function for the address probe.
 */
if (ret = wsio_install_driver(&c720_wsio_info)) {
    c720_saved_pci_attach = pci_attach;
    pci_attach = (int (*)(*)())c720_pci_attach;
    wsio_register_addr_probe(parallel_scsi_probe,
        "scsi_c720");
}
return ret;
}

Driver address probe functions registered by \textit{wsio_register_addr_probe()} must have the following prototype:

\begin{verbatim}
int drv_addr_probe(void *node, int (*dev_probe)(),
        drv_info_t *drv_info, void *probe_id,
        hw_path_t *hw_path,
        struct isc_table_type *isc,
        int probe_type, char *name,
        char *desc);
\end{verbatim}

For additional information refer to Table 4-6, “Address Probe Parameters.”

\begin{table}[h]
\centering
\begin{tabular}{|l|l|p{0.7\textwidth}|}
\hline
\textbf{node} & \textbf{IN} & A pointer to the io_tree_node \\
\textbf{dev_probe} & \textbf{IN} & A pointer to a class probe function if one exists else NULL \\
\textbf{drv_info} & \textbf{IN} & A pointer to the drv_info_t structure \\
\textbf{probed} & \textbf{OUT} & A unique identifier for the device found \\
\textbf{hw_path} & \textbf{IN} & The hardware path of the last device found \\
\textbf{hw_path} & \textbf{OUT} & The hardware path of the next device found \\
\textbf{isc} & \textbf{IN} & A pointer to the isc_table_type structure \\
\textbf{probe_type} & \textbf{IN} & The type of hardware probe. There are three types previously defined. \\
\textbf{name} & \textbf{OUT} & A pointer to a string initialized with the device's name such as "scsi_disk". This information is used to match the device to a driver based on the information in the drv_path. \\
\textbf{desc} & \textbf{OUT} & A pointer to a string with the device description. This is driver dependent \\
\hline
\end{tabular}
\caption{Address Probe Parameters}
\label{table:address-probe-parameters}
\end{table}

\textbf{NOTE} The only difference between the calling interface of a probe function registered by \textit{wsio_register_dev_probe} and one registered by \textit{wsio_register_addr_probe} is that the latter has an additional parameter, (*dev_probe)(). This is how the two types of probe function can be used together by some driver stacks.
Normally a driver would only register a single probe function by calling `wsio_register_dev_probe` to register either a probe function based on the driver’s name or class. Certainly this will be the case for monolithic drivers. However, there are some driver stacks which are very modular such that a device driver can be paired with several different interface drivers where each interface driver supports a different range of addresses. This is when the two WSIO services can be used to register separate probe functions that will work together.

For example in the above cases the driver install routine of the scsi interface driver `scsi_c720` registered an “address” probe function by calling `wsio_register_addr_probe()` where as the `scsi_ctl.c` driver registered its “class” probe function called `scsi_probe()` by calling `wsio_register_dev_probe`. When probing for scsi devices underneath a `scsi_c720` interface card the WSIO will use both probe functions by calling the address probe function and passing in a pointer to the class probe function. The driver address probe function can set up the address for the next device to be probed and then call the class probe function to talk to the devices underneath.

When a driver class probe function is used with an address probe function and hence is not called directly by the WSIO CDIO probe code, it does not have to adhere to the WSIO calling semantics that were described earlier for class probe functions. The driver stack can define its own parameter list for the class probe function. In this case though it is important that the class probe function is never used as a standalone probe function that is called directly by the WSIO CDIO probe code.

In the following two examples the `scsi_c720.c` interface driver registers an address probe function called `parallel_scsi_probe` and the `scsi_ctl.c` device driver registers a class probe function called `scsi_probe()`. In these examples the two probe functions cooperate and hence the interface to the class probe function `scsi_probe()` does not adhere to the WSIO CDIO specification.

**Example of a driver_addr_probe() Routine**

```c
#include <sys/wsio.h>

int c720_install(void)
{
    ...
    wsio_register_addr_probe(parallel_scsi_probe, "c720");
    return 1;
}

/*
 * parallel_scsi_probe is the system's address probe
 * routine for SCSI interfaces. It is registered by
 * scsi_c720. It first determines the next address to be
 * probed, and then calls the scsi_ctl registered probe
 * function, i.e., probe_func = scsi_probe(),
 * which actually tries to open and identify any
 * underlying hardware.
 */

static drv_info_t c720_info = {
    "c720", "ext_bus", DRV_SAVE_CONF | DRV_SCAN
    | DRV_MP_SAFE,NULL, NULL, NULL, NULL, NULL,
};

static wsio_drv_data_t c720_data = {
    "scsi", T_INTERFACE, DRV_CONVERGED, NULL, NULL,
};

#define NEXT_TARGET 1
#define NEXT_LUN 2

int parallel_scsi_probe(void *this_node, int (*probe_func)(),
    drv_info_t *drv_info, void *probe_id,
    hw_path_t *hw_path,
    ...}
struct isc_table_type *isc,
    int probe_type,
    char *name, char *desc)
{
    int looking_for, found;
    char dev_class[8];
    dev_t dev;
    int instance;
    int node_type, parent_type;
    struct io_tree_node *parent_node;
    int lun, target;
    struct wsio_probe_dev_info probe_dev;

    /*
     * The hardware path should only contain two
     * elements at the most. A target and an LUN.
     * If first_index == last_index, we have only a
     * target address. If first_index + 1 == last_index
     * we have both a target and lun, otherwise we do
     * not know how to interpret the hardware path so
     * just return.
     */
    if((hw_path->first_index + 1) < hw_path->last_index)
        return(0);
    if(probe_func == NULL)
        return(0);
    probe_dev.instance = wsio_isc_to_instance(isc, hw_path);
    probe_dev.opt_1 = 0;
    probe_dev.opt_2 = 0;
    probe_dev.opt_3 = 0;
    if(probe_type == PROBE_NEXT) {
        /*
         * If only a target was passed in then the last thing
         * found was a target so the next thing to look for
         * is the first lun of that target, so get the target
         * address and set lun to -1 so that it gets
         * automatically incremented to 0 in the switch
         * statement below. Else get the previous values for
         * target and lun from the path.
         */
        if(hw_path->first_index == hw_path->last_index) {
            target = (int)hw_path->addr[hw_path->last_index];
            hw_path->last_index++;
            lun = -1;
        } else {
            lun = (int)hw_path->addr[hw_path->last_index];
            target = (int)hw_path->addr[hw_path->last_index-1];
        }
        looking_for = NEXT_LUN;
    }
    found = FALSE;
    do {
        switch(probe_type) {
            case PROBE_FIRST:
                target = lun = 0;
                hw_path->last_index++;
                looking_for = NEXT_TARGET;
                probe_type = PROBE_NEXT;
                break;
            case PROBE_NEXT:
                /*
                 * If we didn’t find a device (target) or the last
                 * lun used was invalid (out of range) then go
                 * to the next target and scan starting
                 * with lun 0.
                 */
                if((found == NO_DEV) || (found == INVAL_LUN)) {
                    target++;
                } else {
                    lun = Lun++;
lun = 0;
hw_path->last_index = hw_path->first_index;
looking_for = NEXT_TARGET;
} else {
    lun++;
    looking_for = NEXT_LUN;
} break;
case PROBE_ADDRESS:
    if(hw_path->first_index == hw_path->last_index) {
        target = (int)hw_path->addr[hw_path->last_ix];
        lun = 0;
        looking_for = NEXT_TARGET;
    } else {
        lun = (int)hw_path->addr[hw_path->last_index];
        target = (int)hw_path->addr[hw_path->last_index-1];
        looking_for = NEXT_LUN;
    }
    break;
default:
    return 0;
}
probe_dev.target = target;
probe_dev.opt_1 = lun;
found = probe_func(isc, NULL, &probe_dev, probe_type,
                    probe_id, dev_class, desc);

/*
  * We need to keep looking if we are not looking for a
  * particular address (probe_type == PROBE_ADDRESS) and
  * while we haven't run out of valid targets and either
  * we are looking for the next target and we haven't
  * found a device or we are looking for the next lun and
  * we have found a device but not a valid lun for it.
  */
} while ((probe_type != PROBE_ADDRESS) &&
    ((found != INVAL_TGT) &&
    ( (looking_for == NEXT_TARGET) ||
    (found == NO_DEV)) ||
    (looking_for == NEXT_LUN) &&
    (found != VALID_LUN)));

if(((looking_for == NEXT_TARGET) && (found >= VALID_TGT)) ||
    (looking_for == NEXT_LUN) && (found == VALID_LUN)) {
    if((looking_for == NEXT_TARGET) {
        strcpy(&name[0],"target");
        hw_path->addr[hw_path->last_index] = target;
        desc[0] = '\0';
        /* don't return the description for targets*/
    } else {
        strcpy(&name[0],"scsi_");
        strcpy(&name[5],dev_class);
        hw_path->addr[hw_path->last_index] = lun;
        hw_path->addr[hw_path->first_index] = target;
    }
    return 1;
} else {
    return 0;
}

} /* parallel_scsi_probe */
Example of the scsi_probe()

/*
 * scsi_probe is the system’s class (device) probe
 * routine for SCSI interfaces.
 * It is registered by scsi_ctl.
 * It is called by parallel_scsi_probe.
 */

static drv_info_t sctl_info = {
    "sctl", "ctl", DRV_CHAR, -1, 203, NULL, NULL, NULL,
};

static wsio_drv_data_t sctl_data = {
    "scsi_ctl", T_DEVICE, DRV_CONVERGED, NULL, NULL,
};

int sctl_install(void) /* scsi_ctl driver install */
{
    wsio_register_dev_probe (if_class, scsi_probe, "scsi");
    return( wsio_install_driver(&sctl_wsio_info));
}

scsi_probe(
    struct isc_table_type *isc,
    hw_path_t *hw_path,
    struct wsio_probe_dev_info *probe_dev,
    int probe_type,
    void *dev_id,
    char *dev_class,
    char *desc
)
{
    int found;
    int major_num;
    int dev_type;
    int max_target, max_lun;
    union inquiry_data iqr_data;
    struct sctl_io *sctl_io;
    dev_t dev;

    /*
     * Driver has used a field in the ISC telling the maximum
     * width of the bus it can support. The maximum number of
     * targets is one less (since the IDs start at 0).
     * A sanity check is done to ensure we’re dealing with
     * positive number.
     */

    max_target = isc->bus_max_width - 1;
    if (max_target <= 0)
        return INVAL_TGT;
    max_lun = 7;
    if ((probe_dev->opt_1 < 0) || (probe_dev->opt_1 > max_lun))
        return INVAL_LUN;
    if ((probe_dev->target < 0) || (probe_dev->target > max_target))
        return INVAL_TGT;

    /*
     * check if this target is the address of the scsi board.
     */
    if (probe_dev->target == isc->my_address)
        return NO_DEV;

    /*
     * Get the major number of the driver
     * and build a dev_t structure.
     */
    major_num = sctl_wsio_info.drv_info->c_major;
    dev = ((major_num << 24) & 0xff000000);
dev = ((probe_dev->instance << 16) & 0x00ff0000);
dev = ((probe_dev->target << 12) & 0x0000f000);
dev = ((probe_dev->opt_1 << 8) & 0x00000f00);
found = NO_DEV;
if (((sctl_open(dev)) == 0) {
    if (((sctl_ioctl(dev, SIOC_INQUIRY, &iqr_data, 0)) == 0) {
        dev_type = iqr_data.inq2.dev_type;
        if (dev_type != NO_SCSI_DEV) {
            found = VALID_TGT;
            /* Get first four bytes of the inquiry data
               * for dev_id and the vendor and product id for
               * desc. and set dev class according to the
               * dev_type.
               */
            *(int *)dev_id = *((int *)&iqr_data);
            strncpy(desc, iqr_data.inq2.vendor_id, 8);
            strncpy(&desc[8], iqr_data.inq2.product_id, 16);
            desc[24] = '\0';
            switch(dev_type) {
                case SCSI_PROCESSOR:
                    strcpy(dev_class, "processor");
                    break;
                case SCSI_DIRECT_ACCESS:
                case SCSI_CDROM:
                case SCSI_MO:
                case SCSI_WORM:
                    strcpy(dev_class, "disk");
                    break;
                case SCSI_SEQUENTIAL_ACCESS:
                    strcpy(dev_class, "tape");
                    break;
                case SCSI_AUTOCHANGER:
                    strcpy(dev_class, "changer");
                    break;
                case SCSI_PRINTER:
                    strcpy(dev_class, "printer");
                    break;
                case SCSI_SCANNER:
                    strcpy(dev_class, "scanner");
                    break;
                case SCSI_COMMUNICATIONS:
                    strcpy(dev_class, "comm");
                    break;
            } /* switch */
            /* We know we found a valid device now lets
               * make sure that it is also a valid LUIN.
               */
            if (iqr_data.inq2.periph_qualifier == 0)
                found = VALID_LUN;
        } /* dev_type != NO_SCSI_DEV */
    } /* sctl_ioctl */
    sctl_close(dev);
} /* sctl_open */

return found;
} /* scsi_probe */
Writing a driver_minor_build() routine

The `driver_minor_build()` routine is used when the driver has a special method for building minor numbers. The following is an example:

```c
static int
pciide_minor_build(struct isc_table_type * isc,
                    hw_path_t *path, char *option)
{
    int card_instance;
    int minor = 0;
    int dev_id,func;

    /*
     * minor number has the format
     * 23  16 15  8  7  0
     * <if card_instance><dev_id><func>
     */
    card_instance = wsio_isc_to_instance(isc,null);
    minor |= card_instance << 16;

    /*
     * dev_id & func are the last two elements of the
     * hardware path of the device.
     */
    dev_id = path->addr[path->last_index -1];
    func = path->addr[path->last index];

    minor |= ((dev_id << 8) & 0xFF00;
    minor |= (func & 0xFF);
    return minor;
}
```
Step 6: Writing Entry Point Routines

Most drivers have defined entry point routines. An entry point routine is a driver routine that is called through a (non NULL) field of the drv_ops_t structure. Interface drivers do not have entry point routines by this definition. Their routines such as driver_isr() are in the “Step 7: Writing Other Driver Routines” section. Refer to the discussion in “Step 4: Identifying Routines for Your Driver” to determine applicable routines for your driver.

Writing a driver_open() Routine

The driver_open() routine prepares a device for I/O. driver_open() is provided by the driver writer. It can have any unique name. The developer passes the name to WSIO services by specifying it in the d_open field of the drv_ops structure. Commonly, driver is replaced by the driver's name.

A user process makes the open() system call for a device file (usually in /dev). Then the kernel file system open() routines check permissions and do other housekeeping tasks eventually calling the corresponding driver_open() routine, dispatching control to the driver routine defined in the drv_ops_t structure.

In general, the driver_open() routine does the following:

- Opens the device.
- Returns an error if the device is an exclusive-open device and this is not the first open.
- Allocates a set of data structures, if the device is a multiple-open device.
- Finds the device (determines the hardware location, based on the dev_t value passed to it).
- Sets a flag indicating the device is open.
- Initializes data structures as needed.
- Initializes hardware (puts the device in a known state).
- Clears the open flag and returns an error if an error occurs during initialization.
- Returns zero if the open was successful.

The device driver must implement the type of open() required by the device. There are three types:

1. Exclusive Open

   Opening a device exclusively allows only one process at a time to access the device. Magnetic tapes and printers are such devices.

   To enforce this exclusiveness, the driver maintains a flag that indicates whether the device is currently open. If only one process at a time should open the device, the driver_open() routine should return an error whenever it executes and finds the device is already opened.

2. Shared Open

   Devices that allow more than one process at a time to access them can be opened shared. Terminals are typically shared-open devices so users can communicate with each other using the write() command, for example; see write (1) and write (2).

   All processes that have a shared-open device open at the same time share a common set of global data structures. If one process modifies a value in one of these data structures, all processes that have opened the device can see the modified value.
3. Multiple Open

Devices that allow more than one process at a time to access them can also be opened by a multiple-open operation. Disks are multiple-open devices.

Each process that opens a multiple-open device has its own copy of the device’s global data structures. This allows each process to modify the data structures’ values independently.

Decide which type of open() routine is needed, and add appropriate code to the routine. See the skeleton routines. (Also see the “Sample driver_open() Routine for a Device Driver” section.)

```
int driver_open(dev_t dev, int flag);
```

**dev**

The dev_t device number of the file to be opened. See “Major and Minor Numbers” in Chapter 2, “HP-UX I/O Subsystem Features.” The driver_open() routine can extract the major and minor numbers from the device number. (See major (KER2) and minor (KER2) in the HP-UX 11i v1 Driver Development Reference.)

**flag**

A value corresponding to the oflag parameter of the open() system call. The kernel executes the oflag functions (described in fcntl(5) and open(2) in the HP-UX Reference) before it calls the driver. The driver, therefore, can usually ignore these flags.

Nevertheless, the kernel translates the O_xxxx values into corresponding Fxxxx values, which it passes to the driver_open() routine. The flags of possible interest to the driver include: FREAD, FWRITE, FNDELAY, and FEXCL.

The driver_open() routine for a magnetic tape, for example, checks the value of FWRITE. If the tape is being opened for writing and the tape is write protected, the driver_open() routine returns an error to the open() system call.

If the kernel calls a terminal driver’s driver_open() routine, for example, and FNDELAY is set, the routine does not wait for the hardware to respond before returning control to the open() system call. (See fcntl (5), open (2), and termio (7) in the HP-UX Reference for more information.)

---

**NOTE**

The kernel does not check that the minor number coded in the dev parameter is valid, because minor numbers are defined by the driver. For example, if mknod is used to create a dev with minor number 0x0, the dev structure that the kernel passes to the driver_open() routine contains minor number 0x0. If this is not a valid minor number for the device, the driver_open() routine should discover this error.

The driver_open() routine should return either a zero or an errno value to the open() system call, respectively indicating success or failure.

If the driver_open() routine is successful, the kernel’s open() call returns a file descriptor to the user. If it is unsuccessful, the kernel returns -1 to the user and sets errno to the value returned by the driver_open() routine. The user’s process can check the returned value and errno to determine whether an error occurred. See the <errno.h> header file for possible values for errno.

The driver_open() routine should return an error under these conditions. See open (2) in the HP-UX Reference for the expected error names.

- The device is off line.
- The device does not exist.
- The device was never configured into the system.
- The initialization of the device failed.
The device is an exclusive-open device, and it is already open.

Sample driver_open() Routine for a Device Driver

This example is for a character driver that controls a Centronics interface. This is an exclusive open device:

```c
static int CharDrv_open(dev_t dev, int flag)
{
    struct isc_table_type *isc;
    CentIfSwitch_t *ifsw;
    u_int minor_number;

    wsio_get_isc(dev, &isc, &CharDrv_wsio_info);

    /* our device? */
    if (isc == NULL || isc->if_id != CENT_SV_ID)
        return ENXIO;

    if (DevIsOpen) /* is the flag nonzero? */
        return EACCES;

    /*
     * Lets initialize the device (call the interface
     * driver). If it fails return an error.
     * Note that a return value of 0 indicates success.
     */
    ifsw = (CentIfSwitch_t *)isc->ifsw;
    minor_number = minor(dev);
    if (((*ifsw->dev_init)(isc, minor_number) != 0)
        return EIO;

    /* reserve the device */
    DevIsOpen++;
    return 0;
}
```

Writing a driver_close() Routine

driver_close() is provided by the driver writer. It can have any unique name. Pass the name to WSIO services by specifying it in the d_close field of the drv_ops structure. Commonly, driver is replaced by your driver's name.

What a driver_close() routine does depends on how the device is opened. The kernel invokes the driver_close() routine when a process uses the close() system call to close a device file, but not every time for all devices.

While a user process specifies a file descriptor in the close() system call, the kernel invokes the driver_close() routine with the following declaration:

```c
int driver_close(dev_t dev, int flag);
```

- **dev** The device number of the file to be closed. The driver_close() routine can extract the major and minor numbers from the device number (see major (WSIO3) and minor (WSIO3) in the HP-UX 11i v1 Driver Development Reference).

- **flag** A value corresponding to the flag field in the driver_open() call. (See the parameter description in “Writing a driver_open() Routine” for values that can appear in the flag parameter.)
The kernel always returns success (0) to the caller of `close()`, ignoring the return value it gets from the driver. Therefore, the `driver_close()` routine need not return a valid value. However, to avoid problems (as with strict compiler return value checking), the `driver_close()` routine could return some integer value (such as 0).

**Writing a driver_close() Routine for an Exclusive-Open Device**

The `driver_close()` routine for an exclusive-open is simple. Because only one process at a time can open the device, the kernel invokes the `driver_close()` routine each time a process calls the `close()` system call. The `driver_close()` routine for such a device does the following:

1. Completes all I/O in progress.
2. Releases data structures.
3. Clears the driver's open flag that was set by `driver_open()`, indicating the device is closed.
4. Takes other, device-specific actions (for exclusive-open devices).

Examples of device-specific actions a `driver_close()` routine takes are unlocking the door of a floppy disk and rewinding the tape in a tape drive.

When such a driver releases data structures, those structures are the ones allocated by the driver when the device was opened.

**Writing a driver_close() Routine for a Shared-Open or Multiple-Open Device**

Because more than one process at a time can open a shared-open or multiple-open device, a `driver_close()` routine for such a device is likely to be more complex than one for an exclusive-open device.

The `driver_close()` routine for a shared-open or multiple-open device does the following:

1. Completes all I/O in progress.
2. Releases data structures.
3. Clears the open flag, indicating the device is closed.
4. Takes other, device-specific actions.

When a user calls `close()` for a device, the following occurs:

1. The `close()` system call dispatches control to the driver's `driver_close()` routine.
2. The `driver_close()` routine is called only on the last close of the device file, unless the `C_ALLCLOSES` flag was specified in the `drv_ops_t` structure for the device. The file system maintains a count of the number of opens on any file.

   Calling `driver_close()` only on the last close of the device prevents a process from closing a device while another process is accessing it. If the device has more than one device number, `dev`, the kernel calls `driver_close()` once for each number.

3. A file must be unmounted before a block-device file is closed. If the file is mounted, the `close()` system call returns control to the calling process without calling the `driver_close()` routine.
4. The device must remain open until all active inodes for the device are closed. If more than one active inode can exist for the same device, (if two device files with the same major number can access the same device), the driver should set the `C_ALLCLOSES` flag and maintain its own count of how many processes have the device open.
NOTE  Setting this flag does not guarantee that the kernel will call the driver_close() routine for every close() call for the device, but only when the link count in the inode is 0 or 1. It will not call the driver_close() routine, for instance, when a forked child process calls close() for an open-file descriptor it has inherited.

Using the C_ALLCLOSES Flag

The C_ALLCLOSES flag allows your driver to track directly how many times a device has been opened and closed.

The kernel maintains a count of opens and closes that have been issued for each device. The driver_open() routine is called for every open() system call.

Without the C_ALLCLOSES flag, the kernel only calls the driver_close() routine for a close() system call when the count drops to zero (the last close).

With the C_ALLCLOSES flag, the kernel calls the driver_close() routine on every close() system call. This allows the developer to keep track of complex device structures, such as a device with two device files having the same major number.

Be aware that, when file descriptors created by fork() or dup() are closed, your driver_close() routine is not called, regardless of the C_ALLCLOSES flag.

Sample driver_close() Routine

This example is for a character driver that controls a Centronics interface. See “Sample driver_open() Routine for a Device Driver”, for the corresponding driver_open() routine.

```c
static int CharDrv_close(dev_t dev, int flag)
{
    struct isc_table_type *isc;
    CentIfSwitch_t *ifsw;

    /* be sure device is open */
    if (DevIsOpen <= 0) {
        return EACCES;
    }

    /* invoke interface driver shutdown routine */
    wsio_get_isc(dev, &isc, &CharDrv_wsio_info);

    ifsw = (CentIfSwitch_t *)isc->ifsw;
    if (((ifsw->dev_end)( isc) != 0) {
        return EIO;
    }

    /* release the device */
    DevIsOpen--;
    return 0;
}
```
Writing a driver_read() or driver_write() Routine

The driver_read() and driver_write() routines control device I/O. The names reflect the kernel view of an I/O transaction. A read transaction moves data from the device to processor memory, while a write transaction moves data from processor memory to a device. driver_read() and driver_write() are provided by the driver writer. They can have any unique name. Pass the names to WSIO services by specifying them in the d_read and d_write fields of the drv_ops structure. Commonly, driver is replaced by the driver's name.

When a user process issues a read(), readv(), write(), or writev() system call for a character device, the kernel puts information about the request in the uio and iovec structures and dispatches control to the driver_read() or driver_write() routine for that device, passing the uio structure to the driver as a parameter.

The kernel does not buffer data between the user process and a character device. The driver must buffer the data.

Character devices typically need different processing for read requests and write requests, so they can have separate routines for reading and writing operations. If character devices share a great deal of code, common code can be combined into a single "strategy" routine that both driver_read() and driver_write() call.

If a device can be accessed as both a block device and a character device, the driver_strategy() routine can often be shared, reducing redundant code. In this case the easiest method is for the driver_read() and driver_write() routines to perform whatever processing is unique to the request's mode of access, and then call the kernel routine physio(), which calls the driver_strategy() routine.

The driver_read() and driver_write() routines can also process requests using uiomove(). The following sections describe the driver_read() and driver_write() routines, as well as how to use physio() and uiomove(). See physio(KER2) and uiomove(KER2) in the HP-UX 11i v1 Driver Development Reference.

Implement a driver_read() routine in two ways:

- Call physio() with the appropriate parameters, allowing the driver_strategy() routine to complete the request. If physio() is used, write a driver_strategy() routine.
- The driver strategy routine is passed as a parameter to physio().
- Use uiomove() to buffer the data and then to complete the request. If you use uiomove(), the driver_read() routine does the following:
  1. Initializes data structures.
  2. Sets a flag indicating that I/O is in progress.
  3. Requests an I/O operation from the device.
  4. Waits or sleeps while the device completes the I/O operation.
  5. Calls uiomove() to transfer the data from the kernel's buffer to the user's buffer.
  6. Returns a value to the read() or readv() call.

You can implement the driver_write() routine in two ways:

- Use physio() and a driver_strategy() routine.
- Use uiomove(), in which case the driver_write() routine does the following:
  1. Initializes data structures.
  2. Calls uiomove() to copy the data into kernel space.
  3. Sets a flag indicating that I/O is in progress.
4. Requests that the device start the I/O operation.
5. Waits or sleeps while the device completes the I/O operation.
6. Returns a value.

The kernel places values in the `uio` structure, and then passes the structure and the device number to the `driver_read()` routine.

See the “Using `physio()`” and “Using `uiomove()`” sections for more information about using these routines in implementing the `driver_read()` and `driver_write()` routines.

```c
int driver_read(dev_t dev, struct uio * uiop);
int driver_write(dev_t dev, struct uio * uiop);
```

- **dev**  
  The device number of the associated device file. The routine can extract the major and minor numbers from the device number. The `driver_open()` routine should have verified that the minor number is valid before accessing the kernel’s data structures. See the Parameter section in “Writing a `driver_open()` Routine”.

- **uiop**  
  A pointer to a `uio` structure. See “System-Defined Header Files”. The `uio` structure contains information about the data being read or written.

`driver_read()` and `driver_write()` are executed to return the following values:

- 0  
  Successful completion.

- <>0  
  Error. The value is expected to be an `errno` value.

**Using `physio()`**

The character driver can call `physio()` and use a common `driver_strategy()` routine for both `driver_read()` and `driver_write()` routines.

The kernel passes the device number and `uio` structure as parameters to the `driver_read()` and `driver_write()` routines. The `driver_read()` and `driver_write()` routines pass them to `physio()`, along with the parameters shown in this declaration:

```c
int physio(int(*strat)(), struct buf *bp, dev_t dev, int flag,  
void(*mincnt)(), struct uio *uiop);
```

- **strat**  
  The `driver_strategy()` routine, which sets up an I/O request.

- **bp**  
  A pointer to a `buf` structure. If the pointer is `NULL`, `physio()` allocates a `buf` structure from the file system’s buffer cache.

  Alternatively, the driver can allocate its own `buf` structure and pass a pointer to it.

- **dev**  
  The device number.

- **flag**  
  A read-write flag. Set the value to `B_READ` for a read request; set it to `B_WRITE` for a write request.

- **mincnt**  
  The routine that divides a data transfer that is larger than the system’s maximum size for a single request (determined by the `block_size` system parameter) into several requests to a driver, each request no larger than the system’s maximum size. The developer can use the kernel’s `minphys()` routine, which most drivers use, or write their own. The `physio()` routine sends the requests to the routine specified in `strat`.

- **uiop**  
  Pointer to the `uio` structure.

The `physio()` routine handles the I/O transfer for the `driver_read()` and `driver_write()` routines as described in `physio` (KER2) in the *HP-UX 11i v1 Driver Development Reference*. 
The `physio()` routine locks the user's data area so it cannot be swapped out during the transfer. Then for each `mincnt` size chunk to be transferred, `physio()` calls the `driver_strategy()` routine with a `buf` structure `*bp`.

The `driver_strategy()` routine initiates I/O on `*bp` and returns control to the `physio()` routine. After `driver_strategy()` returns control to it, `physio()` sleeps on the buffer header. It awakens when the driver sets the `b_done` flag in the buffer's header and calls `biodone(bp)`. The driver does this when the transfer is complete. This means that `physio()` provides synchronous reads and writes.

`physio()` continues to call `driver_strategy()` for each `mincnt` size chunk, updating the `uio` structure each time until the transfer is done or an error is returned. `physio()` then unlocks the user's data area, saves the residual count (from `bp->b_resid`) in the `uio` structure, interprets errors returned in `b_error`, if any, and returns to the `driver_read()` or `driver_write()` routine that called it.

**Sample driver_read Routine Using physio()**

```c
/* SKEL_READ */
static int
skel_read(dev_t dev, struct uio * uiop)
{
    return physio(skel_strategy, NULL, dev, B_READ,
                  minphys, uiop);
}
```

**Using uiomove()**

The `uiomove()` routine moves data from one address space to another. In general, use `uiomove()` if your driver transfers small amounts of data for each request. A driver can call this routine to copy data between user space and kernel space. If a `driver_write()` routine calls `uiomove()` to move data from user space to kernel space, it does not have to map the user's data area into kernel space.

The `uiomove()` call has the following declaration:

```c
int uiomove(caddr_t cp, int n, int flag, struct uio * uiop);
```

- `cp` (pointer to kernel space to hold data)
- `n` (count of bytes to transfer)
- `flag` (transfer direction: `UIO_READ`, `UIO_WRITE`)
- `uiop` (pointer to the `uio` structure for the transfer)

To write a routine using `uiomove()`, observe the following points:

- Set `flag` to `UIO_READ` to have `uiomove()` copy `n` bytes, starting at address `cp`, into user space starting at address `uio`.
- Set `flag` to `UIO_WRITE` to have `uiomove()` copy `n` bytes from `uio` into kernel space starting at `cp`.
- If the developer wants a driver to buffer data between the device and the user's buffer, use `geteblk()` to get an empty buffer and a buffer header from the kernel's buffer cache.
- `geteblk()` allocates a buffer from the file system's buffer cache and returns it to the driver; the file system temporarily loses access to this buffer. The `buf` structure and its associated buffer belong exclusively to the routine that called it.
- When the driver completes the request, it should release the `buf` structure and the buffer it obtained from `geteblk()` by using the kernel's `brelse()` routine.
When a device driver gets a buffer using `geteblk()`, it is borrowing a buffer that would otherwise be used by the file system to cache data. This means a device driver that allocates buffers indiscriminately using `geteblk()` can affect the system's performance.

See `uiomove (KER2)` in the *HP-UX 11i v1 Driver Development Reference* for more details and example code.

**Writing a driver_ioctl() Routine**

The `driver_ioctl()` routine is used to execute driver-specific control functions. The `driver_ioctl()` routine is provided by the driver writer. It can have any unique name. Pass the name to WSIO Services by specifying it in the `d_ioctl` field of the `drv_ops` structure. Commonly, `driver` is replaced by the driver's name.

The `ioctl()` system call allows drivers to perform driver-dependent control functions on character devices. See `ioctl (2)` and `ioctl (5)` in the *HP-UX Reference*. Because devices vary in the control functions they support, this system call is flexible, which means the developer can implement the control functions the device requires.

```c
int driver_ioctl(dev_t dev, int cmd, caddr_t data, int flag);
```

- `dev` The device number of the associated device.
- `cmd` The command word described in greater detail later.
- `data` Pointer to the command's arguments, if any.
- `flag` The file-access flags. Most drivers ignore this parameter.

You can use `ioctl()` to:

- Modify a driver's behavior.
- Modify the configuration of a device.
- Implement any special processing not provided by other system calls.

*HP-UX Reference* describes the functions of `ioctl()` for existing HP-UX drivers.

User programs call `ioctl()` using the following declaration:

```c
int ioctl(int fildes, int request type arg);
```

The parameters are:

- `fildes` A file descriptor obtained from an `open()` or a `dup()` call made earlier.
- `request` The command word, a 32-bit integer that specifies the size of `arg`, whether `arg` is passed to the driver or returned by the driver or both, and the command to perform. (The following section describes the command word in detail.)
- `arg` The type and value of `arg` is driver dependent.

The requests specified in the `requests` field have two varieties:

1. Requests to be processed by one driver.
2. Requests to be processed by more than one driver.

Request names in the form `Fxxxx` are reserved for those requests that are general enough to be implemented by several different device drivers that are trying to do the same sort of thing. `FIOASYNC` is an example of these requests. On choosing to implement any of these requests, th driver should process the request in a way that is consistent with other drivers that use them. Typical `ioctl()` requests include rewinding a tape and changing a printer's column width. (Refer to `ioctl (5)` in the *HP-UX Reference* for a list of these requests and
the standard processing your driver should perform.) Examine the header files in /usr/include/sys for examples of ioctl() command definitions. The command `grep 'define.*_IO' /usr/include/sys/* | more` will give you a large list of the commands used by many device drivers.

Command words are 32-bit integer values used for the ioctl() request argument. Define the command words for the driver in a header file. User programs that issue ioctl() calls for your driver must include this file.

Use one of the following statements to define each command word:

```c
#define command _IO('t', n)
#define command _IOR('t', n, object)
#define command _IOW('t', n, object)
#define command _IOWR('t', n, object)
```

The _IO* routine names are defined in <sys/ioctl.h>. They combine the t, n, and object parameters into a 32-bit integer. They specify how the kernel is to copy the data structure to which arg points between the user’s address space and the kernel’s address space. Normally, arg is seen by the driver as a pointer to a kernel buffer.

- **_IOR** Read data from the driver. That is, the driver writes into the kernel buffer pointed to by arg. Before returning to the user, the system copies the kernel buffer to the user specified buffer.
- **_IOW** Write data to the driver. That is, the driver reads from the kernel buffer pointed to by arg. Before calling the driver, the system copies the user specified buffer to the kernel buffer.
- **_IOWR** Both _IOR and _IOW.
- **_IO** Indicates that the ioctl command does not pass in an argument.

The parameters are defined as follows:

- **command** The identifier you assign to your command.
- **t** An arbitrary character of your choice, used to associate the ioctl() call with your driver. Use the command `grep 'define.*_IO' /usr/include/sys/* | more` to ensure that you are not choosing a value (after macro expansion) that can conflict with another driver.
- **n** A number (0 to 127) that identifies a driver-specific command for the driver.
- **object** The type of object to which arg points. The object has a size limit of 16 KB.

**driver_ioctl()** is expected to return the following values:

- **0** Successful completion.
- **><0** Error. See ioctl (2), ioctl (5) and errno (2) for a list of standard error values. ioctl() returns the error value to the user process in errno.

**LP64 Considerations**

The ioctl commands generated by the _IO macro have the same encoded values for both ILP32 and LP64 data models. The _IOR, _IOW, and _IOWR macros, however, generate different encoded values where the sizeof(object) differs between the two data models.

The driver requires no changes when sizeof(object) is fixed in size for both ILP32 and LP64; but if the size is scalable, the best method of handling this is to have the driver accommodate two versions (one for each data model) of the ioctl. The following example demonstrates this.

Consider an ioctl cmd that specifies the scalable data type long.

```c
/* Public Header File */
#define SOME_IOCTL _IOR('X', 1, long)
/* Private Header File */
/* IOCTL for 32 Bit Applications */
```
ifdef __LP64__
#define SOME_IOCTL_32 _IOR('X', 1, int)
/* from 32-bit app */
#endif __LP64__
/* Driver ioctl code snippet */
switch (cmd) {
  case SOME_IOCTL:
    ifdef __LP64__
    case SOME_IOCTL_32:
      <do SOME_IOCTL processing>
    endif /* __LP64__ */
  }

Consider an ioctl cmd that specifies a structure with a pointer data type as a data member.

#define COPY_IOCTL _IOWR('X', 2, struct buf_copy)
struct buf_copy {
  caddr_t buf_ptr;
  int32_t buf_size;
}
/* Private Header File */
#ifdef __LP64__
#define COPY_IOCTL_32 _IOWR('X', 2, struct buf_copy_32)
struct buf_copy_32 {
  ptr32_t buf_ptr32;
  int32_t buf_size;
}
#endif /* __LP64__ */
/* Driver ioctl code snippet */
switch (cmd) {
  case COPY_IOCTL:
    <do COPY_IOCTL processing>
    break;
  /* On 64 bit kernels these ioctls will process ioctls made by 32 bit applications */
  #ifdef __LP64__
  case COPY_IOCTL_32:
    <do COPY_IOCTL_32 processing>
    break;
  #endif /* __LP64__ */
}

To copy the 32-bit application buffer, the driver calls copyin(). For example,

    copyin((caddr_t)(struct _buf_copy_32 *)data->buf_ptr32), mybuf, sizeof(mybuf));

The copyin() and copyout() functions in the 64-bit kernel check to determine whether the calling thread is a 32-bit application and, if so, do the necessary 32-to-64-bit address conversion.

If there is a need with _IO to determine whether the calling thread is a 32-bit or 64-bit application, define two cmd values: one for ILP32 and another for LP64. The following example demonstrates this.

/* Public Header File */
#ifdef __LP64__
#define LONG_IOCTL _IOW('A',1)
#else
#define LONG_IOCTL _IOW('a',1)
#endif /* __LP64__ */
/* Private Header File */
#ifdef __LP64__
#define _LONG_IOCTL_32 _IOW('A',1)
/* long data from 32 bit app */
#endif /* __LP64__ */

/* Driver ioctl code snippet */
switch (cmd) {
Example

The `mydevice_ioctl()` routine implements the `ioctl()` commands defined for the `mydevice` driver. Example code follows.

Public Header file (`mydevice.h`) used to define `ioctl` commands:

```c
#include <sys/ioctl.h>
struct mydevice_ioctl_arg {
    char reg_value;
    caddr_t location;
};
#define CLEAR 0
#define SET 1
#define CARD_RESET _IO ( 'X', 0 );
#define CARD_STATUS _IOR ( 'X', 1, struct mydevice_ioctl_arg);
#define CARD_CONTROL _IOW ( 'X', 2, struct mydevice_ioctl_arg);
#define CARD_BUFADR _IOWR ( 'X', 3, struct mydevice_ioctl_arg);
```

This example defines four commands that the driver performs:

- **CARD_BUFADR**: Assigns a memory location to the buffer on the device. The `location` field of the `mydevice_ioctl_arg` structure tells the driver where the buffer should be located in memory. If `location` is 0, the driver uses a default location. The location of the buffer is returned to the user in the `location` field.

- **CARD_CONTROL**: Sets or clears the bits in the device's control register. In the `mydevice_ioctl_arg` structure, the `reg_value` field specifies the bits to be affected, and field `location` contains `SET` or `CLEAR` to indicate the action the driver is to take.

- **CARD_RESET**: Resets the device to its default state.

- **CARD_STATUS**: Returns the contents of the device's status register to the user in the `reg_value` field of the `mydevice_ioctl_arg` structure.

User program code segment, which sets a bit in the device's control register:

```c
#include <sys/errno.h>
#include "mydevice.h"
#define SET_TIMEOUT 0x04
#define MEMDEFAULT 0
struct mydevice_ioctl_arg ioctl_arg;
ioctl_arg.reg_value = SET_TIMEOUT;
ioctl_arg.location = SET;
if (ioctl(fd, CARD_CONTROL, &ioctl_arg) < 0)
    printf("ioctl call failed, error number = %d\n", errno);
```

Private Header file to handle 32-bit applications running on a 64-bit OS:

```c
#ifdef __LP64__
struct mydevice_ioctl_arg32 {
    char reg_value32;
    ptr32_t location32;
};
```

```c
```
define CARD_STATUS_32  IOR ('X', 1, struct mydevice_ioctl_arg32 );
define CARD_CONTROL_32 _IOW ('X', 2, struct mydevice_ioctl_arg32 );
define CARD_BUFADR_32 _IOWR ('X', 3, struct mydevice_ioctl_arg32 );
#endif

Driver code snippet to implement the ioctls. Note that the global variable my_device registers point to the registers of a hypothetical piece of hardware.

/* header files this code segment needs */
#include <sys/errno.h>
#include <sys/type.h>
#include <sys/mydevice.h>

struct my_device_registers *dev_rp;
mydevice_ioctl( dev_t dev,
  int cmd,
  struct mydevice_ioctl_arg *arg,
  int flag )
{
  #ifdef __LP64__
    struct mydevice_ioctl_arg32 *arg32 =
      (struct mydevice_ioctl_arg32 *) arg;
  #endif

  switch(cmd)
  {
  case CARD_RESET:
    dev_rp->reset = 0;
    return(0);
  case CARD_STATUS:
    arg->reg_value = dev_rp->status;
    return(0);
  case CARD_CONTROL:
    switch(arg->location)
    {
    case SET:
      dev_rp->control |=arg->reg_value;
      return(0);
    case CLEAR:
      dev_rp->control &= ~arg->reg_value;
      default:
      return(EINVAL);
    } /* switch */
  case CARD_BUFADR:
    arg->location = set_buf_addr(arg->location);
    return(0);
  #ifdef __LP64__
  case CARD_STATUS_32:
    arg32->reg_value = dev_rp->status;
    return(0);
  case CARD_CONTROL_32:
    switch(arg32->location)
    {
    case SET:
      dev_rp->control |=arg32->reg_value;
      return(0);
    case CLEAR:
      dev_rp->control &= ~arg32->reg_value;
      default:
      return(EINVAL);
    } /* switch */
  #endif
  #endif
}
Writing a Driver

Step 6: Writing Entry Point Routines

return(0);

case CARD_BUFADR_32:
    arg32->location = set_buf_addr(arg32->location);
    return(0);
#endif

default;
    return(EINVAL);
}
}

Writing a driver_minphys() Routine

The driver_minphys() routine adjusts a physio() transfer count into the size the driver_strategy() can use when the system supplied minphys() routine does not provide the correct transfer count for the device.

The driver_minphys() routine compares y->b_bcount with whatever transfer size your device requires. If bp->b_bcount is larger, then bp->b_bcount is set to the devices transfer size. Otherwise, bp->b_bcount is unchanged.

#include <sys/types.h>
#include <sys/buf.h>

void driver_minphys(struct buf *bp);

bp

minphys()

driver_minphys()

Example

#include <sys/types.h>
#include <sys/buf.h>
#define MYDRIVERPHYS 0x800

static int
mydriver_write(dev_t dev, struct uio * uiop)
{
    return physio(mydriver_strategy,NULL, dev, B_WRITE,
                 driver_minphys, uio);
}

static void
mydriver_minphys(struct buf *bp)
{
    if (bp->b_bcount > MYDRIVERPHYS)
        bp->b_bcount = MYDRIVERPHYS
    return 0;
}
Writing a driver_select() Routine

The driver_select() routine is used to test I/O completion on a device. This routine is provided by the driver writer. It can have any unique name. Pass the name to WSIO Services by specifying it in the d_select field of the drv_ops structure. Commonly, driver is replaced by your driver's name.

The select() system call invokes a driver_select() routine to determine whether I/O has been completed or is ready, or whether an exceptional condition exists (see select (2). Use select() and driver_select() only for character devices.

Performing select() on device files can have different interpretations depending on the device. Use select() to poll a device for status.

The driver of a character device should return “true” (a non-zero value) if its device is always ready for I/O. A character driver that does not have a driver.select() routine should always return “true”.

Do this by specifying the kernel function seltrue() in the d_select field of the drv_ops structure.

The driver_select() routine has no access to the readfds, writefds, and exceptfds values that the user passed to the select() system call. (See select (2) in the HP-UX Reference.) The driver_select() routine is passed only the device number and flag.

The driver must check for collisions in which two or more threads call select() on the same device file for the same condition. To do this, the driver saves the pointer to the calling kernel thread and uses that pointer as the argument to waiting_in_select() when another thread enters driver_select(). If multiple threads are waiting on a select condition, the driver must set the collision argument to selwakeup() when the select condition becomes true.

NOTE Calling waiting_in_select is new to HP-UX 11i. In prior releases the driver checked if the thread was sleeping on the global variable selwait.

The mask returned to the user applies only to the moment when the driver_select() routine was invoked. This means that, if select() returns true for a file descriptor, it does not guarantee that the device will still be ready when a read or write is later issued for this file descriptor.

For each file descriptor that a user specifies for the select() system call, select() invokes the corresponding driver_select() routine. If more than one file descriptor has the same major number, select() invokes the corresponding driver_select() routine once for each file descriptor.

```c
int driver_select(dev_t dev, int flag);
```

- **dev** The device number.
- **flag** The type of readiness to test, according to the following values:
  - FREAD Read
  - FWRITE Write
  - 0 Exception conditions

**driver_select() is expected to return the following values:**

- <>0 True. The device or driver is ready for read or write or an exception condition was found. The kernel sets the corresponding bit in the bit-mask field that select() returns to the user.

- 0 False. The device or driver is not ready for read or write or no exception condition was found. select() puts the calling process to sleep until the condition becomes true. The driver must inform the system when this condition becomes true.
The `select()` system call handles collisions if a true value is passed in the `selwakeup()` call’s second parameter. A true return (non-zero) indicates the select succeeded. When false is returned the `select(2)` system call sleeps and waits.

```c
#include <sys/types.h>
#include <sys/param.h> /* for user.h */
#include <sys/user.h>   /* for u def */
#include <sys/kthread_iface.h>
#include <sys/file.h>   /* for FREAD, FWRITE */

struct my_sel_struct
{
    struct kthread *read_waiter;
    struct kthread *write_waiter;
    int state;
};
extern int selwait;
extern lock_t *mylock; /* my driver’s spinlock */
struct my_sel_struct *my_sel_struct;

static int
skel_select(dev_t dev, int flag)
{
    struct kthread *t;
    spinlock(my_lock);
    switch(flag) {
        case FREAD:
            if (available data) {
                spinunlock(my_lock);
                return 1;
            }
            if ((t=my_sel_struct->read_waiter) &&
                waiting_in_select(t))
                mysel_struct->state |= READ_COLLISION;
            else
                mysel_struct->read_waiter = u.u_kthreadp;
            break;
        case FWRITE:
            if (ready for more data) {
                spinunlock(my_lock);
                return 1;
            }
            if ((t=my_sel_struct->write_waiter) &&
                waiting_in_select(t))
                mysel_struct->state |= WRITE_COLLISION;
            else
                mysel_struct->write_waiter = u.u_kthreadp;
            break;
    } /* end switch */
    spinunlock(my_lock);
    return 0;
}
```

When the driver knows there is more input, or knows output can be started, it calls `selwakeup()` to awaken all processes sleeping for this condition. The `skel_output_ready()` routine is called when the driver finds that the device is ready to output more characters. The skeleton routine `skel_output_ready()` looks like this:

```c
/*
 * Wake up any writers when driver is ready for more
 * output.
 */
static void
```
skel_output_ready(struct my_sel_struct * mystruct)
{
    /*
     * Wake up any process sleeping on a select for this
     * condition.
     */
    if (mystruct->write_waiter) {
        selwakeup(mystruct->write_waiter,
                  mystruct->state & WRITE_COLLISION);
        mystruct->write_waiter = NULL;
        mystruct->state &= ~WRITE_COLLISION;
    }
}

The driver calls the skel_input_ready() routine when the device/driver has input available. The skeleton
routine skel_input_ready(), which awakens all processes sleeping for the read condition looks like this:

    /*
    * wakeup any readers when driver has more input
    * available
    */
    static void
    skel_input_ready(struct my_sel_struct * mystruct)
    {
        /*
         * if a process is sleeping on select for
         * this condition - wake it up
         */
        if (mystruct->read_waiter) {
            selwakeup(mystruct->read_waiter,
                      mystruct->state & READ_COLLISION);
            mystruct->read_waiter = NULL;
            mystruct->state &= ~READ_COLLISION;
        }
    }

Writing a driver_strategy() Routine for a Block Device

The driver_strategy() routine is used to execute block read or write for character or block devices This
routine is provided by the driver writer. It can have any unique name.

For a block device, pass the name to WSIO Services by specifying it in the driver_strategy field of the
drv_ops structure. For a character device, pass the name as a parameter of physio(). Commonly, driver is
replaced by the driver's name.

driver_strategy() functions provide block I/O for block and character devices. One is required for block
devices; one is optional for character devices. Often, for a device with block and character access, the same
routine is used for both accesses, since most of the code is usually the same for the two methods.

driver_strategy() is called by the file system as a result of a read or a write on an ordinary file, a directory,
or a block device. It is called via physio() by the driver_read() or driver_write() routine as a result of a
read or write on a character device.

Most of the following discussion explicitly refers to a driver_strategy() routine for a block device. There
are subtle differences for a character device such as mapping of the user buffer and allocation of the buf
structure. See physio (KER2) for additional information if the developer is writing driver_strategy() routine for a character device.

Use a driver_strategy() routine to perform I/O to or from the device. The tasks this routine performs are:

- Initializing data structures, such as DMA buf headers.
After scheduling an I/O request, the `driver_strategy()` routine returns control to the routine that invoked it. The `driver_strategy()` routine must not call `sleep()`, because a strategy routine may be executing on the interrupt stack. The process that invokes `driver_strategy()` determines whether to wait for the I/O to be completed.

On completing the I/O request, the driver's lower half should do the following:

1. If an error occurs, it should set `B_ERROR` in `b_flags` and put a value in `b_error` in the `buf` structure.
2. It should set `b_resid` to indicate the amount of data remaining to be transferred.
3. It should awaken the driver's top half by calling `biodone()`.

```c
void driver_strategy (struct buf * bp);
```

- `bp` A pointer to a `buf` structure, which contains all the information that the `driver_strategy()` routine needs to process the request.

### Using `driver_strategy()` for Write

When a user process calls the `write()` system call for a block device, the kernel allocates a `buf` structure and a kernel buffer for the I/O request. The kernel associates the buffer with the device number and block number that the buffer represents.

The kernel fills the buffer header with information that describes the I/O request. For example, the kernel sets the `B_WRITE` flag in `b_flags` to indicate to the `driver_strategy()` routine that the request is a write request.

Provided that the `C_MAP_BUFFER_TO_KERNEL` flag was set in the driver's `drv_opts_t` structure, the kernel maps the data from the user's data area into the kernel's buffer. The kernel then sets the `b_un.b_addr` field in the `buf` header to point to this kernel buffer. Then the kernel calls the `driver_strategy()` routine, passing it a pointer to the `buf` structure as a parameter. The `driver_strategy()` routine now has exclusive access to this kernel buffer because the `B_BUSY` flag is set in `bp->b_flags`.

The `driver_strategy()` routine schedules the I/O to the device. For write requests, the `driver_strategy()` routine schedules (usually through DMA queues) the data in the kernel buffer to be copied by DMA to the device. The `driver_strategy()` routine should then return control to the routine that invoked it.

If the request is for an asynchronous write, the `write()` system call does not wait for the I/O to be completed. The `write()` system call returns control to the user, so the value returned to the user process simply indicates that the data has been successfully copied to the buffer, and scheduled for I/O. If the write is synchronous, `write()` calls `biowait()` to wait for the I/O to be completed.

When the I/O is completed, the lower half of the driver sets `b_resid` to the amount of data remaining to be transferred. If an error occurred, it sets `B_ERROR` in `bp->b_flags` and sets `bp->b_error` to an `errno` value. Then it calls `biodone()`, which awakens all processes sleeping on the buffer. The kernel frees this buffer and the `buf` structure, which another process can now use.
Using driver_strategy() for Read

For `read()` system calls for block devices, the kernel first looks for the requested data in the buffer cache. If present, the kernel copies the data from the kernel's buffer to the user's data area and returns control to the calling process without calling the `driver_strategy()` routine. If the data is not in the buffer cache, the kernel allocates a `buf` structure and a kernel buffer for the I/O request. The kernel fills the buffer header with information that describes the I/O request. For example, the kernel sets the `B_READ` flag to indicate to the `driver_strategy()` routine that the request is a read request.

For read requests, the `driver_strategy()` routine schedules the data to be copied from the device to the kernel buffer. The `driver_strategy()` routine should then return to the routine that invoked it. For read requests on block device files, the kernel always waits for the I/O to complete before returning to the user. The `read()` system call invokes `biowait()`, and waits for the I/O to complete.

When the I/O completes, the lower half of the driver should set `bp->b_resid` to the amount of data remaining to be transferred, set `B_ERROR` and set `bp->b_error` to an `errno` value if an error occurred, and call `biodone()`. The `read()` system call copies the data in the kernel buffer into the user's data area. The requested data is now available to the user process. The kernel releases the `buf` structure by clearing the `B_BUSY` flag, and calls `wakeup()` to wake up any processes sleeping on the buffer.

Examples

The following code shows a driver routine named `skel_strategy()`, derived from an actual device driver; as a result, some references are hardware specific. This driver uses PIO instead of DMA.

```c
#include <sys/types.h>
#include <sys/errno.h>
#include <sys/vme2.h>
#include <sys/buf.h>

struct skelregs *skel;  /* board registers */
struct buf *skelbuf;    /* io buffer */
char r_int_enable_reg;  /* software reg copy */

static void
skel_strategy(struct buf * bp)
{
    extern lock_t *my_lock;
    register caddr_t addr;
    register short cnt;
    struct isc_table_type *isc;
    isc = wsio_get_isc ( bp->b_dev,&isc,&skel_wsio_info )
    spinlock( my_lock );
    addr = bp->b_un.b_addr;
    cnt = bp->b_bcount;
    /* set up device */
    /* isc->if_reg_ptr set up in attach */
    skel = (struct skelregs *)isc->if_reg_ptr;
    skel->registerX = ..........

    if (bp->b_flags & B_READ)
    {
        /* This device doesn't read */
        bp->b_flags |= B_DONE;
        spinunlock( my_lock );
        return;
    }
    else
    {
        /* Complete Write Transfer */
```

Chapter 4
if (!cnt)
{
    bp->b_flags |= B_DONE;
    spinunlock( my_lock );
    return;
}
else
    skel_start(bp);
}
spinunlock( my_lock );
}

void
skel_start(struct buf *bp)
{
    addr = bp->b_un.b_addr;
    cnt = bp->b_bcount;
    /* special last byte setup, if needed */
    if (cnt == 1) /* last byte, do it now */
        skel->control = AUX_SEOI;

    skel->int_enable_reg = DOIE;
    skel->data_out_reg = *addr++;
    bp->b_bcount = cnt-1;
    bp->b_un.b_addr++ ;
}

static void
skel_isr(void)
{
    skel->ch1.status_reg = D_CLEAR;
    skel->int_enable_reg = DOIE;
    r_int_enable_reg |= skel->int_enable_reg;
    r_int_enable_reg &= ~DO;

    cnt = skelbuf->b_bcount;
    if (cnt == 0)
        biodone(skelbuf);
    else
        skel_start(skelbuf);
}
Step 7: Writing Other Driver Routines

This section describes primary driver routines that are not defined entry point routines; that is, they are not called through entry points defined in fields of your `drv_ops_t` structure. Interface drivers typically have `driver_attach()` and `driver_isr()` routines. They also may have `driver_if_init()` and `driver_probe()` routines.

When an interface is shared by multiple device drivers, some method of linkage is required between the two types of drivers. Typically this is done using some type of I/O “switch” structure, as described in “I/O Switch Tables” section in Chapter 2, “HP-UX I/O Subsystem Features.”

Device drivers also may have `driver_dev_init()` routines. Some of this material has been covered in previous sections, so this section will cover the `driver_isr()` and `driver_psize()` routines.

Interface management and device queue management are also briefly discussed.

Writing a `driver_isr()` Routine

The `driver_isr()` function is used to handle device interrupts in interrupt context.

The `driver_isr()` routine is provided by the driver writer. It can have any unique name. Pass the name to WSIO Services by specifying it as a parameter of the `wsio_intr_alloc()` function, executed in the `driver_attach()` or `driver_if_init()` routine. Commonly, `driver` is replaced by the driver’s name.

The WSIO service `wsio_intr_alloc()` has the following interface:

```c
int wsio_intr_alloc (struct isc_table_type * isc, 
                    wsio_drv_isr_t driver_isr, 
                    uintptr_t arg, 
                    uint64_t flags, 
                    wsio_intr_object_t * wsio_intr)
```

The first parameter is a pointer to the `isc_table` entry that represents the device. The next two arguments, `driver_isr` and `arg` are the driver’s isr routine and the argument that will be passed into the `driver_isr` when it is called. Typically the driver specifies the `isc` pointer as the argument. The next argument is a flags field. The flags can be used to indicate if the driver wants an exclusive or shared interrupt resource. With shared interrupt a driver’s ISR can get called when its device did not interrupt. If this is not acceptable the driver should set the flag to `WSIO_INTR_EXCLUSIVE`.

The last argument is a pointer to an interrupt handle which `wsio_intr_alloc()` returns to the driver. The driver must pass this handle into the other WSIO interrupt services such as `wsio_intr_activate()` which enables the interrupt.

There are additional WSIO interrupt services that allow the driver to specify whether it wants to use a line based or transaction based interrupt. For example, if the driver uses a line based interrupt it would call the service `wsio_intr_set_irq_line()` to specify a line based interrupt and then `wsio_intr_get_irq_line()` to get the IRQ line. On the other hand, if the driver wanted to use a transaction based interrupt it would call `wsio_intr_set_cpu_spec()` to set up a transaction based interrupt and `wsio_intr_get_txn_info()` to get the transaction based address and vector. Of course, the underlying platform hardware may limit what type of interrupt a driver can use. Some I/O controllers do not allow interface cards underneath them to use transaction based interrupts.

In an interface driver or a monolithic driver, the ISR processes interrupts from an interface card. The ISR performs the following tasks:

- Stops the interface card from interrupting.
- Determines a reason for the interrupt (if appropriate).
Take appropriate action, such as cleanup or retry.

Calls `wakeup()` or `biodone()`, or initiates the next step in processing an I/O request.

Devices may share interrupt resources. The ISR associated with each device's driver may be called for interrupts not originating from its device. The ISR should be able to handle this and return 0 to the caller. Otherwise, the ISR returns 1, indicating that the interrupt has been serviced.

An ISR executes in an interrupt context, not a kernel thread context. Therefore, an ISR must never call `sleep()` or a function that may block. An ISR has the following interface:

```c
int driver_isr (long arg1);
```

- **arg1**: Driver defined parameter passed in the call to `isrlink()` as `arg1`.

`driver_isr()` is expected to return the following values:

- 0: The card does not belong to this driver.
- 1: This routine handled the interrupt.

### Examples

The following are examples of `driver_if_init` and `driver_isr` functions. First, the `driver_if_init` function calls `wsio_intr_alloc()` to allocate an interrupt handle and register its `driver_isr` and `arg`. Then it calls `wsio_intr_set_irq_line()` to specify a line based interrupt and `wsio_intr_activate()` to enable the interrupt.

#### NOTE

The driver passes in the flag `WSIO_IRQ_LINE_AUTO` to `wsio_intr_set_irq_line()` which informs the services that they should determine the IRQ line value for the particular device.

```c
static int CentIF_init (struct isc_table_type * isc)
{
    « do any additional driver initialization »
    …………………………………
    …………………………………
    /* Allocate an interrupt object */
    status = wsio_intr_alloc(isc, CentIf_isr, (uintptr_t)isc,
        0, &intr_obj);
    if(status != WSIO_OK)
        return(ERROR);
    /* Get a level Sensitive IRQ */
    status = wsio_intr_set_irq_line(isc, intr_obj,
        WSIO_IRQ_LINE_AUTO, 0);
    if status != WSIO_OK)
        return(ERROR);
    /* Active the interrupt */
    status = wsio_intr_activate(isc, intr_obj);
    if(status != WSIO_OK)
        return(ERROR);
    else
        return(wsio.ok)
}
```
The code that follows is an ISR routine for the centif driver.

```c
static int
CentIf_isr(long arg1, long arg2)
{
    static struct sw_intloc intloc;
    struct isc_table_type * isc;
    PortData_t * pdp;

    isc = (struct isc_table_type *)arg1;
    pdp = (PortData_t *)isc->if_drv_data;

    if (pdp->pd_intr_reg == INTR_READ) {
        pdp->pd_intr_reg = INTR_INFO;
        sw_trigger(&intloc, RealIntrHndlr, isc, 3, 0);
        return 1; /* interrupt has been serviced */
    } else {
        return 0; /* interrupt not from my device */
    }
}
```

**Writing a driver_psize() Routine**

The `driver_psize()` function is used to get the swap partition size of a device.

```c
int driver_psize (dev_t dev);
```

`dev` Contains encoded major and minor numbers;

The `driver_psize()` WSIO function is provided by the driver writer. It can have any unique name. You pass the name to WSIO Services by specifying it in the `d_psize` field of the `drv_ops` structure. Commonly, `driver` is replaced by the driver's name.

The `driver_psize()` WSIO function should return the size of the swap partition on a block swapping device, It is called by the kernel. Consider writing this routine only if the device is used for swapping.

`driver_psize()` returns the following values:

>0 Successful completion. The value is the swap partition size.

-1 Error.

This SCSI example assumes that `driver_psize()` is never called when the device is closed, hence there is no need to do an explicit open and close in the routine. Note the use of the SCSI Services `m_scsi_lun()` function.

```c
static int
mydriver_psize(dev_t dev)
{
    struct scsi_lun *lp = m_scsi_lun(dev);
    struct mydriver_lun *llp = lp->dd_lun;
    int nblks, rshift;

    nblks = llp->nblks;
    rshift = llp->devb_lshift;

    return (rshift > 0 ? nblks >> rshift : nblks << -rshift);
}
```
Interrupt-Driven Routines for Device Drivers

This section discusses the routines the developer can use to manage device queues for device drivers that are interrupt-driven rather than context-driven.

If writing an interrupt-driven driver that also requires management of device queues the developer will also need to deal with problems related to the environment (registers, variable states, and so on).

Management of Device Queues

If more than one instance of the device driver can run simultaneously, the developer needs to provide the driver with a device queue. This will prevent the driver from sending requests to the device faster than the device can complete them. A device queue allows requests to be queued, to await their turn for the device, and to have their I/O completed.

Routines that manage device queues use the buf data structure. Be familiar with this data structure, its contents, and its intended usage.

This section describes only the fields in the structure that are explicitly used in managing device queues.

The buf Structure The buf structure is defined in <sys/buf.h>. See buf (KER4) in the HP-UX 11i v1 Driver Development Reference. All I/O requests end up as a buf structure. The buf structure contains information on the current and potential owners for devices, and contains all information necessary for a driver to complete the I/O request.

The following buf fields are of particular importance in the context of device queue management (refer to the code for a complete list):

av_forw Points to the next buf structure in the queue. Its value is NULL if the current buf structure is the last one in the queue.

Interface Management

Whenever more than one device can reside on an interface card, the interface must be managed. For example, an HP-IB bus requires such management, but a built-in parallel interface does not. Because all the devices on an interface card cannot do I/O at the same time, each interface card must have an owner that has the sole ability to do I/O on devices connected to the card.

Fields in the buf structure describe the owner of an interface. Because there can only be one owner of an interface card at a time, interface drivers usually use a queue to hold buf structures waiting for their chance to own the interface card. There is one queue for each interface.

Data Structures for Managing Interface Cards Routines that manage interface cards use two data structures, buf and isc_table_type. You need to be familiar with these data structures, their contents, and their intended usage.

The following sections describe the fields in these structures that are used to manage interface cards.

The isc_table_type Structure The isc_table_type structure is defined in <sys/io.h>. See isc_table_type (WSIO4) in the HP-UX 11i v1 Driver Development Reference. The isc_table_type structure contains all pertinent information about the interface driver space. This structure is initialized by the WSIO before calling the driver's attach routine. When claiming an interface card the driver will initialize some additional fields such as the isc->gfsw_init field.
New WSIO Services for 11i

This section describes the new WSIO services that are introduced in 11i. Some of these services replace the functionality provided by older WSIO services, while others provide entirely new functionality. Driver developers are encouraged to convert to the new services, as these services provide enhanced functionality over the older ones.

The new services can be grouped into the following sets:

1. I/O Space Services
   - Register Services
   - Configuration Space Services
   - I/O Port Space Services
   - Endian Services
2. DMA Services
3. Interrupt Services
4. Memory Allocation Services
5. Driver Event Handling Services
6. System Services
   - Description Service
   - System Attribute Services
   - I/O Synchronization Services

The following provides an overview of each set of services. For detailed information on how to use the services driver, writers should consult the particular manpage.

- **Register Services** — Drivers can call these services to discover, map and access memory mapped registers. Although the first register has already been mapped and is hence readily available to drivers, they can use these services to discover and map additional register sets. These services replace the legacy WSIO services:

  ```
  map_mem_to_host()
  unmap_mem_to_host()
  ```

- **Configuration Space Services** — This set of services can be used by device drivers to discover and access an IO card’s configuration space. It replaces services provided by other CDIOs such as the PCI CDIO.

- **IO Port Space Services** — This set of WSIO services allows a driver to discover, map and access a card’s I/O port space. There are no equivalent legacy WSIO services that they replace.

- **Endian Services** — These WSIO services can be used by device drivers to determine the endianness of the local bus. Drivers can use them to determine whether to perform endian translation for I/O port accesses or memory mapped registers.

- **DMA Services** — These new WSIO DMA services replace the older legacy WSIO DMA services (`wsio_map`, `wsio_fastmap`, `wsio_remap`, `wsio_unmap`, etc.). The advantages of the new services are that they allow drivers to allocate multiple DMA objects and tune each object for different types of DMA. They also allow drivers to register callback functions when resources become available.
Interrupt Services — This new set of WSIO services replaces the older WSIO DMA which include `isrlink()`. They allow drivers to claim and configure interrupt resources for their cards. With the new services, drivers can allocate multiple interrupt objects per card and configure each one to use different resources.

Description Service — This service can be used by drivers to set the description of a device. The description appears in the ioscan output.

Ordered Interrupt Service — This service can be used by drivers to determine whether the I/O system is ordered with respect to interrupts.

I/O Synchronization Service — This is a function that is used by device drivers to synchronize the CPU and the hardware device's view of memory.

System Attribute Services — This set of services allows the driver to obtain information about the platform.

Driver Event Handling Services — The event handling services allow drivers to register both an event handler and event mask. The driver's event handler is called to handle events and the event mask indicates which events the driver is interested in.

Memory Allocation Services — The new WSIO memory allocation services allow drivers to allocate host memory. With the new services a driver can specify different types of memory it wants to allocate, such as memory below four gigabytes (32-bit memory addresses) or physically contiguous memory.

Register Services

These services are used by device drivers to discover and map device registers. The first register set of a device is usually mapped in by the kernel, but the driver can discover additional register sets and map them when using these services. An interface driver would usually do this in its `attach` or `init` routine. Registers can contain information about the hardware device or sometimes be used to configure and fine-tune the device. The functions provided are listed and briefly described in Table 4-7, “Register Services.” For further information concerning parameters, return codes and example codes, consult that function’s manpage.

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wsio_map_reg()</code></td>
<td>Maps a device register to host memory</td>
</tr>
<tr>
<td><code>wsio_unmap_reg()</code></td>
<td>Unmaps a device register</td>
</tr>
<tr>
<td><code>wsio_get_all_registers()</code></td>
<td>Gets an array of all available device registers</td>
</tr>
<tr>
<td><code>wsio_read_regXX()</code></td>
<td>Reads from a mapped device register</td>
</tr>
<tr>
<td><code>wsio_write_regXX()</code></td>
<td>Writes to a device register</td>
</tr>
</tbody>
</table>
Configuration Space Services

These services are used by device drivers to discover and use configuration space resources. Configuration space is an I/O space that can contain information about the hardware device as well as some locations that are used to configure and fine-tune the device. The functions provided are listed and briefly described in Table 4-8, “Configuration Space Services.” For further information on each routine, including parameter information, return code information, and example code, the manpage for that function should be consulted.

Table 4-8 Configuration Space Services

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wsio_map_cfg_handle()</td>
<td>Obtain a configuration space access handle</td>
</tr>
<tr>
<td>wsio_unmap_cfg_handle()</td>
<td>Release a configuration space handle</td>
</tr>
<tr>
<td>wsio_cfg_inXX()</td>
<td>Reads from configuration space</td>
</tr>
<tr>
<td>wsio_cfg_outXX()</td>
<td>Writes to configuration space</td>
</tr>
</tbody>
</table>

I/O Port Space Services

These services are used by device drivers to obtain access to I/O Port Space. I/O Port Space is an I/O space that is sometimes necessary to communicate with devices. I/O Space accesses do not do any endian checking so, if necessary, a driver will have to perform any necessary endian translation. The functions provided are listed and briefly described in Table 4-9, “I/O Port Space Services.” For further information on each routine, including parameter information, return code information, and example code, the manpage for that function should be consulted.

Table 4-9 I/O Port Space Services

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wsio_map_port()</td>
<td>Obtains an I/O port handle</td>
</tr>
<tr>
<td>wsio_unmap_port()</td>
<td>Unmaps an I/O port</td>
</tr>
<tr>
<td>wsio_get_ioports()</td>
<td>Obtains the addresses and sizes of I/O ports</td>
</tr>
<tr>
<td>wsio_port_inXX()</td>
<td>Reads from an I/O port</td>
</tr>
<tr>
<td>wsio_port_outXX()</td>
<td>Writes to an I/O port</td>
</tr>
</tbody>
</table>

Endian Services

These services are used by device drivers to determine the endianness of the local bus. This service is necessary to decide whether to perform endian translation for shared memory accesses. Such translation is automatically performed for any access to or from configuration space, but for access to registers, I/O ports, or shared memory, the device driver might need to perform their own translation. The functions provided for this service are listed and briefly described in Table 4-10, “Endian Services.” For further information on each routine, including parameter information, return code information, and example code, the manpage for that function should be consulted.
DMA Services

These services are used by device drivers to configure and use DMA resources. DMA is a service that allows memory to be shared between an I/O device and the host processor’s main memory. Although routines already exist in WSIO to use DMA functionality, these new interfaces allow device drivers to: specify callback functions for allocating resources, allocate multiple DMA handles that can each be associated with different sets of hints, and specify the number of bits of addressing that the device will use. The functions provided are listed and briefly described in Table 4-11, “DMA Services.” For further information on each routine, including parameter information, return code information, and example code, the manpage for that function should be consulted.

Table 4-10   Endian Services

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSIO_BIG_ENDIAN()</td>
<td>Returns true (1) if the local bus is big endian</td>
</tr>
<tr>
<td>WSIO_LITTLE_ENDIAN()</td>
<td>Returns true (1) if the local bus is little endian</td>
</tr>
</tbody>
</table>

Table 4-11   DMA Services

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wsio_allocate_dma_handle()</td>
<td>Obtain a handle used to setup DMA</td>
</tr>
<tr>
<td>wsio_free_dma_handle()</td>
<td>Releases a DMA handle</td>
</tr>
<tr>
<td>wsio_init_map_context()</td>
<td>Initializes the context used for DMA mapping</td>
</tr>
<tr>
<td>wsio_allocate_shared_mem()</td>
<td>Setup an I/O virtually contiguous DMA buffer</td>
</tr>
<tr>
<td>wsio_free_shared_mem()</td>
<td>Release an I/O virtually contiguous DMA buffer</td>
</tr>
<tr>
<td>wsio_flush_shared_mem()</td>
<td>Flush an I/O virtually contiguous DMA buffer</td>
</tr>
<tr>
<td>wsio_map_dma_buffer()</td>
<td>Maps an existing memory object for packet DMA</td>
</tr>
<tr>
<td>wsio_map_dma_buffer()</td>
<td>Maps an existing memory object for packet DMA</td>
</tr>
<tr>
<td>wsio_remap_dma_buffer()</td>
<td>Maps pre-allocated IOVAs to new host ranges</td>
</tr>
<tr>
<td>wsio_unmap_dma_buffer()</td>
<td>Removes a DMA packet mapping</td>
</tr>
<tr>
<td>wsio_iova_to_phys()</td>
<td>Translate an I/O virtual address to a host virtual address</td>
</tr>
<tr>
<td>wsio_set_dma_callback()</td>
<td>Sets the callback function and argument for DMA</td>
</tr>
<tr>
<td>wsio_dma_pass_thru()</td>
<td>Calls a pass-through function that might not otherwise be accessible</td>
</tr>
<tr>
<td>wsio_dma_set_service_attributes()</td>
<td>Associates DMA hints with a device</td>
</tr>
<tr>
<td>wsio_set_dma_attributes()</td>
<td>Associates DMA hints with a DMA handle</td>
</tr>
</tbody>
</table>
Typically a driver would allocate one or more DMA handles in its driver `init` routine. It would then use the services `wsio_dma_set_service_attributes()` or `wsio_set_dma_attributes()` to specify attributes of the type of DMA it will use the object for. A driver may configure one handle for large packet DMA and another for continuous DMA involving small buffers. Later, when setting up a DMA the driver would pass the handle to one of the DMA mapping routines.

### Interrupt Services

These services are used by device drivers to obtain and set up interrupts. These particular services allow a device driver to allocate multiple interrupts, and associate separate interrupt service routines with each. The functions provided are listed and briefly described in *Table 4-12, “Interrupt Services.”* For further information on each routine, including parameter information, return code information, and example code, the manpage for that function should be consulted.

**Table 4-12 Interrupt Services**

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wsio_intr_alloc()</code></td>
<td>Allocate a WSIO interrupt object</td>
</tr>
<tr>
<td><code>wsio_intr_free()</code></td>
<td>Free a WSIO interrupt object</td>
</tr>
<tr>
<td><code>wsio_intr_active()</code></td>
<td>Enable a WSIO interrupt object</td>
</tr>
<tr>
<td><code>wsio_intr_deactivate()</code></td>
<td>Disable a WSIO interrupt object. This call blocks</td>
</tr>
<tr>
<td><code>wsio_intr_deactivate_nowait()</code></td>
<td>Disable a WSIO interrupt object with callback. Non-block</td>
</tr>
<tr>
<td><code>wsio_intr_set_cpu_spec()</code></td>
<td>Set up transaction based interrupts</td>
</tr>
<tr>
<td><code>wsio_intr_set_irq_line()</code></td>
<td>Set up line based interrupts</td>
</tr>
<tr>
<td><code>wsio_intr_get_assigned_cpu()</code></td>
<td>Get the assigned CPU</td>
</tr>
<tr>
<td><code>wsio_intr_get_irq_line()</code></td>
<td>Get the IRQ line</td>
</tr>
<tr>
<td><code>wsio_intr_get_txn_info()</code></td>
<td>Get the transaction address and data values</td>
</tr>
</tbody>
</table>

### Description Service

This service is used by device drivers to set their I/O tree description. This service is necessary because in some cases, a meaningful description cannot be figured out automatically. The one function provided for this service is listed and described in *Table 4-13, “Description Service.”* For further information on parameters, return codes and example code, check the manpage..

**Table 4-13 Description Service**

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wsio_set_description()</code></td>
<td>Sets the I/O tree node description for this driver</td>
</tr>
</tbody>
</table>

A driver would call this in its `init` routine.
Ordered Interrupts Service

This service is used by device drivers to determine whether interrupts are ordered with respect to DMA transactions. This tells device drivers whether it is necessary to explicitly perform a sync to ensure that DMA transactions have completed. The one function provided for this service is listed and described in Table 4-14, “Ordered Interrupts Service.” For further information on parameters, return codes and example code, check the manpage.

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSIO_ORDERED_INTERRUPTS()</td>
<td>Indicates whether interrupts are ordered with respect to DMA transactions</td>
</tr>
</tbody>
</table>

I/O Synchronization Service

This service is used by device drivers to explicitly synchronize the CPU and the I/O device’s views of memory. If such a synchronization is not necessary, this function call will avoid doing it. The one function provided for this service is listed and described in Table 4-15, “I/O Synchronization Service.” For further information on parameters, return codes and example code, check the manpage.

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wsio_io_sync()</td>
<td>Performs a sync of shared memory if necessary</td>
</tr>
</tbody>
</table>

System Attribute Services

These services are used by device drivers to obtain information about specific system features. Currently, only information about cacheline size, default page size, and the number of CPUs in a system is available. The functions provided for these services are listed and described in Table 4-16, “System Attribute Services.” For further information on parameters, return codes and example code, check the manpage.

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wsio_get_system_params()</td>
<td>Obtains information about the system</td>
</tr>
<tr>
<td>wsio_get_processor_count()</td>
<td>Number of processors on the system</td>
</tr>
<tr>
<td>wsio_get_active_processor_count()</td>
<td>Active number of processors on the system</td>
</tr>
<tr>
<td>wsio_get_cpu_name()</td>
<td>Name (Address) of a CPU</td>
</tr>
</tbody>
</table>
Driver Event Handling Services

These services allow drivers to register an event handler and event mask with the WSIO. When an event occurs and the event mask indicates that the driver is interested in that type of event it's handler is called. Events include PCI OLAR actions such as suspend, resume or remove.

Event handlers are registered on a per driver basis. This is usually done in the driver's install routine. The mask, however, is registered by the driver on a per instance basis. This is done in the driver's init routine after claiming an I/O card. Using this method, drivers can register a single event handler, but specify what events it can handle on a per-instance basis.

When an event impacts an I/O device the WSIO first checks to see if there is an event handler associated with the device. If there is, it then checks the event mask to see if the handler will respond to that type of event. If it does, it then calls the handler.

All driver handlers must have the following caller syntax:

```c
void my_drv_handler(wsio_generic_event_t * event_info_ptr);
```

For more information on the driver handler and the structure `wsio_generic_event_t`, see the manpages, or refer to Table 4-17, “Driver Event Handling Services.”

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wsio_install_drv_event_handler</td>
<td>Register a driver's event handler</td>
</tr>
<tr>
<td>wsio_reg_drv_capability_mask</td>
<td>Register an event capabilities mask</td>
</tr>
</tbody>
</table>

Memory Allocation Services

The first two services, `wsio_alloc_mem_handle()` and `wsio_free_mem_handle()` are used to create and destroy memory allocation handles. The driver can use these services to create and destroy handles that specify the type of memory it wants to allocate.

The second two services, `wsio_alloc_mem()` and `wsio_free_mem()` are used to allocate and free memory using the memory allocation handle created by the service `wsio_alloc_mem_handle()`.

Typically drivers would call `wsio_alloc_mem_handle()` in their init routines to specify the type of memory they want to allocate. The service will return a handle that the driver would then pass into the services `wsio_alloc_mem()` and `wsio_free_mem()` when allocating and free buffers.

Drivers can allocate multiple memory handles for the different types of memory buffers they use. For example, a driver could specify a memory handle that only allocates physical memory below four Gbyte. It could then specify another handle for allocating memory that is always physically contiguous.

For more specific information on how to call these services and the parameters passed in, see the manpage for each in *HP-UX 11i v1 Driver Development Reference*.

For additional information, refer to Table 4-18, “Memory Allocation Services.”
### Memory Allocation Services

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wsio_alloc_mem_handle()</code></td>
<td>This service is used to specify the type of memory a driver will allocate. It returns a memory handle.</td>
</tr>
<tr>
<td><code>wsio_free_mem_handle()</code></td>
<td>This service frees a memory handle allocated by <code>wsio_alloc_mem_handle()</code></td>
</tr>
<tr>
<td><code>wsio_alloc_mem()</code></td>
<td>This service is called to allocate memory.</td>
</tr>
<tr>
<td><code>wsio_free_mem()</code></td>
<td>This service is called to free memory.</td>
</tr>
</tbody>
</table>
Support for PCI OLAR

PCI OLAR is available in the 11i version of HP-UX for those platforms that have hardware support. Drivers that want to take advantage of this feature must provide an event handler and register an event mask for each interface card they claim. The handler is called by the WSIO CDIO to take action when a PCI OLAR event occurs that impacts a card owned by the driver. It is only called if the driver specified an interest in the event via the registered event mask for the affected card.

For 11i the type of events that a driver can register an interest in are:

- WSIO_EVENT_SUSPEND
- WSIO_EVENT_RESUME

Drivers register their handlers with the WSIO in the driver install routines. The event mask is registered later in the driver attach routine each time it claims an I/O card.

When a driver handler is called with an event of type WSIO_EVENT_SUSPEND it should suspend the activities of the card, including DMA and interrupts. It does not have to give up any long term resources like MMIO space or memory buffers for long term mappings.

When the driver handler is called with an event type of WSIO_EVENT_RESUME it should resume the activities of the specified driver instance.

For 11i HP-UX supports PCI card replacement. This means that a driver instance can be SUSPENDED, the card replaced, and the driver instance RESUMED.

Registering a Driver Handler

A driver registers its event handler with the WSIO CDIO in its install routine by calling the WSIO service wsio_install_drv_event_handler(). The calling interface for this service is as follows:

```c
int wsio_install_drv_event_handler __attribute__((always_inline))
    ((wsio_drv_info_t * info_ptr, wsio_drv_event_handler_t handler));
```

The first parameter is a pointer to the driver's wsio_drv_info_t structure and the second parameter is a pointer to the driver handler. The driver handler should have the following calling interface:

```c
void (*drv_handler) (wsio_generic_event_t *);
```

It takes a single parameter which is a pointer to a structure of the type wsio_generic_event_t. This structure is defined in the header file wsio.h as follows:

```c
typedef struct wsio_generic_event {
    wsio_event_t event; /*suspend, resume, and so on*/
    /* Event_id is a tag to identify an instance of an event.
    * Driver just passes it back in the callback */
    wsio_event_id_t event_id;
    struct isc_table_type * isc;
    generic_complete_callback_t wsio_completion_cb;
    void *arg;
} wsio_generic_event_t;
```

The first field specifies the type of event that the driver handler is being called for. For PCI OLAR the following types of wsio_event_t are or will be defined:

- WSIO_EVENT_SUSPEND
- WSIO_EVENT_RESUME
The second argument, \texttt{event\_id}, is used to identify a specific event. The third argument is the \texttt{isc} structure of the card. The fourth argument, \texttt{wsio\_completion\_cb}, is a callback function that the driver must call when it completes the event to report status. The driver must report the status of an event asynchronously via this callback mechanism. The calling interface for the callback function is:

\begin{verbatim}
int (* wsio\_completion\_cb)( struct isc\_table\_type * isc,
        wsio\_event\_id\_t event\_id,
        void * status );
\end{verbatim}

The first parameter is the \texttt{isc} of the driver instance, the second is the value of the \texttt{event\_id} field passed into the driver handler and the third is the status of the action. If the handler successfully completed the event it should set the status to \texttt{WSIO\_OK}, otherwise it should set it to either \texttt{WSIO\_UNSUPPORTED\_EVENT} or \texttt{WSIO\_ERROR}. The former is returned if the event is not supported by the driver, the latter for a failure to complete the event successfully. The following example shows a driver install routine:

\begin{verbatim}
#include <wsio/wsio.h>
static wsio\_drv\_info\_t my\_drv\_info {
    &my\_drv\_info;
    &my\_drv\_ops,
    &my\_drv\_data,
    WSIO\_DRV\_CURRENT\_VERSION,
}
my\_driver\_install()
{
    wsio\_install\_driver(&my\_drv\_info);
    if ( (wsio\_install\_drv\_event\_handler(&my\_drv\_info,
        my\_drv\_handler))
            != WSIO\_OK)
    {
        wsio\_uninstall\_driver(&my\_drv\_info);
        return(0);
    }
    /**********************************************************************
    * Let’s go ahead and link the driver’s attach
    * routine into the global PCI attach list
    **********************************************************************
    my\_drv\_saved\_attach = pci\_attach;
    pci\_attach = my\_driver\_attach;
    return(1);
}
\end{verbatim}

### Registering a Driver Event Mask

Drivers that register handlers must register an event’s capabilities mask for each interface card they claim. This mask indicates what events their handlers are interested in. The driver registers the capabilities mask by calling the WSIO service \texttt{wsio\_reg\_drv\_capabilities\_mask()}. The calling interface for the service is:

\begin{verbatim}
int wsio\_reg\_drv\_capability\_mask __
    ((struct isc\_table\_type * isc,
        wsio\_event\_mask\_t mask));
\end{verbatim}

The first parameter is the driver \texttt{isc} structure. The second parameter is the mask which is formed by ORing one or more \texttt{wsio\_event\_t} values. For PCI OLAR, drivers must pass in a mask as follows:

\texttt{(WSIO\_EVENT\_SUSPEND | WSIO\_EVENT\_RESUME)}
Setting Event Timeout Values

The WSIO CDIO sets a timeout when it calls the driver handler. If the handler does not reply via the callback function before the timeout pops, the WSIO assumes an error of some sort has occurred, and reports this. Drivers can change the value of the timeout by calling the WSIO service `wsio_set_parm()` in their attach routine. They can also query the default value for the timeout by calling the WSIO service `wsio_get_parm()`. They can get and set the values for each type of event, `WSIO_EVENT_SUSPEND` or `WSIO_EVENT_RESUME`. The calling interfaces for these WSIO services are:

```c
int wsio_get_parm( struct isc_table_type * isc,
                   wsio_parm_t I_parm,
                   void **I_value);

int wsio_set_parm( struct isc_table_type *I_isc,
                   wsio_parm_t I_parm,
                   void *I_value);
```

The first parameter is a pointer to an `isc` structure of the driver instance. The second is of type `wsio_parm_t` which indicates what value is being set or queried. Valid values for this parameter are:

- `WSIO_HW_SUSPEND_TIMEOUT` — Hardware Suspend Timeout parameter
- `WSIO_HW_RESUME_TIMEOUT` — Hardware Resume Timeout parameter.

The final parameter is the value to be set or returned. For `wsio_get_parm()` the current value is returned in this parameter. For `wsio_set_parm()` the caller passes in the current value.

See the manpages in `HP-UX 11i v1 Driver Development Reference` for more details on calling these services.

The following example shows a driver attach routine setting the capabilities mask and timeout values:

```c
int my_driver_attach(uint32_t id, struct isc_table_type * isc)
{
    int ret;
    wsio_event_mask_t my_task;
    if (id == MY_CARD_ID)
    {
        /* do any other driver/instance processing ....
         **************************************************
         * Let's claim the card and register an event's
         * capabilities
         **************************************************/
        isc_claim(isc, &my_drv_info);

        /****************************************************/
        * The return value should be either WSIO OK which
        * indicates that the mask was registered or WSIO_HA_NA
        * which indicates that the platform does not support
        * PCI OLAR. If it's not one or the other we've hit
        * some other unknown condition, so let's just bail.
        *******************************************************/
        my_mask = WSIO_EVENT_SUSPEND | WSIO_EVENT_RESUME;
        ret = wsio_reg_drv_capabilities_mask(isc, my_mask);
        if( (ret != WSIO_OK) && (ret != WSIO_HA_NA) )
        {
            /* free any allocated resources for ....
             * this card instance.           ....
             */
            isc_unclaim(isc, &my_drv_info);
        }
        else
        {
            /****************************************************/
        }
    }
```
* Let’s modify the timeout values for
  * WSIO_EVENT_SUSPEND and WSIO_EVENT_RESUME.
  * The default values are 1 second. Timeout
  * values are given in microseconds.

```
wsio_set_parm(isc, WSIO_HW_SUSPEND_TIMEOUT, 2000000);
wsio_set_parm(isc, WSIO_HW_RESUME_TIMEOUT, 1500000);
```

**Driver Event Handler**

This function has one input parameter, a pointer to a data structure. The content of fields in the structure is dependent on the event. Most error handlings in the example are left out for simplicity. A driver will return the status of its operation in the callback function: `WSIO_OK` for success, `WSIO_UNSUPPORTED_EVENT` for unknown request, and `WSIO_ERROR` for failure (in this case, a driver should log messages to `dmesg` buffer).

```
#include <wsio/wsio.h>

/* Following is a sample of a global data structure to
temporarily save the WSIO’s isc, hint, event_id, and
callback function. The algorithm for a suspend event is
* like this: 1) The event handler is called, 2) The
* handler stores away the isc, hint, event_id, and
* callback function in this global data structure
* 3) Call Ktimeout() to schedule another function,
* my_suspend(), to really do the suspension. 4) Event
* handler returns right away. This is just an example.
* This structure can be pointed to by one of the fields
* in the isc pointer. The mechanism to implement the
* handler is driver dependent. */

typedef struct my_suspend_information {
  struct isc_table_type *isc;
  wsio_event_id_t event_id;
  wsio_suspend_info_t hint;
  int (*complete_cb)(struct isc_table_type *isc;
                    wsio_event_id_t event_id, void *status);
} my_suspend_information_t;
my_suspend_information_t *my_suspend_info;

/* This my_suspend() function is invoked by Ktimeout().
* Its input parameter is a pointer to
* my_suspend_information_t. The event handler scheduless
* the Ktimeout() call for this function */

void my_suspend(my_suspend_information_t *my_suspend_op) {
  struct isc_table_type *isc=my_suspend_cp->isc;
  wsio_suspend_info_t hint = my_suspend_cp->hint;

  /* Tell the card to stop interrupt */
  * Do whatever is needed */

  If error, my_suspend_cp->complete_cb(isc,
    my_suspend_cp->event_id, (void *)WSIO_ERROR)

  /* The last parameter of complete_cb() is void *status. WSIO
  * will typecast it according to what event is being handled.
  * For suspend, resume, and remove, drivers should return a
* define, such as WSIO_OK, WSIO_UNSUPPORTED_EVENT, or
  * WSIO_ERROR. It can be typecasted as (void *) to avoid
  * compiler warning */

my_suspend_cp->complete_cb(isc, my_suspend_cp->event_id,
  (void *)WSIO_OK)
}

/* The following event handler first stores away the
 * event_id, isc, call back function, and hint into a global,
 * my_suspend info. Then it calls Ktimeout() with a timeout
 * value of 0 to schedule my_suspend(0 to run. The function
 * return right away. A driver can decide on a timeout value
 * depending on whether it wants to accomodate any in-progress
 * activities. This is just an example. A driver can design
 * the best approach to fit its requirement. The event ID is
 * tag that WSIO generates to prepare for future use (match up
 * request with a reply); driver should return in the
 * completion callback. */

void my_handler(wsio_generic_event * handler_arg) {
  switch (handler_arg->event) {
    case WSIO_EVENT_SUSPEND:
      /* Store away the values in signal my_suspend_info
       * Malloc a structure pointed to by my_suspend_info; */
      my_suspend_info->event_id = handler_arg->event_id;
      my_suspend_info->isc = handler_arg->isc;
      my_suspend_info->complete_cb = handler_arg->wsio_completion_cb;
      my_suspend_info->hint = handler_arg->arg;

      /* Schedule the call to my_suspend() to run immediately */
      Ktimeout(my_suspend, my_suspend_info, 0, NULL);
      break;

    case WSIO_EVENT_RESUME:
      /* Saving of my_resume_info not shown */
      Ktimeout(my_resume, my_resume_info, 0, NULL);
      break;

    case default:
      handler_arg->wsio_completion_cb(handler_arg->isc,
          handler_arg->event_id,
          (void *) WSIO_UNSUPPORTED_EVENT);
      break;
  }
  return;
}
5 Installing Your Driver

This chapter describes how to build your driver into the kernel. For information about demand loadable module drivers see Chapter 16, “Writing a DLKM Driver.” For other drivers, do the following:

1. Compile the driver; if building a driver for both 32-bit and 64-bit kernels, build separate libraries for each kernel flavor.
2. Add your driver's name to the system file.
3. Create a master file for your driver in the /usr/conf/master.d directory. Do not update files shipped with the system.
4. Run the config command to build your driver into the kernel.
5. Reboot the system.
6. Use the ioscan command to associate your driver with your hardware.
7. Use the mknod command to create a device-special file for your driver.
8. Use the ioscan command to test the configuration.

The following sections give detailed descriptions of the previous steps.

In these procedures, we assume that your driver’s name is mydriver.
Step 1: Compile Your Driver

Before adding your driver to the HP-UX kernel, you must compile your driver to generate a new object file. To do this, change to the directory that contains your source files. Use a subdirectory of `/usr/conf`, such as `/usr/conf/mydriver`.

Next, compile your driver using the HP ANSI C compiler. The HP product numbers for the compiler are:

- B3901BA — HP 9000 servers
- B3899BA — HP 9000 workstations
- B6237AA — Documentation set

NOTE  You may use the bundled compiler `/usr/ccs/bin/cc` to build the kernel from `/stand/build/config.mk`. Use the HP ANSI C compiler to build the driver into the kernel.

Compiling a Driver

The easiest method is to use the compiler options found in the makefile `/stand/build/config.mk`. Copy this file to Makefile under the driver subdirectory. An example of `mydriver` is:

```
  cp /stand/build/config.mk /usr/conf/mydriver/Makefile
```

Make the following modifications to the Makefile to build the driver:

1. In the Makefile, change the target `all` to your driver object file. So for `mydriver` example, change the following line in the Makefile.
   
   from:  all: $(DIR)/$(HPUX)
   to:    all: mydriver.o

2. In the Makefile, change `CONF` with your driver name. So for `mydriver` example, change the Makefile line.

   from: CONF=/stand/build/conf
   to:    CONF=mydriver

3. If needed, change the C compiler location. For example, the ANSI C compiler is located at `/opt/ansic/bin`. For the `mydriver` example, change the Makefile line.

   from: CC= $(CKRN)/cc
   to:    CC= /opt/ansic/bin/cc

4. When the ANSI C compiler is used to build your driver, also include additional flags `-Ae` to the compiler options list. Refer to the `cc (1)` manpage for more details. For the `mydriver` example, change the Makefile line.

   from: CFLAGS= -w $(COPTS) $(K_CCOPTS)
   to:    CFLAGS= -Ae -w $(COPTS) $(K_CCOPTS)

5. If building a 32-bit driver, make sure that the destination architecture flag is correctly specified as `+DA1.1`. The Makefile should have the following line:

   DAFLAG=+DA1.1
6. If building a 64-bit driver, be sure the destination architecture flag is correctly specified as +DA2.0W. The Makefile should have the following line:

DAFLAG=+DA2.0

Build the driver by typing make at the command prompt. Alternatively, use the compiler options directly at the command line as shown:

If using an ANSI C compiler and building a 32-bit driver:

```
/opt/ansic/bin/cc -I/usr/conf -I -c -o mydriver.o \-Ae -w -U__hp9000s700 -D__HIGHC__ -D__STDC_EXT__ \-D_XPG4_EXTENDED -D_HPUX_SOURCE -D_UNSUPPORTED \-D_hp9000s800 -D_KERNEL -DKERNEL 
+ES1.Xindirect_calls \
-Wp,-H300000 +XixdU +Hx0 +R500 -Wl,-a,archive +DA1.1 \+DS2.0 +ESfc +ESssf \mydriver.c
```

If using an ANSI C compiler and building a 64-bit driver:

```
/opt/ansic/bin/cc -I/usr/conf -I -c -o mydriver.o \-Ae -w -U__hp9000s700 -D__HIGHC__ -D__STDC_EXT__ \-D_XPG4_EXTENDED -D_HPUX_SOURCE -D_UNSUPPORTED \-D_hp9000s800 -D_KERNEL -DKERNEL 
+ES1.Xindirect_calls \
-Wp,-H300000 +XixdU +Hx0 +R500 -Wl,-a,archive +DA2.0 \+DS2.0 +ESfc +ESssf \mydriver.c
```

When the compilation succeeds, it produces a file named mydriver.o in the current directory.
Step 2: Create a Master File

Create a master file for your driver in the `/usr/conf/master.d` directory.

The master files in `/usr/conf/master.d` provide configuration information about drivers. The HP master files group drivers along functionality lines, there is a master file for core drivers and subsystems, a master file for basic networking, and so on. The format of these files is defined in `master (4)` in the *HP-UX Reference*. The `/usr/sbin/config` command generates a comprehensive master file from the individual master files in `/usr/conf/master.d`.

Creating a Master File for Your Driver

Create a new master file in `/usr/conf/master.d` that contains the information specific to the driver.

The file can have any name, but we recommend naming the master file in a way that identifies it with the driver, so customers will recognize the connection. In our example, we call it `mydriver`. The master file's name must be unique within `/usr/conf/master.d`. Perhaps the developer can identify it with his/her company name to avoid conflicts with other master file names from other suppliers.

**CAUTION** Do not add your driver's information to one of the existing master files supplied by Hewlett-Packard in `/usr/conf/master.d`. These files will be overwritten when you or your customers update the system with the next HP-UX release.

The name of the master files in `/usr/conf/master.d` must conform to the following conventions:

- File names must not contain the characters period (.), tilde (~), or pound sign (#).
- File names must not contain the word core.
- File names must not contain the word RMTBRANCH.
- Files should have names no longer than 14 characters to allow your driver to be used on short file name systems.

Editing the Master File

Make the following entries in the master file:

- Each driver needs an entry in the `$DRIVER_INSTALL` table in a master file.

Here is an example of a portion of a `$DRIVER_INSTALL` table:

```
$DRIVER_INSTALL
***************
* Driver * Block major * Char major * Required for minimal system *
asio0 -1 1
cpd -1 -1
kepd -1 227
dev_config -1 69
klog -1 189
nn -1 0
mm -1 3
pty0 -1 16
...
```
The following paragraphs describe the fields in the $DRIVER_INSTALL table and what to put in them for the driver. The fields are separated by spaces:

— The first field, Driver, contains the driver's name, for example, mydriver.

— The second and third fields, Block major and Char major, specify the driver's block and character major numbers. Specify -1 in both fields so that the kernel will assign the driver a major number dynamically. (These fields are informational; the values in the drv_info_t structure take precedence.)

— The fourth field specifies whether the driver is needed to run a minimal system. Enter 1 if true; otherwise, enter 0 or leave it blank.

❏ If your driver has dependencies on other drivers, list these other drivers in the $DRIVER_DEPENDENCY table in the master file. The format is:

```
driver_name otherdriver1 otherdriver2 ...
```

❏ Use the $DRIVER_LIBRARY table to specify libraries that contain functions the driver uses. This should be set only if the entries listed here are in /usr/conf/lib directory. Look at Step 4 for additional details on different options for building a kernel containing your driver. The format is:

```
driver_name lib1 lib2 ...
```

Use $LIBRARY table to specify whether a library has to be always included in the kernel or not.
If you list the driver library under $DRIVER_LIBRARY, include the driver library entry under this. The format is:

```
driver_library_name   <0 | 1>
```

wherein, 0 - if the library is optional

1 - if the library should "always" be included

The driver may require entries in other master file tables.

See master (4) in the HP-UX Reference for details on the format of master file tables. It may be useful to examine an existing master file, such as core-hpux in /usr/conf/master.d.
Example Master File

An example master file for mydriver is described in this section. The master file must be located under /usr/conf/master.d with a name of mydriver. Refer to the beginning of this section for file name restrictions.

* * * $Revision: 1.1.106.2 $ * The following devices are those that can be specified in the * system description file. The name specified must agree with * the name shown, or with an alias.
  * name handle type mask block char
   *
$DEVICE $$$
$CDIO
  * The following entries form the cdio table.
  * field 1: cdio name
  * field 2: required, optional

  $$$
$DRIVER_INSTALL *************************************************************** Driver install table
  * This table contains the name of drivers which have converged * I/O header structures and install entry points.
Drivers in * this table should not be defined in the driver table above.
  * Note : as of 10.30, lan0 and lan1 are obsolete ***************************************************************
Driver Block major Char major
mydriver -1 -1 $$$
$ALIAS
  * The following entries form the alias table.
  * field 1: product # field 2: driver name
  * $$$
$TUNABLE
  * The following entries form the tunable parameter table. Any
  * of these values can be overwritten in the dfile. However,
  * before changing any value, know the ramifications of your
  * change.
  * $$$
$DRIVER_DEPENDENCY
  * Driver dependency table, if one driver is present, what
  * other drivers should also be in the dfile.
  *
  * Driver name <dependency> <dependency> ...... $$$
$DRIVER_LIBRARY
  * The driver/library table. This table defines which libraries * a given driver depends on. If the driver is included in the
  * dfile, then the libraries that driver depends on will be
  * included on the ld(1) command line. Only optional libraries
  * need to be specified in this table, (but required ones can be * included, as well).
  *
  * Driver handle <libraries>
  *
  * subsystems first $$$
$LIBRARY
* 
* The library table. Each element in the library table
* describes one unique library. The flag member is a boolean
* value, it is initialized to 1 if the library should *always* * be included on the ld(1) command line, or 0 if the
library is * optional (i.e. it is only included when one or more drivers
* require it). The order of the library table determines the
* order of the libraries on the ld(1) command line, (i.e.
* defines an implicit load order). New libraries must be added * to this table.
* 
* Note: libhp-ux.a must be the last entry, do not place * anything after it.
* 
* Library <required>
* 
 $$$

$SUBSYSTEMS_DEFINE
* Subsystem #define table. For any subsystem that is found in
* the dfile and appears below, a #define is generated in the
* conf.c file. This is used to include any space defined in
* space.h.

$$$ 

$DRIVER_PRODUCT
* 
* Driver product table. The following generates a table of
* supported products for a given interface card. The input is a * driver handle followed by a list of
* product/vendor id’s. What * is produced is a table of these product/vendor id’s to be
* used by the interface driver for further qualification during
* booting and the setting of root device location.
* 
$$$
Step 3: Modify the System File

Make an entry for the driver in the system file. The /bin/config uses the system file to build the link/install table. The contents of that table are traversed at configuration time to indicate which drivers will be preinstalled into the kernel.

The system file (the default is /stand/system) is the description file for a particular system configuration. See config (1M) to get more information about the format of the system file.

1. Change directory to the build environment (/stand/build). Then execute the system-preparation script, system_prep, which extracts the system file from the current kernel:
   
   cd /stand/build
   /usr/lbin/sysadm/system_prep -s system

   The system_prep script writes a system file in the current directory. (It creates /stand/build/system.) Alternatively, copy the current system file into the build directory.
   
   cp /stand/system ./system

2. Edit the /stand/build/system file to add the name of the driver mydriver to the first section Drivers/Subsystems of the file on a separate line. See config (1M) for details on the format of the system file.
Step 4: Build a Kernel Containing Your Driver

Having created a master file for the driver and having edited the system file to include the driver. Follow these steps to build the kernel and prepare it for testing. The current directory is still /stand/build. There are two methods for building the kernel.

1. Using mk_kernel

   Use the following command to build the kernel containing the driver.

   ```bash
   mk_kernel -s system -o vmunix
   ```

   The `mk_kernel` builds an executable file which can be used as a bootable kernel and kernel modules if any are configured. If the build succeeds, the newly built kernel is called `vmunix_test`. The file and directory are placed in the build directory, as defined.

   If the path used to designate the system file is `/stand/system`, the build directory is `/stand/build`. If another path is used to designate the system file, the build directory is the current working directory.

   If `-o /stand/vmunix` is specified, the target kernel file and kernel function set directory are not overwritten. The new kernel file and the kernel function set directory are moved to the default path as the system shuts down or starts up. The previous versions of the file and directory are renamed `/stand/vmunix.prev` and `/stand/dlkm.vmunix.prev`. Until the system reboots, the new kernel file and the directory must be kept as `vmunix_test` and `dlkm.vmunix_test`, respectively.

   If the `-o` option is specified with other than `/stand/vmunix`, the kernel file and kernel function set directory are created or updated immediately. In case the administrator needs to place these targets to the system default path, the `kmupdate` command must be used to trigger the replacement. Manually replacing the default kernel (`/stand/vmunix`) or any file under the kernel function set directory (`/stand/dlkm`) must be avoided.

2. Using config and make

   Use the following command to generate the files used to create a new kernel containing the driver.

   ```bash
   /usr/sbin/config -s system
   ```

   This command has two output files, both placed in `/stand/build`, `config.mk` and `conf.c` (i.e., both are placed in the current directory).

   - The `config.mk` makefile generates a kernel. The two macro definitions, `OFILES` and `XOPBJS=`, are in `config.mk`.
   - The `conf.c` file links the drivers specified in your system file with the kernel.

     (See `config` (1M) in the *HP-UX Reference.*

     Include the object file in the new kernel. There are four methods. The first and second are suitable for testing; the third and fourth are preferable for installing the production version:

     - **Alternative 1:** Use the XOBJJS variable.

       Build the new kernel with the following command:

       ```bash
       make -f config.mk XOBJJS=/usr/conf/mydriver/mydriver.o
       ```

       Wait a moment while `config.mk` links the driver with the rest of HP-UX, and builds a file named `vmunix_test` in the current directory.
Installing Your Driver

Step 4: Build a Kernel Containing Your Driver

- **Alternative 2: Use the OFILES variable.**

  Modify `config.mk` so that the definition of OFILES includes an object file. Look in `config.mk` for lines that look something like:

  ```
  $(ROOT)/libcdfs.a $(ROOT)/libhp-ux.a
  OFILES=/usr/conf/mydriver/mydriver.o
  ```

  Add a reference to your driver by adding it to the OFILES variable as follows:

  ```
  $(ROOT)/libcdfs.a $(ROOT)/libhp-ux.a
  OFILES = mydriver.o
  ```

  Build the new kernel with the following command:

  ```
  make -f config.mk
  ```

  Wait a moment while `config.mk` links the driver with the rest of HP-UX and builds a file named `vmunix_test` in the current directory.

- **Alternative 3: Place Your Object File in a Library**

  Add the object file to the `/usr/conf/lib/libusrdrv.a` library. If the library exists, the `LIBUSRDRV` variable automatically links the kernel with the members of the library. Create the library, if necessary, and add your driver to it.

  Use this command:

  ```
  ar -r /usr/conf/lib/libusrdrv.a mydriver.o
  ```

  Build the new kernel with the following command:

  ```
  make -f config.mk
  ```

  Wait a moment while `config.mk` links the driver with the rest of HP-UX and builds a file named `vmunix_test` in the current directory.

- **Alternative 4: Place The Driver Library Under `/usr/conf/lib`**

  For example, in the case of `mydriver`.

  Wait a moment while `config.mk` links the driver with the rest of HP-UX and builds a file named `vmunix` in the current directory.

  ```
  cd to /usr/conf/mydriver/
  Build the driver library with ar -r libmydriver.a mydriver.o
  Copy the driver library to /usr/conf/lib/ cp libmydriver.a /usr/conf/lib
  cd back to /stand/build.
  ```

  Modify the driver master file to include the driver library. From the example master file shown, add the driver library entry under `$DRIVER_LIBRARY_` and `$LIBRARY` tables. Since `mydriver` is optional in the kernel, we use a value of 0 for `libmydriver.a` under the #LIBRARY section. The modified parts of the master file look like this:

  ```
  ....
  $DRIVER_LIBRARY
  *
  * The driver/library table. This table defines which
  * libraries a given driver depends on. If the driver is
  * included in the dfile, the libraries that driver
  * depends on will be included on the ld(1) command
  * line. Only optional libraries *need* to be specified in * this table,(but required ones can
  be included, as
  ```
Installing Your Driver

Chapter 5

Step 4: Build a Kernel Containing Your Driver

* well).
* Driver handle <libraries>
* subsystems first
mydriver libmydriver.a

$$ $LIBRARY
*
* The library table. Each element in the library table
* describes one unique library. The flag member is a
* boolean value, it is initialized to 1 if the library
* should *always* be included on* the ld(1) command line, * or 0 if the library is optional
(i.e. it is only
* included when one or more drivers require it). The
* order of the library table determines the order of the * libraries on the ld(1) command line,
(i.e. defines an
* implicit load order). New libraries must be added to
* this table.
* Note: libhp-ux.a must be the last entry, do not place anything after it.
*
* Library <required>
*
libmydriver.a 0

$$

....

After the master file is changed, use the following commands to generate the files used to create a
new kernel containing the driver.
/usr/sbin/config -s system

This command has two output files, config.mk and conf.c, both placed in /stand/build directory
(which is your current directory). Run the following command to build the kernel containing the
driver.
make -f config.mk

Save the current system file and kernel by renaming them. If anything goes wrong, use a bootable kernel. For
example,

mv /stand/system /stand/system.prev
mv /stand/vmunix /stand/vmunix.prev

Move the new system file and new kernel into place, ready to be used upon rebooting the system.

mv /stand/build/system /stand/system
mv /stand/build/vmunix_test /stand/vmunix

Reboot the system with the new kernel. Enter the following command:
exec reboot

For more information about building and installing the kernel, see the HP-UX System Administration Tasks.
Step 5: Create a Device-Special File

Once the system has rebooted successfully, to create a device-special file for the device in the /dev directory. To do this, determine the device’s major and minor numbers and use the mknod command.

1. Use the \texttt{lsdev} command to identify the major number assigned to the device driver. \texttt{lsdev} lists all device drivers configured into the kernel, and their block- and character-major numbers.

   By adapting an example from the \texttt{lsdev (1M)} manpage, one can extract the numbers from the display:

   \begin{verbatim}
   lsdev -h -d mydriver | awk '{print $1}'
   \end{verbatim}
   character major number

   \begin{verbatim}
   lsdev -h -d mydriver | awk '{print $2}'
   \end{verbatim}
   block major number

2. Invoke the \texttt{/usr/sbin/ioscan} command with the \texttt{-f} or \texttt{-k} option, and note the hardware path for which \texttt{ioscan} reports an unknown class of device.

3. Construct a minor number for the device by using the bit assignments for the driver. See “Major and Minor Numbers” in Chapter 2, for more information about bit assignments and dev_t.

4. Use the \texttt{mknod} command to create the device-special files for the device. See \texttt{mknod (2)} in the HP-UX Reference. For information on file-naming conventions, see Configuring HP-UX for Peripherals.

   In the example below, \texttt{mydriver} was (dynamically) assigned the block- and character-major numbers 65 and 234, respectively. Its minor number, 0x026000, is constructed like that of \texttt{instr0} (bits 8 through 15 encode 2 as the instance of the interface card, and bits 16 through 19 encode 6 as the device’s address).

   \begin{verbatim}
   /usr/sbin/mknod /dev/mydriver b 65 0x026000
   /usr/sbin/mknod /dev/mydriver c 234 0x026000
   \end{verbatim}

5. Verify the configuration by invoking \texttt{ioscan} with the \texttt{-fun} or \texttt{-fkn} options. If the device-special files are created properly, \texttt{ioscan} displays them beneath the configured device.

For information on packaging third-party drivers for distribution with HP-UX, see Chapter 19, “Creating a Software Depot.”
6 Creating Networking Device Drivers

This chapter provides information for designing and writing PCI networking device drivers.

The information in this chapter is intended for developers with extensive experience in designing and writing networking device drivers for non-HP UNIX target systems.

Basic STREAMS module/driver development and general networking concepts and RFCs are not included in this documentation.

The major difference in the network device driver model from HP-UX 10.20 for third party developers is that third party developers are expected to implement their own DLPI layer and not depend on or use the HP DLPI. Also, HP supports “pure” STREAMS model drivers; BSD style drivers are not supported anymore.

The first section contains an overview of the structure of networking drivers. Use the steps outlined in this section as a general guide to HP-UX driver design. The second section introduces the HP-UX networking interface architecture for the PCI platform. Review this section before beginning development of the networking driver. The remaining sections of this chapter contain network device driver topics and sample code for each part. Refer to this information, as well as the sample driver provided in the Driver Development Kit, to create the PCI networking device driver.
HP-UX Networking Interface Architecture

This section describes the HP-UX networking interface architecture for the PCI bus. The interface supports OSI protocols, Internet protocols, and DLPI protocols on HP-UX platforms.

The HP-UX networking subsystem comprises three logical layers, as shown in Figure 6-1, “Three Layered HP-UX Interface to the PCI Bus,” and is briefly described in the following four subsections:

“Data Link Interface Layer”

“Network Protocol Layer”

“Protocol Interface Layer”

“STREAMS Environment”

Figure 6-1 Three Layered HP-UX Interface to the PCI Bus
Data Link Interface Layer

Data Link layer has STREAMS DLPI drivers. A DLPI driver interacts with STREAMS modules in the system. The network interface part of the driver is responsible for manipulating its hardware devices (e.g., Ethernet cards) and for encapsulating/decapsulating link level (e.g., SNAP) headers that are required to deliver messages to a destination. The data link layer:

- Directly connects to the network interface hardware (network interface, physical layer).
- Consists of the hardware interfaces and their respective device drivers.
- Implements DLPI Version 2.0 to interact with STREAM/UX Transport Stack.

Network Protocol Layer

The network protocol layer, above the datalink interface layer, encompasses four protocol families:

- Internet:
  - TCP/IP
  - UDP/IP
- OSI
- X.25
- ARP

Each network protocol family belongs to a domain and uses the address scheme in that domain. For example, the Internet (INET) family of protocols form the Internet domain.

The network protocols of other domains, such as the OSI stack, may be functionally equivalent to the Internet stack, but are generally not compatible with Internet domain protocols.

Protocol Interface Layer

This interface layer directly supports applications; its main functions are to:

- Identify different applications on the same host (for example, a socket interface or a device file interface).
- Provide services from transport layer protocols and to applications.

The interface for this layer provides the following abstract objects that applications can create, open, connect, or use for sending or receiving data:

- Sockets
- Streams
- Device files
STREAMS Environment

The kernel modules for the HP-UX transports (e.g., TCP/IP, UDP, OSI) are now STREAMS modules. Drivers that interface to the transport stacks must now work within this environment.

Driver developers should refer to the following documents for information concerning STREAMS modules and device drivers. Attention should be paid to the DLPI references. This document only briefly discusses the STREAMS mechanisms and concentrates on specific HP variants.

The following documents are recommended sources:

Hewlett-Packard Company:

- HP 9000 Networking DLPI Programmer's Guide, HP Part No. 98194-90059

Other References:

- Data Link Provider Interface Specifications, Unix International
Networking Driver Structure

The flowchart in Figure 6-2, “Steps to Develop a Networking Driver,” shows a suggested sequence to use when developing networking drivers on HP-UX systems. Step one lists the mandatory information necessary, or standard knowledge base, for a basic driver. Steps two through eight list the options available for increased network driver capabilities. The sequence of information in the flowchart closely follows the organization of this chapter. Refer to each step’s description for pointers to its applicable detailed subject areas.

Figure 6-2  Steps to Develop a Networking Driver
1. This step in the network driver development lists the mandatory knowledge base needed to tailor the driver basic functions. The topics are:

   HP-UX Network Interface Architecture
   An overview of the STREAMS environment. Refer to “STREAMS DLPI Overview”.

   Data Structures
   Describes the data structures in the networking interface layer: hw_ift_t, hw_dlpi_t and device driver data structure framework. Refer to “Data Structures and Interfaces” for detailed information about these data structures.

   Protection and Synchronization
   Describes the OSF/Encore spinlock protection model. Refer to “Protection and Synchronization” for more detailed information about supporting the spinlock scheme in HP-UX.

   Network Driver Initialization
   Describes the install and initialization routines for the STREAMS DLPI driver. The attach routine is discussed for the driver. Refer to “Initializing Networking Device Drivers” for detailed information about these routines.

   Protocol Configuration, Binding, and Demultiplexing
   Describes configuration of the INET stack for the STREAMS model drivers. Also, the routines for the driver to bind and demultiplex upper layer protocols to a device are explained. Refer to “Protocol Binding and Demultiplexing” for detailed information on these routines.

   mblk and queue macros
   These are macros commonly used by STREAMS networking drivers. Refer to “Message Block and Queue Functions”.

   DLPI Interface
   Describes how upper layers are linked to the network drivers via the DLPI. Refer to “DLPI Interface”.

   STREAMS DLPI Network Driver
   Provides an overview of the DLPI and WSIO interface portions of the STREAMS DLPI network driver. Major driver functions are also explained. Refer to “STREAMS DLPI Overview” for more detailed information.

   The following steps list the options available when developing a network driver.

2. Auxiliary Code
   HP customers expect to have network management and tracing and logging support in their networking products. HP recommends adding these routines to your network driver.

   If selected, implement the code, then proceed to the next option. If not selected, go to step five.

3. Network Management Support
   A description of the routines that support Network Management requests. Refer to “Network Management Support” for more detailed information.

   Select or go to the next option.
4. Network Tracing and Logging Support
   A detailed discussion of the topic is provided in Chapter 8, “Tracing and Logging in LAN Drivers.”
   Select or go to the next option.

5. Auxiliary Files
   HP customers expect to have automated configuration through the System Administration Manager
   (SAM) and be able to display link and encapsulation statistics and tracing and logging messages.
   If Auxiliary Files are not required, the driver is complete. If they are required, go to the next option.

6. SAM Support
   Refer to Chapter 9, “SAM Support for LAN Drivers,” for detailed information on adding SAM support in
   IHV network interface drivers.
   Select or go to the next option

7. LAN Commands Support
   Refer to Chapter 7, “LAN Commands,” for more information on LAN commands and how to add support
   for them in IHV network interface drivers. This includes discussion regarding any required shared
   libraries.
   The driver is now complete.

STREAMS Device Drivers
Starting with HP-UX 11.0, IHVs and ISVs are expected to write their own DLPI layer implementation in the
STREAMS network interface driver. A network driver in HP-UX 11i is a native STREAMS DLPI driver. This
document provides a framework that includes a native STREAMS DLPI PCI network interface driver, enet,
which has a sample DLPI implementation and the device interface part, as part of the Driver Development
Kit.

NOTE The names STREAMS DLPI driver, native STREAMS DLPI, native DLPI driver and DLPI
driver are used interchangeably in this chapter.
Data Structures and Interfaces

The following data structures are used by the network interface layer:

- **hw_ift_t** (defined in `sio/lan_dlpikrn.h`)
- **hw_dlpi_t** (contained in `hw_ift_t`; defined in `sio/lan_dlpikrn.h`)

Each device driver may maintain its `hw_ift_t` and `hw_dlpi_t` structure as part of a larger structure, the driver control block `enet_ift_t`, shown in Figure 6-3, “Networking Driver Control Block and Structures.” The driver control block provides information used in driving and controlling the interface hardware.

**Figure 6-3 Networking Driver Control Block and Structures**

![Networking Driver Control Block and Structures](image)

**hw_ift_t Structure**

The `hw_ift_t` structure provides a consistent interface to the network system utilities `lanscan` (see `lanscan` (1M)), `driveradmin`, and `driverlinkloop` to display detailed information for all network devices. The `hw_ift_t` structure is described:

```c
typedef struct hw_ift
{
    hw_dlpi_t     hp_dlpi;
    uint32_t  mac_type;
    uint32_t  llc_flags;
    uint32_t  mjr_num;
    uint32_t  nm_id;
    uint32_t  instance_num;
    uint32_t  mtu;
    char *    name;
    uint8_t  hdw_path[MAX_HDW_PATH_LEN];
    uint32_t  hdw_state;
    uint32_t  mac_addr_len;
    uint8_t  mac_addr[MAX_MAC_ADDR_LEN];
    uint32_t  features;
    uint8_t  *arpmod_name;
    uint32_t  ppa;
    uint32_t  watch_timer;
    uint32_t  reserved2;
    lock_t   *hwift_lock;
    struct hw_ift *next;
} hw_ift_t;
```
Creating Networking Device Drivers
Data Structures and Interfaces

The following fields must be properly initialized by the device driver during system initialization to support the HP-UX system utilities:

- **hp_dlpi**: Must be initialized to all zeros
- **mac_type**: Device type
- **llc_flags**: **Link Level Control** (LLC) encapsulation method.
- **mjr_num**: Major number of the device file. The major number should be set to -1.
- **nm_id**: Network management ID, should be initialized via a call to the `get_nmid()` routine.
- **instance_num**: Device instance number — the value returned by calling the `wsio_isc_to_instance()` routine.
- **mtu**: Maximum transmission unit (number of bytes) for a particular type of link or encapsulation.
- **name**: Driver device name; used for naming shared libraries for `lanscan` and `driveradmin`.

**NOTE** The driver names `lan` and `fddi` are reserved for HP devices.

- **hdw_path**: Hardware path obtained by calling `io_node_to_hw_path()` followed by `io_hw_path_to_str()`.
- **hdw_state**: Hardware state of the device; zero, if the device is OK. If the device is not available, `hdw_state` is set to `LAN_DEAD`.
- **mac_addr_len**: Number of bytes of `mac_addr[]` for MAC address.
- **mac_addr**: MAC address of the device. For Ethernet/IEEE 802.3 and FDDI, the address is in canonical form. For IEEE 802.5, the address is in wire form.
- **features**: Features supported by the device. The following flags are provided:
  - **DRV_MP**: Set this flag and make sure the device driver is MP scalable or MP safe; that is, uses `spinlock()` or `spinunlock()` to avoid race conditions. See “**Protection and Synchronization**” for more information. When this flag is set, the driver cannot use any `spl*` calls.
  - **DRV_MBLK**: This flag must be set, the third party network driver is purely based on STREAMS model.
  - **DRV_IP_MULTICAST**: This flag must be set if a driver supports the IP multicast feature.
  - **DRV_LANC_PROMISC_SUPPORT**: This flag must be set if a driver supports promiscuous listening.
  - **DRV_NO_FAST_PATH**: This flag must be set if a driver does not support fast path as described in “**Transmission of Message Blocks**”.
  - **DRV_CKO**: This flag must be set if a driver supports TCP or UDP checksum calculations in hardware.

- **arpmod_name**: The name of ARP STREAMS helper module. This module complements the generic ARP module to resolve addresses in networks like Token Ring and Fiber Channel.
- **ppa**: **Physical Point of Attachment** (PPA) number for the interface. The driver should initialize this field with `hw_ift->instance_num`. 

**Chapter 6** 169
Creating Networking Device Drivers
Data Structures and Interfaces

watch_timer  For Hewlett-Packard internal use only. This field must be set to zero for non-Hewlett-Packard devices.

reserved2  For Hewlett-Packard internal use only. This field must be set to zero for non-Hewlett-Packard devices.

hwift_lock  Pointer to a hwift_lock spinlock structure to protect the hw_ift structure. This field is initialized in hw_ift_attach().

next  Pointer to next hw_ift structure in list. This field is set by calling the hw_ift_attach() routine during device driver initialization. See “Initializing Networking Device Drivers” in this chapter for detailed information.

The following example shows the initialization of the hw_ift structure. Initialization is generally done in the driver init routine:

```c
struct enet_ift_t *enetift_ptr;
hw_ift_t *hw_ift_ptr;
char mac_addr[6];
struct isc_table_type *isc_ptr; /* pointer to an isc_table structure */

hw_path_t hw_path;

hw_ift_ptr = &(enetift_ptr->hwift);

hw_ift_ptr->mac_type = DEV_ETHER;
hw_ift_ptr->llc_flags = IEEE | SNAP;
hw_ift_ptr->mjr_num = enet_drv_info.drv_info->c_major;
hw_ift_ptr->nm_id = get_nmid();
hw_ift_ptr->instance_num = wsio_isc_to_instance(isc_ptr, NULL);
hw_ift_ptr->mtu = ETHER_MTU;
hw_ift_ptr->name = "enet";
io_node_to_path(isc_ptr->card_node,NULL,&hw_path);
io_hw_path_to_str(hw_ift_ptr->hdw_path,NULL,&hw_path);
hw_ift_ptr->hdw_state = LAN_DEAD;
bcopy((caddr_t)(mac_addr),
     (caddr_t)(hw_ift_ptr->mac_addr), 6);
hw_ift_ptr->features = DRV_MP | DRV_MBLK;
hw_ift_ptr->arpmod_name = (u_char *)"";
hw_ift_ptr->watch_timer = 0;
hw_ift_ptr->ppa = hw_ift_ptr->instance_num;
hw_ift_ptr->reserved2 = 0;
```

hw_dlpi Structure

This structure provides support for HP-UX DLPI connections; it should be initialized to zero. Further discussion of structure fields is not provided.
Protection and Synchronization

The major synchronization issue with networking device drivers is avoiding data corruption and race conditions when shared structures are accessed by multiple threads in MP systems. Driver data structures also need protection against interrupts. HP-UX transport networking adopted the OSF/Encore spinlock protection model to gain parallelism and provide scalable network performance. The spinlock scheme provides finer granularity locks, protecting data structures at finer levels, as opposed to grabbing a global network lock. More information on spinlocks is available in Chapter 3, “Multiprocessing,” of this manual and spinlock (KER2) in the HP-UX 11i v1 Device Driver Reference Manual.

NOTE Each spinlock causes a busy-wait. Device driver developers should be aware of the impact on system performance caused by the frequency of acquiring a spinlock and the duration of holding a spinlock.

As discussed in previous sections, the data structure in the network interface layer that link device drivers to the protocol layer is the hw_ift. The drivers have their own data structures; driver control block, send and receive management. These data structures are protected by using spinlocks.

hw_ift Structure Protection

One spinlock, the hwift_lock field in the hw_ift structure, is defined to protect the access to the structure fields.

The macros to acquire or release the hwift_lock spinlock to protect hw_ift structure fields are defined:

\[
\begin{align*}
\text{HW_IFT_LOCK(hw_ift_ptr)} & \quad \text{Acquire a spinlock on hwift_lock. hw_ift_ptr: pointer to an hw_ift structure.} \\
\text{HW_IFT_UNLOCK(hw_ift_ptr)} & \quad \text{Release previously acquired hwift_lock spinlock. hw_ift_ptr: pointer to an hw_ift structure.}
\end{align*}
\]

NOTE The hwift_lock spinlock is allocated and initialized by the hw_ift_attach() routine. As a result, the HW_IFT_LOCK() and HW_IFT_UNLOCK macros are not available until returning from the hw_ift_attach() routine.

Driver Structure Protection

Networking drivers use spinlocks to protect their internal data structures. HP-UX predefines the order (major order) for spinlocks for LAN and STREAMS drivers to avoid deadlock conditions when non-direct code paths are executed due to faults, traps, or interrupts.

Drivers can increase concurrency with finer granularity locks. The major lock order is predefined by HP-UX so drivers can use different minor order spinlocks to protect access to data structures. For example, a network interface driver can use one lock for transmit path and another for receive path data structures. This allows the driver to receive and transmit concurrently.
A list of the relative predefined lock orders for spinlocks used by HP-UX LAN products is shown:

**LAN_LANX_LOCK_ORDER**
Lock order for a spinlock used by HP-UX LAN device drivers, such as \texttt{btlan3} and \texttt{lan2}, to protect local data structures. This lock order should be used by all third party networking device drivers during initialization of a spinlock used to protect device driver structures.

**LAN_HWIFT_LOCK_ORDER**
Lock order for spinlock \texttt{hwift_lock}, defined in \texttt{sio/lan_dlpikrn.h}, and the lock order protecting the embedded MIB structure.

**STREAMS_USR1_LOCK_ORDER**
Lock order for spinlock used by STREAMS drivers to protect their data structures.
Initializing Networking Device Drivers

In HP-UX version 11i, developing a network interface driver involves developing a STREAMS DLPI network interface driver. A DLPI driver is part of STREAMS/UX and is used by the file system for device open and close. For this reason the DLPI driver is both a STREAMS and WSIO-CDIO driver. Initialization for a DLPI network driver is described in this section.

For a detailed description of generic STREAMS driver development, refer to the *STREAMS/UX for HP9000 Reference Manual*. This section explains the initialization process with the help of excerpts from a sample driver *enet*.

The install routine of a STREAMS DLPI driver, *driver_install()*, should call WSIO-CDIO install *wsio_install_driver()* and STREAMS/UX install *str_install()* functions.

The WSIO-CDIO system requires the following data structures to be defined and initialized before calling *wsio_install_driver()* in *driver_install()*:

```c
drv_ops_t enet_drv_ops;
drv_info_t enet_drv_info = {
    "enet",  /* driver name */
    "pseudo" /* driver clas */
    DRV_CHAR|DRV_PSEUDO|DRV_MP_SAFE, /* type */
    -1,     /* block major number */
    -1,     /* character major number */
    NULL, NULL, NULL /* always NULL */
};
```

STREAMS/UX requires that *streams_info_t* be initialized as shown in the following code sample. This structure is passed in the call *str_install()*:

```c
static struct module_info enet_rminfo= {
    5050, "enet", 0, 65536, 65536, 1
};
static struct module_info enet_wminfo= {
    5050, "enet", 0, 65536, 1, 1
};
static struct qinit enet_rinit= {
    0, enet_rsrv, enet_open, enet_close, 0, &enet_rminfo
};
static struct qinit enet_winit= {
    enet_wput, enet_wsrv, 0, 0, 0, &enet_wminfo
};
struct streamtab enet_info= {&enet_rinit, &enet_winit};
streams_info_t enet_str_info= {
    "enet" /* name */
    -1,  /* dynamic mj# */
    { &enet_rinit, &enet_winit, NULL, NULL},
    "STREAMS|MP|SYSV4", /* stream flags */
    SQLVL_SYNC, /* sync level */
};
```

In addition to a *driver_install* (WSIO_DRV) routine, each HP-UX PCI networking device driver must have a *driver_attach* (WSIO_DRV) routine.
Creating Networking Device Drivers
Initializing Networking Device Drivers

If a networking device driver interfaces with a hardware device, it is required to have a service routine to handle the device interrupts.

The following brief descriptions of the required install, attach and initialization routines introduce the networking device driver initialization.

The following install routine for the sample driver should be prefixed with the driver name.

```c
int driver_install()
```

An entry of `driver_install()` is called during the I/O system configuration process. When the `driver_install()` routine is called, it hooks the `driver_attach()` entry to the top of a linked list of attach routines for all of the interface drivers in the system.

```c
#define __LP64__
int driver_attach(uint32_t product_id, struct isc_table_type *isc_ptr)
#else
driver_attach(PCI_ID product_id, struct isc_table_type *isc_ptr)
#endif

product_id Four bytes of PCI product ID.
isc_ptr Pointer to `isc_table_type` structure.

```c
void driver_init(struct isc_table_type *isc_ptr)
int driver_isr(struct isc_table_type *isc_ptr, caddr_t cb_ptr)
```

`cb_ptr` Pointer to the driver control block; it is driver developer defined and passed as a pointer through the `isrlink()` routine during the `driver_attach()` or `driver_init()` routines.

The `driver_attach()` and `driver_install()` initialization procedures are common to all HP-UX device drivers. More details of each step are presented in Chapter 4, “Writing a Driver.”

**Calling driver_install()**

When the HP-UX system is configured through the `config` command, a table of `driver_install()` entry points is created from information in `/stand/system`.

When `driver_install()` is called by the I/O system configuration process through the `driver_install()` entry point configured in the system, the `driver_install()` routine places the `driver_attach()` entry in a table of drivers to be called at configuration time. The `driver_install()` routine calls the `wsio_install_driver()` routine to register the driver with the I/O subsystem and returns any error.

The following is a call to `driver_install()`:

```c
static drv_ops_t enet_drv_ops = {
    NULL,           /* open */
    NULL,           /* close */
    NULL,           /* strategy */
    NULL,           /* dump */
    NULL,           /* psize */
    NULL,           /* reserved */
    NULL,           /* read */
    NULL,           /* write */
    NULL,           /* ioctl */
    NULL,           /* select */
    NULL,           /* option */
    NULL,           /* reserved */
    NULL,           /* reserved2 */
    NULL,           /* reserved3 */
    NULL,           /* link */
    0,              /* device flags */
};
```
static drv_info_t enet_drv_info = {
  "enet", /* driver name */
  "lan", /* class name */
  DRV_CHAR | DRV_SAVE_CONF | DRV_MP_SAFE, /* driver flags */
  -1, /* block major number */
  -1, /* character major number */
  NULL, NULL, NULL, /* structures always set to NULL */
};

static wsio_drv_data_t enet_data = {
  "enet", /* for matching probes with drivers */
  T_INTERFACE, /* type of hardware, dev or IF */
  DRV_CONVERGED, /* driver flag */
  NULL,/* minor number build routine */
  NULL,/* minor number decode routine */
};

static wsio_drv_info_t enet_wsio_info = {
  &enet_drv_info, /* driver info */
  &enet_drv_ops, /* driver ops */
  &enet_data, /* driver data */
  WSIO_DRV_CURRENT_VERSION
};

/* to attach PCI driver to system */
int (*enet_saved_attach)();

int enet_install()
{
  enet_saved_attach = pci_attach;
  pci_attach = enet_attach;
  bzero((caddr_t)&enet_drv_ops, sizeof(drv_ops_t));
  msg_printf("enet:install\n");
  if(!(rv=wsio_install_driver(&enet_drv_info,
                     &enet_drv_ops))){
    if(rv=str_install(&enet_str_info)) {
      wsio_uninstall_driver(&enet_drv_info);
      msg_printf("enet:install failed\n");
    }
    return rv;
  }

  Calling driver_attach()

Use the driver_attach() routine to determine whether the product ID passed in matches the driver_attach device and vendor ID. If the IDs do not match, the driver_attach() routine calls the next attach routine in the chain by calling the %driver_saved_attach() routine.

NOTE The driver_attach() routine may be called many times before a match is found. For the device in the first slot, the associated driver_attach() routine is called by the number of devices in the PCI backplane. For the device in the last slot of the PCI backplane, the associated driver_attach() routine is called only once.
When the `driver_attach()` routine recognizes the device ID it allocates and initializes its driver control blocks and PCI I/O registers. The `driver_attach()` routine also sets up a driver initialization routine and calls the `isc_claim()` to claim the device. The following is a sample `driver_attach()` routine:

```c
struct gfsw enet_gfsw;
...
int #ifdef __LP64__
enet_attach( uint32_t id, struct isc_table_type *isc)
#else
enet_attach( PCI_ID id, struct isc_table_type *isc)
#endif
{
  msg_printf("enet attach id = %x\n", id);
  #ifndef __LP64__
    /* Support for PCI only */
    if (!((id.vendor_id==DEV_VENDORID && id.device_id==DEV_DEVICEID))) {
      return enet_saved_pci_attach(id, isc);
    }
  #else
    if (!((id == DEV_ID))) {
      return enet_saved_pci_attach(id, isc);
    }
  #endif
  isc->gfsw = &enet_gfsw;
  CONNECT_INIT_ROUTINE(isc, enet_init);
  isc->gfsw->diag = (int (*)())NULL;
  #ifdef __LP64__
    isc->if_id = (int)(id & 0x0000ffffU);
  #else
    isc->if_id = (int)id.device_id;
  #endif
  isc_claim(isc, &enet_wsio_drv_info);
  return enet_saved_pci_attach(id, isc);
}
```

HP-UX calls a `driver_init()` routine to begin driver initialization. It allocates the driver control block and driver data structures, sets PCI configuration information, links the driver ISR to the PCI interrupt, and initializes and resets the controller hardware. The following is the skeleton initialization function showing PCI configuration and linking of the driver ISR:

```c
int enet_init(struct isc_table_type *isc)
{
  enet_ift_t *enet_iftp;
  size_t size;
  u_long phys_base;
  ...
  ubit32 base_addrp, id, revid, latency_timer, int_reg;
  ubit32 sub_id, ssid, cfda, csr6;
  BUS_TRANS_DESC desc;
  ubit32 error;
  ...
  ...
  /*
   * Allocate driver control block - enet_iftp
   */
  ...
  ```
Creating Networking Device Drivers

Initializing Networking Device Drivers

Chapter 6

/*
* Obtain memory for Transmit and Receive Descriptor
* Rings and any additional driver data structures */
...

/ *
* Get/Set PCI configuration
*/
pci_read_cfg_uint32_isc(isc,SSID,&ssid);
enet_iftp->sub_id = (ubit16)(ssid >> 16);
enet_iftp->sub_vendor_id = (ubit16)(ssid & 0x0000ffff);

/ * Read the Configuration ID information */
pci_read_cfg_uint32_isc(isc,CFID,&id);

/ * Read the Configuration Revision information */
pci_read_cfg_uint32_isc(isc,CFRV,&revid);

/ * Read the Configuration Interrupt information */
pci_read_cfg_uint32_isc(isc,CFIT,&int_reg);

/ * Read the Configuration Driver Area information */
pci_read_cfg_uint32_isc(isc,CFDA,&cfda);
cfda = 0;
pici_write_cfg_uint32_isc(isc,CFDA,cfda);
....

/* Turn on PCI memory access and bus master capability
* on host */
pici_write_cfg_uint8_isc(isc, CFCS,
    CFCS_MEMORY_SPACE_ACCESS |
    CFCS_MASTER_OPERATION |
    CFCS_PARITY_ERROR_RESPONSE |
    CFCS_SYSTEM_ERROR_ENABLE |
    CFCS_I_0_SPACE_ACCESS);
....

/* Init and reset the controller
*/

/* Perform general enet_iftp initialization
*/
...

/* Setup hwift structure */
...
...

/* Attach hwift to global list */
hw_ift_attach(&enet_iftp->lancift.hwift);
...
...

/* size: initialized to the size of enet_iftp->tdr
 (transmit descriptor ring) */
/* Allocate the DMA handle for Tx-descriptor ring */
enet_iftp->tdr_DMA_handle = wsio_allocate_dma_handle(isc);

/* Allocate shared memory for Tx-descriptor ring */
if( wsio_allocate_shared_mem(isc, 
    enet_iftp->tdr_DMA_handle, size, 
    (caddr_t *)& enet_iftp->tdr, 0) != WSIO_MAP_OK) { 
    msg_printf("enet - TDR allocation failed...\n");
    return -1;
}

...

If initialization is successful, the driver_init() routine proceeds with the following steps:

- Initializes the MIB structure and the hw_dlpi and hw_ift structures (see the preceding sections hw_ift_t Structure and hw_dlpi Structure for details.
- Calls the hw_ift_attach() routine to link the hw_ift structure to a global list of hw_ift structures of active interfaces. The hw_ift_attach() routine is defined as:

  hw_ift_attach(hw_ift_t * hw_ift_ptr)

  hw_ift_ptr Pointer to the password hw_ift structure.
Protocol Binding and Demultiplexing

This is the mechanism a networking driver uses to associate (bind) an upper layer protocol to a device. The binding ensures the driver correctly demultiplexes and delivers inbound packets to the corresponding upper layer protocol, based on the upper layer protocol's bind request.

To correctly demultiplex inbound packets, a networking driver must:

- Obtain protocol specific information during protocol binding.
- Obtain packet specific information.
- Process packets and information by the upper level protocols.

The following Table 6-1, “Protocol Kind and Value,” summarizes the information a networking driver requires to demultiplex inbound packets for corresponding upper layer protocols. More detailed information is provided in the section DLPI Interface.

Table 6-1 Protocol Kind and Value

<table>
<thead>
<tr>
<th>Packet Type</th>
<th>Protocol Kind</th>
<th>Protocol Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ethernet type</td>
<td>LAN_TYPE</td>
<td>TYPE value</td>
</tr>
<tr>
<td>IEEE 802.2 LLC type</td>
<td>LAN_SAP</td>
<td>SAP value</td>
</tr>
<tr>
<td>SNAP type</td>
<td>LAN_SNAP</td>
<td>OID + extended SNAP info</td>
</tr>
</tbody>
</table>

Protocol kind is the type of protocol to bind. Interpretation of the protocol value field depends on the protocol kind value.

When the networking driver binds a protocol with protocol kind and type values, the driver knows what kind of packets to handle for that bind. The networking driver processes inbound packets on the Interrupt Control Stack (ICS) for all the protocol binds by calling an associated STREAMS queue. To do this, it calls putnext() (see the STREAMS/UX for the HP 9000 Reference Manual) in the device driver’s interrupt service routine. The driver must use the protocol ID that was carried in the dl_sap field of the DL_BIND_REQ to pass the packet to the right Stream that is logged (see “DLPI Interface” section for details).

Protocol Binding

Each upper layer protocol issues a bind request to the networking driver to affect binding. The driver is responsible for keeping track of all upper layer protocols currently bound to it. The networking driver also must have a way to unbind a protocol upon request.

Protocol Demultiplexing

One of the main functions of the device driver’s interrupt service routine is to dispatch inbound packets to the appropriate upper layer protocol. To achieve that, the interrupt service routine in the driver must:

1. Distinguish packet protocol format and type:
   - Ethernet
   - IEEE 802.2 Link Level Control (LLC) (non-SNAP)
SNAP (IEEE 802.2 LLC extended)

2. Locate the proper inbound packet service routine or queue for each valid incoming packet.

Packet Protocol Format

The following information can be used to determine the protocol format and type. To determine whether the packet is an Ethernet type packet:

- If the value of the TYPE field of an inbound packet is equal to or greater than 0x600, the packet is an Ethernet type packet. The protocol kind of the packet is `LAN_TYPE`, and the protocol value is the TYPE field specified in the packet.
- If the value of the TYPE field is less than 0x600, the packet could be an IEEE 802.2 LLC type packet, SNAP or non-SNAP type.

To determine whether the packet is a SNAP type IEEE 802.2 LLC packet:

- The packet is considered to be a SNAP packet (defined in IEEE 802.1a) if both the DSAP and the SSAP values are 0xAA. The protocol kind of the packet is `LAN_SNAP`, the protocol value is 0xAA, and the protocol value extended is the five-byte SNAP protocol data specified in the SNAP header.
- Otherwise, it is an IEEE 802.2 LLC non-SNAP type packet. The protocol kind is `LAN_SAP` and the protocol value is the DSAP field that is specified in the packet.

The relationships of protocol kind, protocol value, and protocol processing for different types of packets are shown in Table 6-1, “Protocol Kind and Value.”

After the device driver has found the protocol kind and value in an inbound packet, the driver locates the protocol input queue that corresponds with the bind request previously received from an upper layer protocol. This queue information is stored by the driver during binding.

If the upper layer requires header stripping, the device driver strips off the `Link Level Control` (LLC) header before passing the inbound packet to the upstream queue.
Promiscuous Inbound and Outbound

For inbound promiscuous, a promiscuous stream receives the packets destined for other streams (protocols) and (depending on the promiscuous level enabled) other NICs.

For outbound promiscuous, the stream traces all packets on the interface (depending on the promiscuous level enabled).

*Table 6-2, “Promiscuous Mode Matrix,” explains each promiscuous mode.*

<table>
<thead>
<tr>
<th></th>
<th>PROMISC_PHY</th>
<th>PROMISC_MULTI</th>
<th>PROMISC_SAP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unbound</strong></td>
<td>The stream gets all outbound packets transmitted on the interface. (broadcast, multicast, self addressed and non self addressed unicast packets).</td>
<td>The stream gets all outbound multicast, broadcast packets transmitted on the interface. No outbound unicast packets will be seen.</td>
<td>The stream gets all outbound packets when the “source” SAP matches one of the protocols enabled on the interface.</td>
</tr>
<tr>
<td>promiscuous stream monitors <strong>outbound</strong> traffic.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Unbound</strong></td>
<td>The stream gets all packets on the wire regardless of SAP or address.</td>
<td>The promiscuous stream gets all multicast, broadcast packets on the wire regardless of SAP or SNAP. No unicast packets will be seen on an inbound traffic.</td>
<td>The promiscuous stream gets all packets which pass the physical level filtering (local MAC, broadcast, or multicast addresses) for the interface and passes the protocol filtering (SAP type or SNAP enabled on that interface).</td>
</tr>
<tr>
<td>promiscuous stream monitors <strong>inbound</strong> traffic.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Bound</strong></td>
<td>The stream gets all outbound packets that match the SAP protocols that the user has bound to on the promiscuous stream.</td>
<td>The stream gets all outbound multicast, broadcast packets that match the SAP protocol the user has bound to on the promiscuous stream. No unicast will be seen.</td>
<td>This primitive has no effect on the interface.</td>
</tr>
<tr>
<td>promiscuous stream monitors the <strong>outbound</strong> traffic.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Bound</strong></td>
<td>The promiscuous stream gets all packets on the wire that match the SAP protocols that the user has bound to on the promiscuous stream.</td>
<td>The promiscuous stream gets all multicast, broadcast and unicast packets that match the SAP protocol the user has bound to on the promiscuous stream.</td>
<td>This primitive has no effect on the interface.</td>
</tr>
<tr>
<td>promiscuous stream monitors <strong>inbound</strong> traffic.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Message Block and Queue Functions

The message block and queue functions and macros are defined by STREAMS/UX. Refer to the STREAMS/UX for the HP 9000 Reference Manual for further information.

Starting with HP-UX version 11i, the header in the message block data structure `mblk_t` is not cacheline aligned. The area in an `mblk` to store data follows the header. Since the header is not cacheline aligned, part of the header shares a cacheline with the data area. If a driver purges the cache corresponding to the data area in order to read DMA data, it may corrupt the message block header since the data area and the header share the same cacheline. Therefore, drivers are required to take precautions to avoid the problem. One solution is to verify the data area and the header are in different cachelines:

The list of commonly used message block functions:

- `allocb()`  Allocate a message block
- `freemsg()`  Free a message block
- `pullupmsg()`  Concatenate and align the data stored in complex message
- `adjmsg()`  Adjust the length of the message
- `dupmsg()`  Duplicate a simple or complex message

The following is the list of queue functions commonly used in a STREAMS driver:

- `putq()`  Queue message to be processed by queue service procedure
- `putnext()`  Call queue’s “put” procedure
- `canput()`  Test whether queue can receive message
- `qreply()`  Send the message back upstream
- `OTHERQ()`  Other queue in the queue pair
- `streams_put()`, `streams_put_release()`  Allow non-STREAMS/UX (e.g., driver ICS) to “put” in a queue.
DLPI Interface

The Data Link Provider Interface (DLPI) specifies a STREAMS based kernel implementation of the ISO Data Link Service Definition (ISO 8886) and Logical Link Control (ISO 8802/2 LLC). DLPI allows a data link service user to access and use a variety of conforming data link services without special knowledge of the provider’s protocol. The interface specifies access to data link service providers and does not define a specific protocol implementation.

Starting with HP-UX version 11.0, transports (e.g., TCP/IP, UDP, OSI) are now STREAMS modules. Third parties are expected to develop a STREAMS DLPI driver conforming to DLPI version 2.0 to support their network interface drivers and can not depend on the HP DLPI implementation. This section provides information about how third party drivers can integrate into a STREAMS/UX framework in HP-UX.

Two styles of DLPI provider are defined by the DLPI document, distinguished by the way they enable a DLPI user to choose a particular physical point of attachment (PPA). The Style 1 provider assigns a PPA based on the major/minor device the DLPI user opened. The Style 2 provider requires a DLPI user to explicitly identify the desired PPA by using a special attach service primitive. This document illustrates the development of a Style 2 DLPI driver.

Device Information

DLPI users can access DLPI providers through generic DLPI device files (i.e., a device file corresponding to a DLPI STREAMS driver). A DLPI device file can be created by `mknod` (2) or `insf` (1M) by using device driver information from `lsdev` (1M). The following example shows the device file `enet` (sample STREAMS DLPI driver). The device files created for the STREAMS DLPI driver are also shown:

```
# lsdev
............................................
............................................
 239          -1         enet            lan

# ll /dev/enet*

   crw-rw-rw-   1   rootsys    72 0x0000f0 Apr 12 18:46 /dev/enet
```

The `lanscan` (1M) lists all the LAN interfaces in the system from the list of `hw_ift_t` structures (every network interface driver should perform `hw_ift_attach()` during initialization). This list identifies the interface name and PPA numbers. Refer to “Initializing Networking Device Drivers” for details of `hw_ift_attach()`.

Table 6-3, “lanscan Output,” shows how `lanscan` illustrates the interface name and PPA numbers for the sample WSIO network driver. The sample driver has “attached” to LAN interfaces at hardware paths 8/0/1/0 and 8/0/2/0.
Once the interface name and the PPA number are known, `ifconfig` (1M) is used to configure IP and ARP. When `ifconfig` is done for `enet1` listed by `lanscan` as previously shown, the IP and ARP streams are set up as listed in the following steps:

1. `ifconfig` opens device file `/dev/enet` and senses PPA configured is 1.
2. `ifconfig` issues an `ioctl` to push IP module to top of `enet` driver.
3. `ifconfig` issues another `ioctl` to issue attach and bind requests for PPA 1.
4. `ifconfig` opens device file `/dev/enet` and issues `ioctl` to push ARP to top of `enet` driver.
5. `ifconfig` again performs step 3 for ARP/enet stream.
6. `ifconfig` opens `/dev/ip` and uses it as dummy multiplexer; IP/enet and ARP/enet steams are linked under dummy multiplexer.

### Table 6-3 lanscan Output

<table>
<thead>
<tr>
<th>H/W Path</th>
<th>Station Address</th>
<th>Card Im. #</th>
<th>H/W State</th>
<th>Net I/F, Name</th>
<th>PPA</th>
<th>NM ID</th>
<th>MAC Type</th>
<th>HP-DL PT Spport</th>
<th>DLPI Mjr#</th>
</tr>
</thead>
<tbody>
<tr>
<td>8/16/6</td>
<td>0x0060B07EDBF0</td>
<td>0</td>
<td>UP</td>
<td>lan0 snap0</td>
<td>1</td>
<td></td>
<td>ETHER</td>
<td>Yes</td>
<td>119</td>
</tr>
<tr>
<td>8/0/1/0</td>
<td>0x0060B07A221E</td>
<td>1</td>
<td>UP</td>
<td>enet1</td>
<td>2</td>
<td></td>
<td>ETHER</td>
<td>No</td>
<td>*</td>
</tr>
<tr>
<td>8/0/2/0</td>
<td>0x0060B0B2D850</td>
<td>2</td>
<td>UP</td>
<td>enet2</td>
<td>3</td>
<td></td>
<td>ETHER</td>
<td>No</td>
<td>*</td>
</tr>
</tbody>
</table>
STREAMS DLPI Overview

The DLPI Sequence in Figure 6-4, “STREAMS DLPI Network Driver Sequence,” shows the basic structure of STREAMS DLPI driver implementation in HP-UX. There are two main data structures, `enet_ift_t` and `enet_dlpi_data_t`. These two data structures establish a linkage between the DLPI specific portion and the network interface portion of the driver functionality. This is only an example implementation and is not exported by HP-UX. Third party developers may define their own interface to address their design needs. Initializing `hw_ift_t` structure was discussed in “Initializing Networking Device Drivers”.

Figure 6-4  STREAMS DLPI Network Driver Sequence
Creating Networking Device Drivers
STREAMS DLPI Overview

The general STREAMS/DLPI buffer/message processing is done in the upper part of the STREAMS DLPI network driver. The lower part of the driver implements device initialization, input, output and control functions. This section provides an overview of the synchronization of the upper and lower parts of the driver.

Device/DLPI Driver Synchronization

For a non-STREAMS character I/O mechanism, synchronization between device driver and device can be accomplished by having the device driver sleep with the sleep() kernel call on a unique number, typically an object address, while waiting for the request to complete.

Upon receipt the request completion information from the device, the device driver resumes the process with the wakeup() kernel call. For STREAMS, however, this kind of sleep-wakeup synchronization mechanism is not permitted because STREAMS may run on either the ICS or the STREAMS scheduler context stack. Synchronization between the DLPI part of the driver and the network interface part is not defined in the DLPI 2.0 documentation.

The sample DLPI driver has an enet_dlpi_wakeup() routine to support the necessary synchronization between DLPI and network interface parts of the driver. This enet_dlpi_wakeup() routine simulates the STREAMS environment wakeup() kernel call.

```c
void enet_dlpi_wakeup(caddr_t addr_ptr)
addr_ptr Address of an object to wakeup. It should correspond to the negative value returned by the enet_dlpi_process_lock() routine.
```

The driver implements a routine enet_dlpi_process_ioctl() to process ioctls. Certain actions are required of the network device driver when device control requests passed through the enet_dlpi_process_ioctl() routine return a negative value.

The following rules summarize actions each networking device driver must take in dealing with such DLPI ioctl requests:

1. The control request does one of the following:
   - If the control request completes immediately with no error, the enet_dlpi_process_ioctl() routine immediately returns zero to DLPI.
   - If the control request completes immediately with an error, the error is returned as a positive value (from errno.h).
   - If the control request cannot complete immediately (that is, the driver must make a request to the hardware), the device driver must hold the hwift_lock and return a globally unique negative value to DLPI.

2. Some time later an interrupt or timeout occurs, and the device driver interrupt service routine determines if the interrupt is for a previously blocked and waiting request.

3. The device driver completes the previous enet_dlpi_process_ioctl() by placing the results in the appropriate location for that ioctl.

4. The device driver calls the enet_dlpi_wakeup() routine with the address of the sleep object that the enet_dlpi_process_ioctl() routine previously returned to DLPI.
STREAMS Synchronization

HP-UX STREAMS supports MP scalable drivers and modules. STREAMS/UX provides five levels of parallelism called queue, queue pair, module, elsewhere and global. The queue synchronization level provides the most concurrency. Refer to the STREAMS/UX for HP 9000 Reference Manual for detailed information. The amount of parallelism for modules and drivers can be configured by specifying the synchronization level in streams_info_t during str_install(). The sample DLPI STREAMS driver uses queue synchronization level.

Entering STREAMS from ICS

When the driver is in interrupt context, it is not in STREAMS context. To enter the STREAMS framework correctly from non-STREAMS/UX code, the STREAMS/UX provides streams_put utilities. The driver ICS function can call streams_put() by passing it a function and a queue. STREAMS/UX runs the function as if it were the queue’s “put” routine. The function passed in the call can safely manipulate the queue and access the same data structures as the queue’s “put” routine. The streams_put_release() routine executes the streams_put functionality on a specified processor. Refer to the STREAMS/UX for HP 9000 Reference Manual for further information.

Driver Support for DLPI

This section discusses the upper portion of the STREAMS DLPI networking driver which buffers STREAMS messages, handles DLPI primitives, and passes data to the network interface part of the driver. This section's objective is to present the code flow of the sample driver enet as background to the sample driver code. Refer to the sample driver code for details. The following topics are discussed:

- DLPI driver data structures.
- Open and close routines.
- Control functions that describe processing of DLPI primitives such as attach/detach, bind/unbind, enable/disable multicast, enable/disable, and promiscuous.
- The main I/O path.
- DLPI primitives supported in the sample driver.

Major Data Structures

NOTE These data structures are part of the sample driver. They do not constitute any interface defined by HP-UX.

enet_dlpi_data_t

This data structure contains STREAMS DLPI driver information for a Stream that is open currently with the driver.

typedef struct _enet_dlpi{
    enet_ift_t*enetiftp;
    cred_t*cred;
    queue_t*queue_ptr;
    dev_tenet_dev;
    uint32_t dlsap_addr_length;
    uint8_t dlsap_addr[MAX_DLSAP_LEN];
    uint16_t service_mode;
    int curr_state;
};
uint32_t xidtest_flag;
int mac_type; /
int mac_mtu;
dlsap_t *dlsap_ptr;
uint8_t ssap;
uint16_t sxsap;
enet_mcast_list_t *enet_mcast_list;
int promiscuous_flg;
int promisc_filter;
uint32_t noloopback_flg;
uint32_t no_src_routing;
uint32_t arp_stream;
uint32_t ip_stream;
int fast_path;
int fast_path_pkt_type;
int fast_path_llc_length;
int pre_state;
} enet_dlpi_data_t;

The following Table 6-4, “enet_dlpi_data_t Data Fields,” explains the fields.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>enetiftp</td>
<td>The interface that is associated with this open stream.</td>
</tr>
<tr>
<td>cred</td>
<td>Credential structure of the user who opened this stream.</td>
</tr>
<tr>
<td>queue_ptr</td>
<td>Queue pointer to the read queue of the stream.</td>
</tr>
<tr>
<td>enet_dev</td>
<td>enet device number.</td>
</tr>
<tr>
<td>dlsap_addr_length</td>
<td>Length of DLSAP address.</td>
</tr>
<tr>
<td>dlsap_addr[]</td>
<td>MAC addr + SAP</td>
</tr>
<tr>
<td>service_mode</td>
<td>Only DL_CLDLS supported in the sample driver.</td>
</tr>
<tr>
<td>curr_state</td>
<td>DLPI state</td>
</tr>
<tr>
<td>xidtest_flag</td>
<td>dl_xidtest_flg from DL_BIND_REQ; indicates to the driver that XID and/or TEST responses for this stream are to be generated by DLPI driver.</td>
</tr>
<tr>
<td>mac_type</td>
<td>Interface MAC type</td>
</tr>
<tr>
<td>mac_mtu</td>
<td>Interface MTU</td>
</tr>
<tr>
<td>dlsap_ptr</td>
<td>dlsap_t structure list of logged SAPs</td>
</tr>
<tr>
<td>ssap</td>
<td>First SAP logged on stream.</td>
</tr>
<tr>
<td>sxsap</td>
<td>First extended SAP logged on stream.</td>
</tr>
<tr>
<td>enet_mcast_list</td>
<td>List of multicast addresses on this stream.</td>
</tr>
<tr>
<td>promiscuous_flag</td>
<td>Set to the promiscuous level specified in the DL_PROMISCON_REQ primitive.</td>
</tr>
<tr>
<td>promisc_filter</td>
<td>Set to one (1) if the stream has been bound with any SAP.</td>
</tr>
</tbody>
</table>
Creating Networking Device Drivers

STREAMS DLPI Overview

Chapter 6

enet_dlpi_data_ptr_arr[]

This array holds enet_dlpi_data_t pointers to keep track of the open streams.

Opening and Closing a Driver

The DLPI driver can be accessed via either a regular device or a clone of the original device. The major number of the device file for a cloneable driver must be the clone driver's major number, 72. (Refer to STREAMS/UX for HP 9000 Reference Manual for more details of clone driver). The minor number is set to the real major number of the device. The clone open is useful because the application does not need to keep track of which minor number is available and does not need to deal with multiple device files.

As can be seen from the following example, /dev/enet is a clone device file of the enet driver.

```
# ll /dev/enet*
-crw-rw-rw- 1 root    sys 72 0x0000ef Apr 12 18:46 /dev/enet
```

The actual major number of the enet driver is 239.

```
# lsdev
.................
.................
239 -1 enet lan
```

However, a clone device file for the enet driver is created as follows:

```
# mknod /dev/enet c 72 239
```

When a clone device is opened, the clone driver invokes the DLPI driver's open routine with the CLONEOPEN flag set. The open function `enet_open()` allocates the enet_dlpi_data_t for the stream being opened and initializes it. The minor number of a normal device file open is used as the index into `enet_dlpi_data_ptr_arr[]` to store and access enet_dlpi_data_t for the stream. The indexes 1 to 99 are reserved for regular open in the sample driver. For clone opens, an unused minor number starting from 100 is allocated. The enet_dlpi_data_t for the stream is stored in the `enet_dlpi_data_ptr_arr` indexed by the new minor number.

### Table 6-4 enet_dlpi_data_t Data Fields (Continued)

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>noloopback_flag</td>
<td>Set when the application wants to handle loopback. This flag is set when DLPI_SET_NOLoopBACK ioctl is issued. DLPI turns on the MSGNOLoopBACK flag in mblk message on every outbound message so driver won't loop back the packet.</td>
</tr>
<tr>
<td>no_src_routing</td>
<td>Set when DLPI_NO_SRC_ROUTING is issued.</td>
</tr>
<tr>
<td>arp_stream</td>
<td>Set if this is ARP stream.</td>
</tr>
<tr>
<td>ip_stream</td>
<td>Set if this is IP stream.</td>
</tr>
<tr>
<td>fast_path</td>
<td>Set if application requests to set up fast path.</td>
</tr>
<tr>
<td>fast_path_pkt_type</td>
<td>The fast path packet type.</td>
</tr>
<tr>
<td>fast_path_llc_length</td>
<td>The LLC header length used in the fast path.</td>
</tr>
<tr>
<td>pre_state</td>
<td>Retains the state before a pending ioctl or control request with the driver; when the request is complete the streams can be set to the correct state.</td>
</tr>
</tbody>
</table>

Chapter 6
Control Functions

The function `enet_wput()`, the STREAMS driver's “put” procedure, calls various control functions to service DLPI `M_PROTO` and `M_PCPROTO` messages with local management primitives (information reporting, attach, bind, and others such as multicast and promiscuous). This function consists of a switch table that calls the service function based on message `dl_primitive`. The following Table 6-5, “Message Service Functions,” is a list of service functions:

Table 6-5  Message Service Functions

<table>
<thead>
<tr>
<th>Function Name [prefixed by <code>enet_dlpi</code>]</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>_attach()</td>
<td>The information for PPA to be attached is found from <code>hw_if_t</code> list; <code>dli_ioctl()</code> is issued to the driver with primitive <code>DL_HP_HDW_INIT</code>. The <code>enet_dlpi_data_t</code> for this stream is updated with network interface information and the stream DLPI state.</td>
</tr>
<tr>
<td>_bind()</td>
<td><code>DL_BIND_REQ</code> primitive request indicates to bind a DLSAP to the stream. Protocol kind (LAN_TYPE, LAN_SNAP or LAN_SAP) is determined by SAP value in the request. The <code>enet_log_sap_value()</code> function is called. Once driver bind is successful, <code>dlsp_t</code> is allocated and initialized with protocol type and value of SAP. The <code>enet_dlpi_data_t</code> structure for this stream is updated with these bind details.</td>
</tr>
<tr>
<td>_control()</td>
<td>The primitives serviced by this function are — <code>DL_ENABMULTI_REQ</code>, <code>DL_DISABMULTI_REQ</code>, <code>DL_SET_PHYS_ADDR_REQ</code>, <code>DL_PROMISCON_REQ</code>, <code>DL_PROMISCOFF_REQ</code> and <code>DL_HP_HW_RESET_REQ</code>. The respective ioctl commands are issued to the driver via <code>enet_dlpi_control</code>. If the request didn't complete immediately, this routine sleeps on the address of the sleep object of the <code>dli_ioctl()</code>.</td>
</tr>
<tr>
<td>_detach()</td>
<td>Disable all multicasts that were enabled through this stream by issuing <code>dli_ioctl()</code>s to the network driver. If promiscuous mode was enabled by this stream, disable it. The <code>clean_str_sp_s_w_q()</code> routine is called to clean up any requests in the STREAMS/UX. Finally, update the state in <code>enet_dlpi_data</code> to <code>DL_UNATTACHED</code>.</td>
</tr>
<tr>
<td>_get_mib_req()</td>
<td>Services <code>MC_GET_MIB_REQ(sys/mci.h)</code>. The driver ioctl <code>DL_GET_STATISTICS</code> is issued to get current MIB statistics.</td>
</tr>
<tr>
<td>_get_mibstats()</td>
<td>Calls <code>enet_hw_req()</code> function to get the standard MIB statistics from the driver structures.</td>
</tr>
<tr>
<td>_getphyaddr()</td>
<td>The <code>enet_hw_req()</code> function is called, which selects the permanent ROM physical address of the network interface, to service <code>DL_PHYS_ADDR_REQ</code>.</td>
</tr>
<tr>
<td>_info()</td>
<td>A service function for <code>DL_INFO_REQ</code>. The information is returned upstream in structure <code>dl_info_ack_t</code>. If the PPA is not attached yet, mac type and mtu is set to <code>DL_CSMACD</code> and <code>IEEE8023_MTU</code>.</td>
</tr>
<tr>
<td>_multicast_list()</td>
<td>This function is called to service the <code>DL_HP_MULTICAST_LIST_REQ</code> primitive. In turn, this function calls driver <code>dli_ioctl()</code> to get the list by passing the command <code>DL_HP_GET_MIB_STATS</code>.</td>
</tr>
</tbody>
</table>
Creating Networking Device Drivers

STREAMS DLPI Overview

Table 6-5 Message Service Functions (Continued)

<table>
<thead>
<tr>
<th>Function Name [prefixed by enet_dlpi]</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>_ppa_req()</td>
<td>Receipt of DL_HP_PPA_REQ results in this function being called. The hw_ift_t list is searched for this PPA and the information from hw_ift_t is returned.</td>
</tr>
<tr>
<td>_set_mib_req()</td>
<td>This function services MC_SET_MIB_REQ. The driver ioctl DL_HP_RESET_STATS is issued to reset the MIB statistics.</td>
</tr>
<tr>
<td>_status()</td>
<td>This function sends the hw_ift-&gt;hw_state upstream in response to the DL_HP_HW_STATUS_REQ request.</td>
</tr>
<tr>
<td>_subs_bind()</td>
<td>When DL_SUBS_BIND_REQ is received, this function is called. If the dl_subs_bind_class is DL_PEER_BIND, a new dlsap_t is allocated and initialized with protocol type and value of SAP. With DL_HEIRARCHICAL_BIND the dlsap_addr information in enet_dlpi_data_t is updated with bind details.</td>
</tr>
<tr>
<td>_subs_unbind()</td>
<td>For each dlsap_t bound, compare the unbind request SAP. If there is a match, routine enet_unlog_sap_value() is called.</td>
</tr>
<tr>
<td>_unbind()</td>
<td>The function enet_unlog_sap_value() is called, dlsap_t is deallocated and the bind information in enet_dlpi_data_t is set to the default value.</td>
</tr>
<tr>
<td>_xidtest_out()</td>
<td>This function services DL_TEST_REQ, DL_TEST_RES, DL_XID_REQ and DL_XID_RES. It builds the test/xid packet and sends it to the driver using dlpi_output().</td>
</tr>
</tbody>
</table>

IOCTL Processing

STREAMS/UX provides the capability for user processes to perform control functions by using ioctl calls on device drivers in a stream. These commands cause the stream head to create an M_IOCTL message that includes the ioctl arguments and to send the message downstream to be received and processed by a device driver. The streams “put” function calls enet_dlpi_process_ioctl() to service M_IOCTL message types. This function consists of a switch block that services various M_IOCTL messages. The IOCTL commands are defined in sys/dlpi_ext.h.

The sample driver implements DLPI_IOC_HDR_INFO, DLPI_IOC_DRIVER_OPTIONS, and DLPI_SET_NOLoopback.

The application sends DLPI an M_IOCTL message with the ioctl command DLPI_IOC_HDR_INFO. The M_IOCTL message block is linked with the M_PROTO message block with the DL_UNITDATA_REQ primitive. The LLC header format is built for the specific interface in a new M_DATA message block and linked to M_PROTO; the whole complex message is sent back to the application.

The ioctl DLPI_IOC_DRIVER_OPTIONS routine is processed by sending hw_ift_t information for the request stream.

Depending on the device capabilities, the driver has to reset the device features which are assumed to be true by default by the transport stack. The features include driver checksum offload (DRIVER_CKO), copy on write (DRIVER_COW), long fat pipe (DRIVER_LFP) and long narrow pipe (DRIVER_LNP). The current version of the HP-UX 11i v1 Driver Development Guide (DDG) does not provide any details on implementing support for the previously listed features. So follow the implementation as given in enet_dlpi_process_ioctl() routine in the sample enet driver to inform the transport stack that the driver does not support any of these features.

Chapter 6 191
Creating Networking Device Drivers

STREAMS DLPI Overview

DLPI_SET_NOLOOPBACK ioctl causes the enet_dlpi_data->nollopback_flg to be set to the value specified in the ioctl parameter.

Transmission of Message Blocks

The message block transmission has two paths in the sample implementation. The regular data path uses the DL_UNITDATA_REQ primitive and the fast path. The regular path is defined in the DLPI standards. The fast path uses DLPI_IOC_HDR_INFO ioctl to set up the path and is an HP extension to the DLPI standard.

Regular Data Path

The regular data path message transmission works as follows. The streams “put” function enet_wput() receives the DL_UNITDATA_REQ primitive request from the application to send a message to a destination specified in the unitdata message. The enet_wput() function calls the enet_dlpi_unitdata_out() function to service the request. The enet_dlpi_unitdata_out() function applies sanity checks for the stream’s DLPI state and request parameters and builds the LLC header. The LLC header message block is linked with the first M_DATA (with DL_UNITDATA_REQ) and calls the driver’s output routine enet_hw_req().

Fast Path

For better performance, fast path is used to transmit and receive data. The DLPI user sends DLPI ioctl DLPI_IOC_HDR_INFO to set up the fast path on the stream. The DLPI builds an LLC header template and sends it back to the user. For an outbound packet, the user prepends the link header to the data, based on the template of the link header, and sends M_DATA messages to DLPI. DLPI passes this packet to the network driver without building the link header. For an inbound packet on the fastpath stream, DLPI strips off the LLC header and passes it to the user without building and prepending the DL_UNITDATA_IND primitive to the data.

Reception of Message Blocks

The message is received by the enet_dlpi_mblk_intr() function that was passed to the driver along with the stream queue pointer. The following sanity checks are applied:

- Drop multicast packets for which there is no enabled multicast address.
- If DL_PROMISC_MULTI is at promiscuous level and stream is in state DL_UNBOUND, discard unicast packets.
- If DL_PROMISC_SAP, discard packets not destined for stream’s network interface.

This function calls enet_dlpi_unitdata_in() or enet_dlpi_fast_in(), based on whether fast path is set or not.

The enet_dlpi_unitdata_in() routine allocates an M_PROTO message block and builds a DL_UNITDATA_IND primitive from the LLC header in the M_DATA message received from the driver. The LLC header is stripped off the M_DATA message, and this block is linked to unitdata message and sent to the application.

The function implemented in enet_dlpi_fast_in() was discussed in “Fast Path” earlier in this section.

DLPI Primitives and IOCTLs

The following Table 6-6, “DLPI Primitives and IOCTLs,” summarizes the DLPI primitives and IOCTLs that have been dealt with in the sample drivers, along with appropriate comments. The processing of most DLPI primitives and IOCTLs involves driver interaction, which is discussed in the “Driving the NIC” section.
### Table 6-6  DLPI Primitives and IOCTLs

<table>
<thead>
<tr>
<th>DLPI Primitive or IOCTL</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DLPI PRIMITIVES DEFINED BY DLPI 2.0</strong></td>
<td></td>
</tr>
<tr>
<td>DL_ERROR_ACK</td>
<td>Information reporting</td>
</tr>
<tr>
<td>DL_INFO_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_INFO_ACK</td>
<td></td>
</tr>
<tr>
<td>DL_ATTACH_REQ</td>
<td>Attach</td>
</tr>
<tr>
<td>DL_DETACH_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_ERROR_ACK</td>
<td></td>
</tr>
<tr>
<td>DL_OK_ACK</td>
<td></td>
</tr>
<tr>
<td>DL_BIND_REQ DL_BIND_ACK</td>
<td>Bind</td>
</tr>
<tr>
<td>DL_ERROR_ACK DL_OK_ACK</td>
<td></td>
</tr>
<tr>
<td>DL_SUBS_BIND_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_SUBS_BIND_ACK</td>
<td></td>
</tr>
<tr>
<td>DL_SUBS_UNBIND_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_UNBIND_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_DISABMULTI_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_ENABMULTI_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_ERROR_ACK</td>
<td></td>
</tr>
<tr>
<td>DL_GET_STATISTICS_REQ DL_OK_ACK DL_PHYS_ADDR_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_PROMISCOFF_REQ DL_PROMISCON_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_UNITDATA_IND DL_UNITDATA_REQ</td>
<td>DLPI Ver 2.0 Connection less Data transfer</td>
</tr>
<tr>
<td><strong>HP EXTENDED DLPI PRIMITIVES</strong></td>
<td>(These are HP extensions to DLPI 2.0 and may change. They are defined in <em>sys/dlpi_ext.h</em>)</td>
</tr>
<tr>
<td>DL_HP_HW_RESET_REQ</td>
<td>Hardware reset. Used by enetadmin.</td>
</tr>
<tr>
<td>DL_HP_HW_STATUS_REQ</td>
<td>Get hardware status req</td>
</tr>
<tr>
<td>DL_HP_MULTICAST_LIST_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_HP_PPA_REQ</td>
<td>Used by commands, enetadmin, enetlikloop, etc.</td>
</tr>
<tr>
<td>DL_HP_RESET_STATS_REQ</td>
<td>Reset statistics. Used by enetadmin.</td>
</tr>
<tr>
<td><strong>(These are HP specific IOCTLs and may change. They are defined in <em>sys/mci.h</em>)</strong></td>
<td></td>
</tr>
<tr>
<td>MC_GET_MIB_REQ</td>
<td>Get MIB statistics</td>
</tr>
<tr>
<td>MC_SET_MIB_REQ</td>
<td>Set MIB statistics</td>
</tr>
<tr>
<td><strong>HP IOCTLs</strong></td>
<td></td>
</tr>
<tr>
<td>DLPI_IOC_DRIVER_OPTIONS</td>
<td>To get driver features.</td>
</tr>
</tbody>
</table>
Driving the NIC

This section briefly explains the code flow of the lower part of the driver. This portion of the driver handles device interrupts, sends and receives frames, handles control requests from the upper part that require interaction with the device, and so forth. The objective here is to present the code flow of the sample driver enet as a background to the sample driver code.

Data Structures

tenet_ift_t

This structure holds network interface PCI information, register addresses, transmit and receive buffers and descriptors, driver state, and MIB statistics. This structure also embeds an enlan_ift structure that holds generic LAN information pertaining to this interface. The following shows the structure organization:

typedef struct enet_ift {
  enlan_ift t lancift;
  /***************************************************************/
  * PCI Configuration information - PCI CONF
  /***************************************************************/
  ... 
  * PCI Control and Status registers. Each field contains the
  * HPA + offset for the network contlr. registers - DEV REG
  /***************************************************************/
  ...
  ... 
  * Device Specific Section - DEV SPEC
  /***************************************************************/
  struct isc_table_type *isc;
  enet_srom_t *srom;/* Serial ROM layout*/
  ubit32 drv_state;/* Driver state info*/
  ubit32 reset_state;/* Driver reset state*/
  ...
  ...
  /***************************************************************/
  * Transmit Section - TX SECT
  /***************************************************************/
  enet_tb_t *tbr; /* Transmit buffer Ring */
  enet_td_t *tdr; /* Transmit Descriptor Ring */
  void *tdr_DMA_handle; /* DMA handle for Tx-desc ring */
  ...
  ...
  /***************************************************************/
  * Receive Section - RX SECT
  /***************************************************************/
  enet_rd_t *rdr; /* Receive Descriptor Ring */
  enet_rb_t *rbr; /* Receive buffer Ring */
  void *rdr_DMA_handle; /* DMA handle for Rx-desc ring */
  ...
  ...

Table 6-6 DLPI Primitives and IOCTLs (Continued)

<table>
<thead>
<tr>
<th>DLPI Primitive or IOCTL</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>DLPI_IOC_HDR_INFO</td>
<td>To get LLC header for fast path.</td>
</tr>
<tr>
<td>DLPI_SET_NOLOOPBACK</td>
<td>Do not loopback the message.</td>
</tr>
</tbody>
</table>
.../* Full Duplex, speed and Transmit Threshold setting - SETTINGS */...

/* Local Driver Receive Stats - STATS */
rcv_stats_t rstats;/* Receive Statistics*/

/* Local Driver Transmit Stats - STATS */
trx_stats_ttstats;/* Transmit Statistics*/

/* Mib Specific Section */
mib_xEntrymib_xstats;
mib_Dot3StatsEntrydot3_ext_stats;
mib_Dot3CollEntrydot3_ext_coll;

/* Misc */

/* lock_t*enet_r_lock; */
enet_ift_t * next; /* pointer to the next interface */

} enet_ift_t;

For a description of the data fields, refer to Table 6-7, “enet_ift Data Fields.”

<table>
<thead>
<tr>
<th>Field Name/Generic Description</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>lancift</td>
<td>Contains generic LAN information.</td>
</tr>
<tr>
<td>PCI INFO</td>
<td>Has PCI configuration information.</td>
</tr>
<tr>
<td>DEV REG</td>
<td>Fields have Control and Status Register addresses.</td>
</tr>
<tr>
<td>DEV SPEC</td>
<td>Device specific information, such as ISC structure, serial ROM data, driver states and cable state.</td>
</tr>
<tr>
<td>TX SECT: tbr, tdr</td>
<td>This set of fields contains transmit buffers, transmit descriptors and counters.</td>
</tr>
<tr>
<td>RX SECT: rbr, rdr</td>
<td>This set of fields contains receive buffers, receive descriptors and counters.</td>
</tr>
<tr>
<td>SETTINGS</td>
<td>Full duplex, link speed, selected connection type and transmit threshold settings.</td>
</tr>
<tr>
<td>STATS</td>
<td>Driver local receiver and transmitter statistics.</td>
</tr>
<tr>
<td>mib_xstats</td>
<td>MIB objects (RFC 1066/1156) and additional counters.</td>
</tr>
<tr>
<td>dot3_ext_stats</td>
<td>Extended MIB statistics</td>
</tr>
<tr>
<td>dot3_ext_coll</td>
<td>Extended MIB collisions</td>
</tr>
</tbody>
</table>
Table 6-7 enlan_ift Data Fields (Continued)

<table>
<thead>
<tr>
<th>Field Name/Generic Description</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupt object</td>
<td>Contains driver interrupt information.</td>
</tr>
<tr>
<td>enet_r_lock</td>
<td>Lock for accessing enet_ift</td>
</tr>
</tbody>
</table>

enlan_ift

This structure holds generic LAN information for the network interface. It is shown here, and Table 6-8, “enlan_ift Data Fields,” explains the fields.

typedef struct{
    hw_ift_t hwift;
    lan_timer lan_timer;
    int ptr_t (*hw_req)();
    int (*dma_time)();

    /* Status and statistics Data Area - STATUS & STAT*/
    uint32_t BAD_CONTROL;
    uint32_t UNKNOWN_PROTO;
    uint32_t RXD_XID;
    uint32_t RXD_TEST;
    uint32_t RXD_SPECIAL_DROPPED;
    short int is_scaninterval;

    /* Configuration info */
    int num_multicast_addr;
    int broadcast_filter;
    int multicast_filter;
    enlanc_promisc_type_t promiscuous_filter;
    int hw_initialized;
    uint8_t mcast[96];
    uint32_t mcast_ref_cnt[16];
    mib_xEntry *mib_xstats_ptr;
    lock_t *enlanc_lock;
} enlan_ift;

Table 6-8 enlan_ift Data Fields

<table>
<thead>
<tr>
<th>Field Name/Generic Description</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>hwift</td>
<td>Generic Hardware information.</td>
</tr>
<tr>
<td>lan_timer</td>
<td>DMA/Control timer to track if a DMA or control operation is taking too long.</td>
</tr>
<tr>
<td>hw_req()</td>
<td>h/w interface request function pointer.</td>
</tr>
<tr>
<td>dma_time()</td>
<td>DMA timeout error handling.</td>
</tr>
<tr>
<td>STATUS &amp; STAT</td>
<td>More statistics</td>
</tr>
<tr>
<td>num_multicast_addr</td>
<td>Number of multicast addresses active.</td>
</tr>
</tbody>
</table>
The logged_link for each bind, the network driver keeps track of the bound SAPs and relevant information about the bind. The following structures are used to maintain this information:

```
struct logged_info{
    int protocol_val[5];
    caddr_t ift_ptr;
    queue_t *q_ptr;
    int flags;
};
```

Refer to Table 6-9, "Bound SAP Data Fields," for additional information.

### Table 6-9  Bound SAP Data Fields

<table>
<thead>
<tr>
<th>Field Name/Generic Description</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>broadcast_filter, multicast_filter, promiscuous_filter</td>
<td>Read packet filters</td>
</tr>
<tr>
<td>mcast, mcast_ref_cnt</td>
<td>Multicast addresses and their reference count.</td>
</tr>
<tr>
<td>mib_xstats_ptr</td>
<td>MIB object</td>
</tr>
<tr>
<td>enlanc_lock</td>
<td>Lock to access enlanc_ift</td>
</tr>
</tbody>
</table>

The following structure is used to link the logged_infos.

```
struct logged_link{
    struct logged_link *next;
    struct logged_info log;
};
```

### Control Functions

The function enet_dlpi_control() communicates the device dependent DLPI primitives to the network interface part of the driver for further processing. Essentially, the DLPI function calls the respective functions passing per instance, driver control structure enet_ift, the ioctl command, and the message block with request data.

The following subsections summarize the driver control commands and the processing by the network driver.
DL_HP_ENABMULTI

`ext_mcast_list[]` is a global array, where each element of the array is an `hw_mcast_entry_t` structure corresponding to a particular interface.

```c
typedef struct {
    mcast_list_t *hw_mcast;
    int    mc_threshold;
    /* Threshold for mcast addresses */
} hw_mcast_entry_t;
```

`hw_mcast` points to a linked list of `mcast_list_t` structures which hold multicast addresses enabled on an interface.

```c
typedef struct mcast_list {
    uint8_t addr[6];
    /* Multicast address */
    int    ref_cnt;
    /* Number of times the multicast address has been enabled */
    struct mcast_list *next;
    /* pointer to next structure */
} mcast_list_t;
```

The `enet_dlp_control()` calls `enet_media_control()` function to process `DL_HP_ENABMULTI` command. `enet_media_control()` function checks validity of multicast address and calls macro `ENET_UPDATE_EXT_MCAST` to update the entry for the multicast address.

This macro checks all multicast entries for the interface. If there is already an entry of the requested multicast address, then it just updates `ref_cnt` and returns.

If the requested multicast address is not there in the list, then `enet_media_control()` calls `ENET_ADD_EXT_MCAST` macro.

This macro allocates an `mcast_list_t` structure, assigns the requested multicast addr to `addr[]`, initializes `ref_cnt` to 1 and then adds `mcast_list_t` (multicast entry) to a linked list of multicast entries for the interface.

Then, the macro calls `enet_hw_req()` to enable the requested multicast address on the device.

DL_HP_DISABMULTI

`enet_dlp_control()` calls `enet_media_control()` function to process the `DL_HP_DISABMULTI` command. `enet_media_control()` function calls the `ENET_DEL_EXT_MCAST` macro.

This macro gets `mcast_list_t` structure (multicast entry) for the requested multicast address and decrements `ref_cnt` of the structure. If `ref_cnt` becomes zero, then `mcast_list_t` (multicast entry) is removed from the linked list.

Then, `ENET_DEL_EXT_MCAST` calls `enet_hw_req()` to remove the multicast address from the device.

DL_HP_PROMISCON

`enet_promisc_list[]` a global array; each array element is a `p_entry_t` structure corresponding to an interface.

```c
typedef struct {
    int (*func_ptr) __((struct lan_ift *,
                        void *, void *, u_int));
    /* Function to call for promiscuous packets */
    caddr_t     data_ptr;
    /* queue pointer of the promiscuous stream */
    uint32_t    filter_cnt;
} p_entry_t;
```
/* ref cnt for SAP-based request to filter */
uint32_t no_filter_cnt;
/* ref cnt for requests to receive all pkts */
uint32_t phys_ref_cnt;
/* ref cnt to enable phys promisc */
uint32_t multi_ref_cnt;
/* ref cnt to enable multi promisc */
uint32_t sap_ref_cnt;
/* ref cnt to enable sap promisc */
}

The `enet_dlpi_control()` calls `enet_media_control()` function to process the `DL_HP_PROMISCON` command. `enet_media_control()` function updates related fields in the `p_entry_t` structure and calls `enet_hw_req()` to enable a specific promiscuous level on the device.

Currently only one stream can be in promiscuous mode per interface. Refer to the ENET driver example source.

**DL_HP_PROMISCOFF**

`enet_dlpi_control()` calls `enet_media_control()` function to process the `DL_HP_PROMISCOFF` command. `enet_media_control()` updates related fields in the `p_entry_t` structure and calls `enet_hw_req()` to disable promiscuous mode on the device. Refer to the ENET driver example source.

**DL_HP_SET_PHYS_ADDR**

Driver calls `enet_media_control()` to `enet_hw_req()` which in turn calls `enet_ctrl_req()` to change the local address.

**DL_HP_RESET_STATS**

The functions called are `enlanc_media_control()`, `enet_hw_req()`, `enet_ctl_req()`, and `enet_ext_clearmib()` to clear MIB.

**DL_HP_HW_RESET**

The following functions are called in order: `enlanc_media_control()`, `enet_hw_req()`, `enet_ctl_req()`, and `enet_reset()` to perform hardware reset.

**Datapath**

**Outbound Path**

The `enet` driver write path starts with the function `enet_dlpi_unitdata_out`. Also refer to Figure 6-5, “Control Flowchart for Outbound Path.”
enet_dlpi_unitdata_out()

This function calls enet_hw_req() to handle the write request.

enet_hw_req()

All LAN_REQ_WRITE write requests and LAN_REQ_WRITE_L loopback write requests are processed when the driver state is ENET_ONLINE. Otherwise, only loopback write requests are processed and other write requests are discarded.

Non-loopback unicast packets are transmitted in the fast path by calling ENET_TRANSMIT_FRAME. Multicast, broadcast, self addressed frames, frames < 14 bytes, and frames with buffers > ENET_MAX_BUF_PER_FRAME are handled in the slow path by calling enet_slow_hw_req().

Non-write requests are passed on to enet_ctl_req().
enet_slow_hw_req()

Non unicast frames are handled in enet_transmit_complt(). If the number of buffers is > ENET_MAX_BUF_PER_FRAME, an attempt is made to copy all the buffers into one, to use only one transmit descriptor and fewer buffer descriptors. The frame is sent by calling ENET_TRANSMIT_FRAME.

ENET_TRANSMIT_FRAME

A check is made to see if transmit buffers are available to send the frame; if not, the frame is dropped. If transmit descriptors are unavailable the buffer is queued up for later transmission. Pending transmits are handled in the transmit complete interrupt. The enet_transmit_complt() routine is called to process transmit complete interrupts. Otherwise, the transmit descriptors are set up and a transmit poll is issued to the device to send out the frame. The device interrupts after all frames waiting transmission on the transmit descriptor list are transmitted. The enet_transmit_complt() routine is called to handle the interrupt.

enet_transmit_complt()

This routine processes transmit complete interrupts. Call enet_slow_complt() to process non-unicast frames or setup frames. Transmit error handling is done by calling the enet_trans_error() routine. If there are frames queued for transmission, call enet_transmit_pended_frames() to restart transmission.

enet_transmit_pended_frames()

While there are frames pending transmission, map the frames, set up the transmit descriptors, and issue a transmit poll to the device to transmit the frames.

Inbound Path

For additional information, refer to Figure 6-6, “Control Flowchart for Inbound Path.”
Figure 6-6  Control Flowchart for Inbound Path

The *enet* read path is on the ICS. The *enet_isr()* routine is called when the network interface’s PCI interrupt is received and the *enet_receive_frame()* routine is invoked to process received frames.

**enet_receivekts()**

This function is called from the receive interrupt handler. Some sanity checking is done on the received frames to determine if they are good. The message block chain is constructed from the receive descriptor. If the driver state is ENET_ONLINE, call the *enet_process_packet()* routine to process the frame. Otherwise, call the *enet_process_looper()* routine to process the frame. Replenishing the receive descriptor ring with buffers is done while doing frame receive processing_p.

**enet_process_packet()**

This function determines the frame header is Ethernet or IEEE 802.2 and *enlanc_ether_ics()* or *enlanc_802_2_ics()* is called, accordingly.
enet_process_looper()

This function processes the loopback packet. The current driver sub-state determines the action taken. The packet buffer is validated but not used, and discarded.

enet_802_2_ics()

The packet type (802.2 or 802 SNAP), protocol kind (LAN_TYPE, LAN_SNAP or LAN_SAP), and protocol value are extracted from the received packet. If the interface supports promiscuous mode and it is set, route the packet to all streams qualified for the set promiscuous level using the enet_route_promisc() routine. The lookup for logged DLSAPs is enet_sap_lookup(), and if there is a match, this routine sends the packet to the logged stream (by calling the function registered during the bind). XID and TEST packets are processed in enet_802_2_test_ctl().

enet_ether_ics()

Protocol kind (LAN_TYPE, LAN_SNAP or LAN_SAP) and protocol value are extracted from the received packet. If the interface supports promiscuous mode and is set, route the packet to all streams qualified for the set promiscuous level using the enet_route_promisc() routine. The lookup for logged DLSAPs is enet_lookup() and if there is a match, this routine sends the packet to the logged stream by calling the function registered during the bind.

Inbound Promiscuous Handling

Depending on the promiscuous level set, the device receives the packets not destined to the interface.

For ex, the device receives all the packets on the wire for PROMISC_PHYS and all multicast, broadcast packets if PROMISC_MULTI is set and under normal operation for PROMISC_SAP.

Bound Promiscuous Stream

The bound promiscuous stream sends packets through the same path as normal packets. For example: enet_ether_ics(), or enet_802_2_ics(), calls enet_sap_lookup() to look for the stream matching the destination SAP of the packet. The packet is then passed to the stream.

Unbound Promiscuous Stream

enet_ether_ics() and enet_802_2_ics() call enet_route_promisc().

The enet_route_promisc() gets the promiscuous stream’s queue pointer from the p_entry_t structure for the interface.

For PROMISC_SAP, enet_route_promisc() passes only those packets to the stream whose destination SAP matches with any SAPs enabled on the interface.

For PROMISC_PHYS and PROMISC_MULTI, enet_route_promisc() passes all the packets to the promiscuous stream because the device already has filtered the packets.

Outbound Promiscuous Handling

The ENET_TRANSMIT_FRAME macro routes all the packets to enet_ether_ics() and enet_802_2_ics() for outbound promiscuous.

Bound Promiscuous Stream

The bound promiscuous stream sends packets through the same path as normal packets. For example: enet_ether_ics(), or enet_802_2_ics(), calls enet_sap_lookup() to look for the stream matching the destination SAP of the packet. The packet is then passed to the stream.
Unbound Promiscuous Stream

`enet_ether_ics()` and `enet_802_2_ics()` call `enet_route_promisc()`.

The `enet_route_promisc()` gets the promiscuous stream's queue pointer from the `p_entry_t` structure for the interface.

For `PROMISC_SAP`, `enet_route_promisc()` passes only those packets to the stream whose source SAP matches with any SAPs enabled on the interface.

For `PROMISC_MULTI`, only multicast and broadcast packets are passed to the stream.

For `PROMISC_PHYS`, all packets are passed to the stream.

Interrupt Service Routine — `enet_isr()`

The `enet_isr()` handles the interrupt generated by the NIC. It can also be invoked by the kernel when any other device (which shares the same interrupt resource as the NIC) generates the interrupt.

The `enet_isr()` must check if the interrupt is generated by the NIC before processing the interrupt. If it is not generated by the NIC, then `enet_isr()` should return zero. The zero value indicates to the kernel that the interrupt is generated by the other device.

The `enet_isr()` can be called even when the NIC is suspended, see Chapter 15, “On-Line Addition/Replacement.” This is due to interrupts generated by other devices which share the same interrupt resource. Therefore, `enet_isr()` must verify that the NIC is online before accessing any card register (to check if the interrupt is generated by that card). If the NIC is suspended, then `enet_isr()` must return zero. The zero value indicates to the kernel that the interrupt is generated by the other device.

Releasing any Pending Timeouts

Before the driver gets suspended during an OLA/R event or before the driver is unloaded in a DLKM operation, the driver shall be free of any pending callback routines. For more information on OLA/R and DLKM, refer to Chapter 15, “On-Line Addition/Replacement.”

ENET driver maintains a list of pending timeout routines. On an OLA/R suspend event or during a DLKM unload, the driver calls `untimeout()` on all the pending timeout entries in the timeout list.

Following enum is a field in `enet_ift` structure which saves this information.

```c
enum {
    ENET_SEND_LOOP_PKT_TIMEOUT = 1 << 0,
    ENET_AUTO_NEG_TIMEOUT      = 1 << 1,
    ENET_FORCE_SPEED_DUPLEX_TIMEOUT = 1 << 2,
} timeout_list;
```

Each flag in `enum` refers to a function that can be on the timeout list. The flag for the function which is called through `timeout()` is set until the function passed to timeout is called.
Platform Specifics

Interface drivers are supposed to take care of platform dependencies so one object can run on any HP platform. The CDIO in the I/O subsystem provides a consistent view of HP-UX platforms to drivers by hiding the platform dependencies as much as possible. Some newer platforms, such as the V Class, have dependencies that require special coding.

DMA Mapping

On the transmit side, packets that are passed to the driver from upper layers may cross a page boundary in virtual address space, and a page-crossing buffer may not be contiguous in physical address space. In the “hints” argument to DMA mapping service `wsio_map_dma_buffer()`/`wsio_map()`, if `WSIO_DMA_CONTIGUOUS/IO_CONTIGUOUS` is specified, then the DMA mapping service tries to map the buffer to a contiguous IOVA range.

On noncoherent systems, it is possible to map physically non-contiguous buffers to a contiguous IOVA range. But on noncoherent systems, IO devices must directly access physical memory. Thus, it is not possible to map the non-contiguous physical buffer with hints `WSIO_DMA_CONTIGUOUS/IO_CONTIGUOUS` on such systems.

To use a single driver source for both coherent and noncoherent systems, `WSIO_DMA_CONTIGUOUS/IO_CONTIGUOUS` hint should not be specified if the driver is expected to be passed with non-contiguous buffers.

For a detailed information on cache coherence issues, refer to Chapter 2, “HP-UX I/O Subsystem Features.”

V Class

The following brief overview of the V class PCI I/O architecture provides a good background for driver writers porting a driver to V Class Systems.

EPIC is the bridge between the PCI bus and processors, memory, and interconnections. Two types of host memory are accessible by an I/O card DMA transaction: non-coherent shared memory on the EPIC bridge, and channel based access to coherent system memory. Multiple channels are available to PCI slots or card functions. Driver instances related to different slots will not share a DMA channel or steal resources from each other. The I/O card cannot access any non-coherent address space beyond EPIC.

The driver model for EPIC expects that all control structures are small and stored in EPIC shared memory. All application data is assumed to be in buffers in coherent system memory. These buffers are read or written as part of DMA stream. Outbound prefetch is initiated when a buffer is mapped for an I/O card’s DMA access (since it is in coherent memory, it can be prefetched).

WSIO mapping calls work the same way on V Class platforms as on other platforms. EPIC CDIO (accessed via WSIO) will not reassign an IOVA range until all mappings within the channel have been released, so one must be careful with long term mappings.

The following points are useful while writing PCI network drivers for V class.

- Allocate transmit and receive descriptor memory in shared memory with the `wsio_allocate_shared_mem()` function.
- Shared memory does not need `wsio_map()` it is already both virtually and physically contiguous.
The following code examples illustrate the use of the function `wsio_allocate_shared_mem()`.

```c
/* This code illustrates the use of shared memory to allocate
 * a transmitter buffer ring for a V_CLASS system network
 * controller. Refer to the sample driver enet.c for more
 * details.
 * Look for #ifdef V_CLASS or if(is_SPP()) statements.
 */
static int
enet_init (struct isc_table_type *isc) {
    enet_ift_t   *enet_iftp;
    size_t       size;
    u_long       phys_base;
    ...

    /* size: initialized to the size of enet_iftp->tdr
       * (transmitter descriptor ring)
       */
    enet_iftp->tdr_DMA_handle = wsio_allocate_dma_handle(isc)

    /* Allocate shared memory for Tx-descriptor ring */
    if( wsio_allocate_shared_mem(isc,
        enet_iftp->tdr_DMA_handle,size,
        (caddr_t *) & phys_base,
        (caddr_t *) & enet_iftp->tdr,
        0) != WSIO_MAP_OK)
        msg_printf(*enet - TDR allocation failed...
    )
        return -1;
    }

    ...
}
```
Network Management Support

Hewlett-Packard’s implementation of MIBs and the access methods to MIB information from HP-UX version 10.00 and previous releases has been monolithic in nature; all MIB support was directly done in kernel. This approach forced Hewlett-Packard to constantly change the kernel to incorporate new MIB instrumentation when new links or drivers, either supplied by Hewlett-Packard or a third party, were added.

Hewlett-Packard moved from a single monolithic agent to a variable number of agents, called subagents. Whenever a new driver is added to a system, a user space subagent specific to this driver is also supplied. This subagent provides the MIB instrumentation needed to access the MIB objects associated with the driver. Figure 6-7, “Master Agent/Subagents Relationship,” shows the master agent/subagent relationship and partitioning of the subagents. The assumption now is that whoever supplies the new driver will also supply the subagent for that driver.

Figure 6-7 Master Agent/Subagents Relationship

An SNMP manager only communicates with the master agent, and the master agent sends requests to the appropriate subagent(s). The subagent(s) reply to the master agent, which replies to the SNMP manager.

The new Network Management interface will be user based, contained completely within a user space library (libnm.a) and in general, will have a one-to-one mapping to the calls provided by the old Network Management Interface.
When replacing the /dev/netman, the following ioctls will not be available:

- NMIOGET
- NMIOSET
- NMIODEL
- NMIOCRE
- NMPEEK
- NMPOKE

Network Management services are to be used by STREAMS based networking interfaces that provide an ifEntry in the MIB-II ifTable object (see the following sample code for ifEntry struct in sys/mib.h.) In the 4.3 Berkeley based networking stack, the ifTable was directly tied to the global ifnet structure list. When a networking interface registered an ifnet structure via if_attach, an ifIndex value was returned which was to be used in the MIB-II ifEntry object managed by that networking interface. The ifTable was known by the TCP/IP protocol stack and all interfaces to retrieve the ifTable and a specific ifEntry was through the TCP/IP protocol stack. With the movement to a STREAMS based TCP/IP protocol stack, the global ifnet structure list no longer exists and, therefore, the global ifTable management no longer exists.

Even though in the STREAMS based networking environment the ifTable is not globally managed, each ifEntry in the ifTable must have a unique ifIndex value so the ifTable can be created. Therefore, the ifIndex values must be globally managed. Along with managing the ifIndex values, the MIB-II ifNumber object must also be managed. The Network Management services described next are for retrieving and returning a unique ifIndex value.

```
u_int32 get_nmid()
```

Allocates a system unique ifIndex value for use in the MIB-II ifEntry object. Any kernel entity that required an entry in the ifTable should use this service for retrieving the value of the ifIndex field.

- > 0 indicates the call succeeded and the value returned is the ifIndex value.
- <=0 indicates the request failed to allocate an ifIndex value.

Example code in enet driver:
```
enet_iftp->lancift.hwift.nm_id = get_nmid();
```

```
u_int32 return_nmid
```

Return a previously assigned ifIndex to the pool of available ifIndex values. This network management service should be called by all kernel entities that own an ifIndex value before it is unloaded from the system.

```
u_int32 return_nmid (u_int32 ifIndex)
```

The ifIndex value to be returned to the pool of available ifIndex values.

- <0 Indicates the ifIndex value being returned was not the previously assigned ifIndex value
- >=0 Indicates the ifIndex was successfully returned to the pool.

In sys/mib.h, mib_ifEntry is defined as:

```c
typedef struct {
int ifIndex;
char ifDescr[64];
int ifType;
inu_int ifMtu;
gauge ifSpeed;
mib_physaddr_t ifPhysAddress;
int ifAdmin;
```
int ifOper;
TimeTicks ifLastChange;
counter ifInOctets;
counter ifInUcastPkts;
counter ifInNUcastPkts;
counter ifInDiscards;
counter ifInErrors;
counter ifInUnknownProtos;
counter ifOutOctets;
counter ifOutNUcastPkts;
counter ifOutDiscards;
counter ifOutErrors;
gauge ifOutQlen;
int ifSpecific;
} mib_ifEntry;

The device driver's job is to fill out the fields in the struct mib_ifEntry in the appropriate order. Any application can then retrieve information for use by the Network Management Support services interface.
The HP-UX LAN commands provide the user with the ability to scan and administer LAN interfaces on an HP-UX system. In addition, HP-UX LAN commands can be used for troubleshooting purposes, such as verifying datalink level connectivity between two LAN interfaces.

For HP-UX 11i v1.0, there are three main HP-UX LAN commands:

1. `lanscan` — Used to determine the LAN interfaces installed in the system.
2. `lanadmin` — Used for managing a LAN interface, getting LAN interface statistics.
3. `linkloop` — Used for troubleshooting purposes and verifying data-link level connectivity.

Of the three commands listed, only the `lanscan` command recognizes and works with the HP and the Independent Hardware Vendor (IHV) network interface drivers. The `lanadmin` and `linkloop` commands, only work with HP network interface drivers and do not recognize IHV drivers. IHVs network driver developers are expected to provide their own `lanadmin` and `linkloop` commands and any associated shared libraries. As a part of the ENET sample network driver, fully functional sources are provided for these commands under the `enet/misc/cmds` subdirectory.
LANSCAN Command

The `lanscan` command may be used to display the list of LAN interfaces that are installed on a system. The `lanscan` commands display all the network interface drivers installed on a system, this includes the HP drivers as well as non-HP drivers. The `lanscan` commands communicates with HP-DLPI and gets the list that is displayed to the user. This also means that all IHV drivers must register with HP-DLPI during their initialization.

The HP-UX `lanscan` command displays a list of all LAN interfaces, irrespective of the type of driver (HP or IHV driver) or underlying physical medium (Ethernet, Fiber Channel, Token Ring and so forth) that is installed on a system. The `lanscan` command retrieves the list of LAN interfaces by communicating with HP-DLPI. For any LAN interface to appear in the `lanscan` output, the network interface driver for the LAN interface must be registered with HP-DLPI. For further details of the registration process, refer to Chapter 6, "Creating Networking Device Drivers."

The `lanscan` output contains the following information displayed in a fixed field format:

- Hardware path
- Active station address
- Interface instance number
- Hardware state
- Network interface
- Network management ID
- MAC type
- Status, whether HP-DLPI is supported or not
- Extended station address and encapsulation methods displayed under verbose output

The diagrams shown in Figure 7-1, "HP LANSCAN Command," depicts how the `lanscan` commands fit in the HP-UX networking stack.
LANADMIN Command

The HP-UX `lanadmin` command is used to manage various network related settings for a LAN interface. The `lanadmin` command can be used to display and modify LAN interface settings, such as MAC/Physical address, MTU, speed, and so forth. The `lanadmin` command that is shipped with HP-UX 11i v1.0 operating system does not work with IHV network interface drivers. It is the responsibility of the IHV network driver developer to provide a command named, `driveradmin`, along with the driver. As an example, for the ENET sample driver, fully functional sources are provided for `enetadmin` command. IHV drivers may modify `enetadmin` sources to get it to work with their specific network driver.

The IHV drivers, must implement the following `ioctl`s and primitives. Refer to Chapter 6, “Creating Networking Device Drivers,” for detailed information about writing network drivers.

The diagram shown in Figure 7-2, “HP and IHV LANADMIN Commands,” shows how the HP-UX `lanadmin` command and an IHV `driveradmin`, fits within the HP-UX networking stack. ENET is used as a representation of an IHV network interface driver.

**Figure 7-2** HP and IHV LANADMIN Commands
The following sections show LAN interface attributes which may be displayed or modified using the HP-UX lanadmin or driveradmin command.

**Displaying a Physical Address**

The lanadmin command allows users to display the current physical address of an interface by using the -a option. The network interface driver must implement the DL_PHYS_ADDR_REQ primitive.

```
REQ: DL_PHYS_ADDR_REQ
ACK: DL_PHYS_ADDR_ACK
```

Usage:

```
lanadmin -a <ppa>
```

Example:

```
$ lanadmin -a 0
Station Address = 0x00306e1cb1f6
```

**Changing a Physical Address**

The lanadmin command allows users with super-user privilege to modify the current physical address of an interface by using the -A option. The network interface driver must implement the DL_SET_PHYS_ADDR_REQ primitive.

```
REQ: DL_SET_PHYS_ADDR_REQ
ACK: DL_OK_ACK
```

Usage:

```
lanadmin -A value <ppa>
```

Example:

```
$ lanadmin -A 0x00047639e284 6
Old Station Address = 0x00047639e283
New Station Address = 0x00047639e284

$ lanadmin -A DEFAULT 6
Old Station Address = 0x00047639e284
New Station Address = 0x00047639e283
```

**Displaying Source Routing Option**

The lanadmin command can be used to display source routing option. This feature may only be used with Token Ring network interfaces. The lanadmin command sends the DLPI_GET_SRC_ROUTE_FLAG ioctl to fetch the information from the driver.

```
IOCTL: DLPI_GET_SRC_ROUTE_FLAG
```

Usage:

```
lanadmin -b <ppa>
```

This option is used only for Token Ring interfaces.

Example:

```
$ lanadmin -b 1
lan1 Source Routing = on
```
Changing Source Routing Option

Users with superuser privilege can use the `lanadmin` command to modify the source routing option. This feature may only be used with Token Ring network interfaces. The `lanadmin` command sends the `DLPI_SET_SRC_ROUTE_FLAG` ioctl to set the information.

```
IOCTL: DLPI_SET_SRC_ROUTE_FLAG
```

Usage:

```
lanadmin -B on|off <ppa>
```

This option is used only for Token Ring interfaces.

Example:

```
$ lanadmin -B off 1
Old lan1 Source Routing = on
New lan1 Source Routing = off
```

Displaying MTU Size

The `lanadmin` command allows users to display the current **Maximum Transmission Unit** (MTU) size of a network interface by using the `-m` option. The network interface driver must implement the `DL_HP_GET_DRV_PARAM_IOCTL`ioctl for the `DL_HP_DRV_MTU` request.

```
IOCTL: DL_HP_GET_DRV_PARAM_IOCTL
IOCTL request: DL_HP_DRV_MTU
```

Usage:

```
lanadmin -m <ppa>
```

Example:

```
$ lanadmin -m 0
MTU Size = 1500
```

Changing MTU Size

Users with superuser privilege can set the MTU size of an interface by using the `-M` option of the `lanadmin` command. The network interface driver must implement the `DL_HP_SET_DRV_PARAM_IOCTL` ioctl for the `DL_HP_DRV_MTU` request.

```
IOCTL: DL_HP_SET_DRV_PARAM_IOCTL
IOCTL request: DL_HP_DRV_MTU
```

Usage:

```
lanadmin -M value <ppa>
```

Example:

```
$ lanadmin -M 1200 0
Old MTU Size = 1500
New MTU Size = 1200
```
Resetting MTU Size

Users with superuser privilege can reset the MTU size of a network interface to the default value by using the \(-R\) option of the \texttt{lanadmin} command. The network interface driver must implement the \texttt{DL_HP_SET_DRV_PARAM_IOCTL} ioctl for the \texttt{DL_HP_DRV_RESET_MTU} request.

\begin{verbatim}
IOCTL: DL_HP_SET_DRV_PARAM_IOCTL
IOCTL request: DL_HP_DRV_RESET_MTU
\end{verbatim}

Usage:

\texttt{lanadmin -R <ppa>}

Example:

\$ lanadmin -R 0

\begin{verbatim}
Old MTU Size = 1200
New MTU Size = 1500
\end{verbatim}

Displaying Link Speed

The \texttt{lanadmin} command allows users to display the current speed of an interface by using the \(-s\) option. For this to work, the interface driver must implement the \texttt{DL_HP_GET_DRV_PARAM_IOCTL} ioctl for a \texttt{DL_HP_DRV_SPEED} request.

\begin{verbatim}
IOCTL: DL_HP_GET_DRV_PARAM_IOCTL
IOCTL request: DL_HP_DRV_SPEED
\end{verbatim}

Usage:

\texttt{lanadmin -s <ppa>}

Example:

\$ lanadmin -s 0

\begin{verbatim}
Speed = 100000000
\end{verbatim}

Changing Link Speed

Users with superuser privilege can change the speed value of an interface by using the \(-S\) option of the \texttt{lanadmin} command. The interface driver must implement the \texttt{DL_HP_SET_DRV_PARAM_IOCTL} ioctl for a \texttt{DL_HP_DRV_SPEED} request.

\begin{verbatim}
IOCTL: DL_HP_SET_DRV_PARAM_IOCTL
IOCTL request: DL_HP_DRV_SPEED
\end{verbatim}

Usage:

\texttt{lanadmin -S <ppa>}

Example

\$ lanadmin -S 10000000 0

\begin{verbatim}
Old Speed = 100000000
New Speed = 10000000
\end{verbatim}
**MIB-II and Interface Specific Statistics**

The `lanadmin` command allows the users to display interface level MIB-II statistics (RFC 1213) by using the `-g` option. The network interface driver must implement the `DL_GET_STATISTICS_REQ` primitive. Users can also use `lanadmin` to display MIB-II statistics in **Menu Mode**.

```plaintext
REQ: DL_GET_STATISTICS_REQ
ACK: DL_GET_STATISTICS_ACK
```

**Usage:**

```
lanadmin -g <ppa>
```

**Example:**

```
$ lanadmin -g 0
```

```plaintext
LAN INTERFACE STATUS DISPLAY
Mon, Mar 17, 2003 13:55:34
PPA Number = 0
Description = lan0 Intel PCI Pro 10/100Tx Server
Adapter [100BASE-TX,FD,AUTO,
Type (value) = ethernet-csmacd(6)
MTU Size = 1500
Speed = 100000000
Station Address = 0x347c08663
Administration Status (value) = up(1)
Operation Status (value) = up(1)
Last Change = 240
Inbound Octets = 36537058
Inbound Unicast Packets = 4789
Inbound Non-Unicast Packets = 474939
Inbound Discards = 14
Inbound Errors = 0
Inbound Unknown Protocols = 180185
Outbound Octets = 380027
Outbound Unicast Packets = 4605
Outbound Non-Unicast Packets = 14
Outbound Discards = 0
Outbound Errors = 0
Outbound Queue Length = 0
Specific = 655367
Ethernet-like Statistics Group
Index = 1
Alignment Errors = 0
FCS Errors = 387
Single Collision Frames = 0
Multiple Collision Frames = 0
Deferred Transmissions = 0
Late Collisions = 0
Excessive Collisions = 0
Internal MAC Transmit Errors = 0
Carrier Sense Errors = 0
Frames Too Long = 0
Internal MAC Receive Errors = 0
```

The statistics that the Hewlett-Packard LAN drivers maintain and that `lanadmin` displays are the MIB-II statistics defined in RFC 1213. These statistics are common to all Hewlett-Packard LAN links, Ethernet, Token Ring, FDDI, and Fiber Channel. In addition, most Hewlett-Packard LAN drivers maintain the link specific MIB statistics. For example, Hewlett-Packard Ethernet/802.3 drivers maintain the Ethernet-like MIB statistics defined in RFC 1398.
Clearing LAN Interface Statistics Registers

Users with superuser privilege can clear the network interface statistics registers to zero using `lanadmin`. The interface driver must implement the `DL_HP_RESET_STATS_REQ` primitive. This feature is available in Menu Mode only.

```
REQ: DL_HP_RESET_STATS_REQ
ACK: DL_OK_ACK
```

Resetting LAN Interface

Users with superuser privileges can reset a LAN interface card using `lanadmin`. The interface driver must implement the `DL_HP_HW_RESET_REQ` primitive. This feature is available in Menu Mode only.

```
REQ: DL_HP_HW_RESET_REQ
ACK: DL_OK_ACK
```

Shared Library for LANADMIN

It may be necessary to provide a driver-specific shared library for `driveradmin` if a network interface driver provides user access to card-specific attributes, parameters or features. Since there is no standard set of such features, it is up to driver developers to decide which attributes to make accessible to the users. The `driveradmin` command allows users to modify specific features for different cards; this maintains a level of uniformity.

The `driveradmin` command provides support for driver-specific features by `-x` or `-X` options. When `driveradmin` is invoked with the `-x` or `-X` option, it loads the driver-specific shared library and calls the appropriate function to interact with the driver. The length of each keyword passed with an `-x` or `-X` option must not exceed 30 characters, and the total length of all the keywords including the separator characters (a space character) must not exceed 255 bytes. A network interface driver can define its own keywords as arguments for the `-x` and `-X` options.

The name of the shared library must be `libds<driver name>.sl` and `libds<driver name>.so` for Precision Architecture (PA) and Itanium Architecture (IA) platforms, respectively. For example, the shared library from the ENET driver would be `libdsenet.sl` (on a PA platform) or `libdsenet.so` (on an IA platform). The shared library must be located in the subdirectory, `/usr/lib/lanadmin`.

The `driveradmin` option `-x` is used for displaying the driver-specific features, attributes or settings.

**Usage:**

```
lanadmin -x [keyword1] [keyword2] ... <PPA>
```

When `driveradmin` is invoked with the `-x` option without any argument, `driveradmin` must displays the current speed and duplex value of the network interface. This behavior is standard for the HP `lanadmin` command for all HP network interface drivers. HP recommends that the IHVs `driveradmin` command mimic this behavior.

**Examples:**

```
lanadmin -x help <PPA>
```

Displays the driver's help menu for information retrieval.

The following is the output displayed from the `igelan` driver:

```
$ lanadmin -x help 5
The supported -x options are:
speed Display the speed and duplex value of the link
fctrl Display whether Receive Flow Control is enabled or not
```
LAN Commands

LANADMIN Command

send_max_bufs  Display the value of send buffer coalescing threshold
recv_max_bufs  Display the value of receive buffer coalescing threshold
send_coal_ticks Display the value of send interrupt coalescing ticks
recv_coal_ticks Display the value of receive interrupt coalescing ticks
stats drv     Display driver and adapter statistics
card_info     Display the adapter and driver revision and settings

The supported -X options are:
  auto_on      Set the card in autonegotiation mode
  auto_off     Turn off Auto-Negotiation mode for a 1000Base-SX card
  fctrl on     Turn on receive flow control
  fctrl off    Turn off receive flow control
  10hd         Set the speed of a 1000Base-T card to 10 Mbps half-duplex
  10fd         Set the speed of a 1000Base-T card to 10 Mbps full-duplex
  100hd        Set the speed of a 1000Base-T card to 100 Mbps half-duplex
  100fd        Set the speed of a 1000Base-T card to 100 Mbps full-duplex
send_max_bufs Set send buffer coalescing threshold [1 - 128] default 10
recv_max_bufs Set receive buffer coalescing threshold [1 - 256] default 1
send_coal_ticks Set send interrupt coalescing ticks [0 - 10000000] default 150
recv_coal_ticks Set receive interrupt coalescing ticks [0 - 10000000] default 0
stats clear   Clear all driver and adapter statistics

lanadmin -x speed <PPA>

Displays the speed and duplex values, and Autoneg status.

    $ lanadmin -x speed 5
    Speed = 1000 Full-Duplex.
    Autonegotiation = On

In the previous examples, help and speed keywords are used for the -x option.
The -X option is similar to the -x option, but is used to set the values.

Usage:

    lanadmin -X [keyword1] <value> [keyword2 <value>] ... <PPA>

Example:

    $ lanadmin -X auto_on 5

Writing the Driver Specific Shared Library

When driveradmin is invoked with the -x option, it loads the driver-specific shared library and calls the
driver-specific function. This function must be named with the following convention:

drivername_get_special(int fd, int ppa, char *driver_special_arg, char *ret);

    fd        File descriptor for the opened device file of the driver.
    ppa       Current Physical Point of Attachment (PPA) number.
    driver_special_arg Arguments passed for the -x option. The arguments are separated by
                        "space" characters. The length of driver_special_arg must not exceed
                        255 characters.
    ret       Pointer to a string which lanadmin displays if the function did not return
                        an error.

This deprecated argument is currently not being used by any driver. Do not use this argument, it has been retained for backward compatibility only.
Return value:

- Zero on success, a non-zero value on failure.

The `<driver name>` must be the same as the name of the driver. The `driveradmin` command gets this name from the HP-DLPI's list of PPAs. When `driveradmin` is invoked with the `-X` option, it loads the driver specific shared library and calls the driver-specific function. This function must be named with the following convention:

```
dl<driver name>_set_special(int fd, int ppa, char *driver_special_arg);
```

- `fd` File descriptor for the opened device file of the driver.
- `ppa` Current PPA number.
- `driver_special_arg` Arguments passed for the `-x` option. The arguments are separated by “space” characters. The length of `driver_special_arg` must not exceed 255 characters.

Return value:

- Zero on success, a non-zero value on failure.

The `<driver name>` must be the same as the name of the driver, which must be the same as the driver special file used to open the driver.
LINKLOOP Command

The linkloop command can be used to verify the link level connectivity between two LAN interfaces. The linkloop command that ships with HP-UX 11i v1.0 only works with HP network interface drivers. The IHVs network driver developers are expected to provided their own version of this command. The ENET sample network driver sources also come with fully functional the enetlinkloop command sources. These can be easily modified by IHVs to get it to work with their network interface driver.

The diagram shown in Figure 7-3, “HP and IHV LINKLOOP Commands,” shows how the HP linkloop and IHV driverlinkloop command fit within the HP-UX 11i v1.0 networking stack.

Figure 7-3 HP and IHV LINKLOOP Commands
The `enetlinkloop` expects the underlying driver to implement the following DLPI primitives and uses them to verify link level connectivity:

- **REQ**: DL_TEST_REQ
- **ACK**: DL_TEST_CON
- **REQ**: DL_INFO_REQ
- **ACK**: DL_INFO_ACK

**Usage:**

`enetlinkloop -i <ppa> <physical address of target interface>`

**Example:**

```bash
$ enetlinkloop -i 0 0x0060B0B63657
Link connectivity to LAN station: 0x0060B0B63657
-- OK
```
8   Tracing and Logging in LAN Drivers

This section describes the use of the HP-UX network tracing and logging facilities. To aid in troubleshooting network problems, support for network troubleshooting must address several trends:

- The complexity of network systems is increasing.
- The number of protocols and standards is large and continues to grow.
- The possible combinations of services and applications created and used on a network is increasing.
- The person troubleshooting is usually far removed from those who understand the network, products, and systems best.

The person troubleshooting needs knowledgeable support tools to address this complexity and difficulty. Support tools must provide as much information as possible about when and where problems occur. The network code must provide the failure occurrence, cause, and suggested repair information.

HP-UX network tracing and logging facilities are tools for capturing network events and packets in a log for analysis to support troubleshooting. Sometimes special diagnostic and test tools must also be used; for example, network traffic analyzers, interpretability tests, and other such aids.

HP-UX network tracing and logging facilities permit subsystems to record events in a central location for subsequent processing. The information can then be provided to customers and support personnel to audit network activity and troubleshoot network problems.
Overview

HP-UX network tracing and logging facilities provide the following general features:

- A mechanism for recording log events and trace data
- A facility for determining what information to capture
- A mechanism for selecting and formatting the recorded information
- A set of user interface commands that:
  - Configure, start, and stop the trace and log services.
  - Format captured messages.

The commands and other HP-UX network tracing and logging facilities (files, subroutines, etc.) discussed in the following sections provide a programmatic interface that allows user and kernel routines to access the services:

- `nett1conf` command
  - `nett1gen.conf` subsystem configuration database
  - `nett1` command
  - `netfmt` command
- Storage buffer in shared memory
- Subsystem Management Table in shared memory
- Storage buffer in kernel
- Subsystem Management Table in kernel
- `ntl_reader` daemon
- `nkt1_daemon` daemon

*Figure 8-1, “Network Tracing, Logging Elements and Data Flow I,”* shows the data flow of the elements in the HP-UX trace and log system.
The elements are explained in the following sections of this chapter.

**nettgen.conf(4)**

The `nettgen.conf` file stores subsystem records, particularly the unique subsystem ID. This subsystem information is used by the `nettl` and `netfmt` commands to identify and control subsystem tracing and logging behavior. Each subsystem must have a unique subsystem ID. The ID is used as identification for all interactions with the tracing and logging facility.

**NOTE** Obtain this subsystem ID from Hewlett-Packard, refer to Assign Subsystem ID.
nettlconf(1M)

The nettlconf command creates and updates the database file /etc/nettlgen.conf, the file used to configure each subsystem. This database file controls the behavior of the nettl and netfmt commands for tracing, logging, and formatting (trace/log) messages. Refer to the manpages that were shipped with the system on nettlconf(1M), nettl (1M), and netfmt (1M). and to the HP-UX 11i v1 Driver Development Reference Guide.

Information such as the subsystem name, library name, and subformatter function are given to the nettlconf command, which stores them in the /etc/nettlgen.conf configuration file. This command is used in the configure script of the subsystem module during a system install/update time to integrate the subsystem into the trace and log tool. Subsystems use the nettlconf command to store a description of themselves in the nettlgen.conf database file — typically performed only once, at product installation time.

nettl(1M)

This command uses the subsystem information to create subsystem management tables in shared memory and in the kernel; it starts, stops, and sets the capture criteria for tracing and logging. Specifically, nettl creates a port where messages can be stored while being written to the output file. The nettl command initializes the ktl driver, also called netdiag1, and nettl starts up the nktl_daemon and ntl_reader daemons. See the manpage in the HP-UX 11i v1 Driver Development Reference Guide for more detailed information.

netfmt(1M)

This command formats binary trace and log data into readable ASCII text. Post-filtering of the data is controlled through this command:

1. The netfmt command uses subsystem configuration information to identify shared libraries provided by subsystems that contain functions to parse subsystem filters and format subsystem data.
2. The netfmt command dynamically loads all shared libraries and finds the functions each time it is executed.
3. The command calls the functions of subsystems for which it has data.
4. The command parses the filter file if it is present. The file is sorted according to the first field, the subsystem name, in the filter file.
5. For each subsystem referenced in the file, the subsys_N_get_options() function for that subsystem is called with the filter data. The subsys_N_get_options() function is responsible for interpreting and storing the filter data.
6. The netfmt command reads the input file. For each record found it calls the corresponding subsys_N_format() function to format the record.

The subsystem will not format the record if the values in the record match the values specified in the filters. The subsystem should format the record according to the format options specified; for example, nice, terse, and raw. Refer to the netfmt (1M) manpage in the HP-UX 11i v1 Driver Development Reference Guide that was shipped with the system for detailed information.
Using HP-UX Logging and Tracing for Troubleshoot Support

The following guidelines can help developers to be “user friendly” when designing tracing and logging facilities to solve the troubleshooting problems of their clients:

- Log only what is needed to solve problems.
- Record all information to diagnose the problem in the log.
- Provide a hex dump to the person troubleshooting — only as the last resort.
- Make each product do as much self diagnosis and repair as possible. Notify the end user only when intervention is required or requested.
- Give the customer what is needed to solve their problems.

The following information will help set up tracing and logging to support troubleshooting:

- Assign Subsystem ID
- Classify Trace Data
- Format Trace Data
- Classify Log Data
- What and When to Log

Assign Subsystem ID

Each networking product requires its own unique subsystem ID number, which must be assigned by the Hewlett-Packard NetTL Team.

To do so, Email a request for a unique subsystem ID for the product to Hewlett-Packard at nettl_support@india.hp.com. In the message identify a suggested interface subsystem name for the product. Check /usr/include/sys/subsys_id.h in the system prior to selecting the name. A name will be assigned if it is not already being used. Do not request names such as lan, lo, ni, X25, and others that are already assigned.

The response from HP will include a unique subsystem ID number and a subsystem name in an up-to-date file of unique subsystem ID numbers and associated subsystem names.

This subsystem ID number is represented as the variable N in the rest of this chapter.

Classify Trace Data

Tracing can capture or make snapshots of loopback or header information, as well as inbound and outbound packets going through the network. The main purpose of tracing is to analyze networking problems discovered in either a log error message or the failure of a networking operation to complete successfully. Tracing follows or records normal events and abnormal events alike and is typically used on events that occur frequently, such as connections opening and closing, or re-transmitted data.

Traces are defined as:

- PDU: Inbound and outbound Protocol Data Units (PDU) including header and data.
- Header: Inbound and outbound protocol headers.
Overview

Loopback  Trace packets emanating and returning to the same system.
Procedure  Trace entry and exit from all procedures.
Error  Invalid state transitions, invalid protocol data units, bad headers, resource errors, system call errors, and protocol violations. Distinguishing when to use an error trace or an error log can be difficult. In some cases, use both. The tracing and logging utility goes to different files, and locating and synchronizing the entries between the two files can be difficult. Having both an error log and error trace helps to synchronize the two files. Sometimes other log messages are also recorded in the trace file when tracing is enabled.
State  Protocol states or connection states, not limited to entry and exit from a layer or procedure. Use this trace kind when recording information about normal state transitions.
Connection  Information about connections as they are made and destroyed.
Logging Trace  A special kind of trace that contains a log message. This trace will help in troubleshooting by locating and synchronizing logging and tracing output.

Format Trace Data

Troubleshooting should trace both incoming and outgoing data through the stack. The trace records from different processes should be threaded together to form a complete record of the path the PDU takes going from the user application out the wire, and vice versa.

Refer to the following guidelines when implementing the tracing routines:

- Each process should trace incoming and outgoing data from both top and bottom. Alternatively, each protocol could trace only its incoming and outgoing headers.
- A subformatter for a process’s trace information must be provided by the implementer of the process.
- The subformatter formats only the data for which that process is responsible. For example, if the X.25 driver sends a trace record, it decodes only the X.25 portion of the PDU, leaving the rest for the process above it to decode. Likewise, OTS decodes only the Network, Transport, and Session layer portions, leaving the upper layers to the application processes.

Classify Log Data

Logging is a way of capturing and recording specific network activities and infrequent significant network events, such as state changes, errors, and connection establishment. The main purpose of logging is to inform the system operator about these significant events and to make a permanent record for later interrogation. Typical log messages are about errors (catastrophic, recoverable and non-recoverable), warnings (major and minor), or system wide information (such as changes to configuration or operation).

Logged events are:

Disaster  Signals that the software detected a severe and irrecoverable error condition that typically affects multiple user applications or connections and can jeopardize system integrity. For example, the condition can cause a system crash or corrupt a system table. Another example is when a condition implies that an action generated by one process can damage other processes.

Error  Signals an event or condition that, while not affecting the overall subsystem or network operation, causes an application program to fail or complete in an error condition. Indicates that the system is not performing as it should, but the underlying networking subsystem was able to recover. For example, an error class condition occurs when a process must abort its operation or take extra steps to recover a certain state.
Warning Indicates an abnormal event, but not necessarily a networking problem event, possibly caused by a subsystem problem. Examples include possible pointer alignment problems or data being accessed that has not been initialized.

Informative Describes infrequent operations and current system activities, such as protocol module initiation and termination sufficiently important to post.

What and When to Log

The most important part of logged messages is the ASCII string describing the event, which is the first item a system operator might see on the system console following an event in the network operation. Deciding what to log and when it should be logged often involves trade-offs in terms of usability, performance, schedule constraints, and management and peer pressure. Other than the items outlined in the preceding tracing or logging sections, some general guidelines include:

- If an event results or causes the product or system to be unusable by all users, it should be logged as a Disaster class log message.
- If an event affects a single application, it should be logged as an Error class log message.
- If an event can cause an error or disaster in the future or cause performance degradations, it should be logged as a Warning class log message.
- If an event occurs infrequently and is something the user may want to know about, but will not cause future problems, it should be logged as an Informative class log message.
- If an event occurs frequently or with regularity, it is probably not appropriate to log it, but to trace it instead. Don't use Informative log messages in place of tracing.
- Do not log “Me Too!” messages in Error or Disaster class. These are events which occur in response to an error or disaster event in another place, but aren't themselves a disaster or error. “Me Too!” messages are characterized as providing no additional information to solve the problem at hand.
- Do not acquire a new log instance if one is already available for the particular event thread (a log instance is a unique static number used to identify the thread of events attending an interface).
- Include as much information as possible in log messages. The person troubleshooting should be able to know what happened, what caused it, and how to proceed to fix the problem, on the basis of the log message alone.
- State the exact commands to use to perform the recommended actions.
- If the explanation is too long to include in the log message, refer to the appropriate manual to take further steps or gain more knowledge about the problem.
- Encapsulate logging calls in functions or macros.
- Adhere to the logging error classes (Disaster, Error, Warning, and Informative) to promote uniformity in the troubleshooting process recommended and to facilitate communication with HP support groups.
- Restrict logged information to only a few well defined types, for example; event number, a bounded array, or a string.
- Identify error recovery procedures for Disaster and Error class events.
- Devote most of the effort to understanding and documenting the procedures listed above. Only after completing error recovery procedures for these events should the focus be on Informative and Warning class events, and then only if they would actually be useful.
Passing Data to HP-UX Tracing/Logging

Kernel subsystems that use the trace and log services must include the following in their source files and makefiles:

```c
#include <net_diag.h>
```
Contains macro calls to check that tracing and logging is enabled for the subsystem.

```c
#include <subsys_id.h>
```
Contains subsystem information and definitions for log classes and trace kinds.

The function calls for kernel subsystems capture trace and log data.

**KTRC_CK()**

This macro is used to trace on an all interface device basis. It allows the calling process to verify if tracing is enabled for the current subsystem. The returned value is one (1) if tracing is enabled. It is defined as:

```c
KTRC_CK(subsys_id, trace_kind)
```

- **subsys_id**: Unique subsystem ID of the calling subsystem. The number is assigned by Hewlett-Packard, refer to the “Assign Subsystem ID” section.
- **trace_kind**: Defines trace kind, these are defined in `subsys_id.h` header file and are detailed in “Classify Trace Data” section, or as follows:
  - HDR_IN_BIT: Inbound header tracing mask
  - HDR_OUT_BIT: Outbound header tracing mask
  - PDU_IN_BIT: Inbound PDU tracing mask
  - PDU_OUT_BIT: Outbound PDU tracing mask
  - PROCEDURE_TRACE_BIT: Procedure entry/exit trace
  - ERROR_TRACE_BIT: Error tracing mask
  - LOGGING_TRACE_BIT: Log call tracing mask
  - LOOP_BACK_BIT: For loopback
  - PTOP_BIT: For point to point

**NOTE**
There are some alias or redefine the `trace_kind` functions in the `net_diag.h` header file mentioned earlier.

For example:

```c
#define TR_LINK_LOOP LOOP_BACK_BIT
#define TR_LINK_INBOUND PDU_IN_BIT
#define TR_LINK_OUTBOUND PDU_OUT_BIT
```

A hypothetical driver named `enet.c` might use this macro as follows:

```c
if (KTRC_CK(ENET_ID, TR_LINK_INBOUND))
{
    ktrc_write(...);
}
```
**ktrc_write()**

This routine is used to send trace messages to the kernel trace and log facility.

Prefiltering is done at the time of the trace call, and unwanted messages are dropped. This routine always returns a success indicator of zero and is defined as:

```c
ktrc_write ( int subsys_id, int trace_kind, int path_id,
            int device_id, caddr_t tl_packet,
            int tl_packet_cnt)
```

- **subsys_id**  
  Unique subsystem ID of the calling subsystem (number assigned by Hewlett-Packard) refer to “Assign Subsystem ID” section.

- **trace_kind**  
  Defines the kind of trace. All kinds are defined in the header file subsys_id.h. The following is the defined trace kind values, also refer to the “Classify Trace Data” section. They can be OR’ed to produce the combination of trace kinds:
  - HDR_IN_BIT  
    Inbound header tracing mask
  - HDR_OUT_BIT  
    Outbound header tracing mask
  - PDU_IN_BIT  
    Inbound PDU tracing mask
  - PDU_OUT_BIT  
    Outbound PDU tracing mask
  - PROCEDURE_TRACE_BIT  
    Procedure entry/exit trace
  - STATE_TRACE_BIT  
    State machine tracing mask
  - ERROR_TRACE_BIT  
    Error tracing mask
  - LOGGING_TRACE_BIT  
    Log call tracing mask
  - LOOP_BACK_BIT  
    For loopback
  - PTOP_BIT  
    For point to point

- **path_id**  
  Connection path on the host. If this is a nonapplicable parameter, pass in −1.

- **device_id**  
  Device ID number (for example, if_unit) of the calling subsystem message. If this is a nonapplicable parameter, pass in −1.

- **tl_packet**  
  Either a pointer to an mbuf chain or a pointer to a set of iovec structures as determined by tl_packet_cnt. The calling routine will pass a pointer (cast to caddr_t) to an mbuf chain or an iovec structure. This structure is immediately copied into an mbuf chain owned by tracing and logging facilities. Therefore, it is not necessary for the calling routine to copy the data and then pass a pointer to it.

- **tl_packet_cnt**  
  If −1, tl_packet points to an mbuf chain. If greater than zero, this is the number of the iovec structure to which tl_packet points.

As with logging, developers should encapsulate tracing calls in functions or macros. The code scenario in the the following section shows a typical use of tracing calls.
Overview

Tracing Code Sample

The following example shows a trace of an outbound packet whose various parts are located in distinct memory locations. The trace uses the vectored data capability of the ktrc_write() call. The same could be accomplished using an mbuf chain as well.

```
1 #include "../h/netdiag1.h"
2 #include "../h/net_diag.h"
3 #include "../h/subsys_id.h"
4 #include "../h/uio.h"
5 ...
6
7 #define MAX_BUF 3 /* any number of vectors are allowed */
8 #define TRACE 0
9 #define FALSE 0
10 ...
11 ...
12 ...
13 int
14 trace_pdu_out(pdu_hdr, pdu_hdr_len, pdu_data, pdu_data_len)
15     char *pdu_hdr;
16     int pdu_hdr_len;
17     char *pdu_data;
18     int pdu_data_len;
19 {
20     int kind;
21     int device_id;
22     int path_id;
23     short subsys_id;
24     struct iovec tl_buf[MAX_BUF];
25     int tl_buf_cnt;
26         /*
27         * Set up variables for KTRC_CHECK()
28         */
29         /*
30         subsys_id = MY_SUBSYS_ID_NUMBER; /* assigned by HP */
31         kind = PDU_OUT_BIT;
32         device_id = -1; /* -1 means not applicable */
33         path_id = -1; /* -1 means not applicable */
34 
35         if (KTRC_CHECK(subsys_id, kind, device_id))
36         {
37             /*
38             * Tracing is enabled for this subsystem
39             * and kind combination.
40             */
41             tl_buf[0].bufptr = pdu_hdr;
42             tl_buf[0].buflen = pdu_hdr_len
43             tl_buf[1].bufptr = pdu_data;
44             tl_buf[1].buflen = pdu_data_len;
45             tl_buf[2].bufptr = NULL;
46             tl_buf[2].buflen = 0;
47             tl_buf_cnt = 2;
48             ktrc_write(subsys_id,
49                     kind,
50                     path_id,
51                     device_id,
52                     &tl_buf,
53                     tl_buf_cnt);
54         }
55         return(0);
56     }
```
KLOG_CK()

This macro allows the calling process to find out whether logging is enabled for the current subsystem. The
returned value is one (1) if logging is enabled. It is defined as:

KLOG_CK (subsys_id, log_class)

subsys_id  Unique ID number (assigned by Hewlett-Packard) of the calling subsystem.
log_class  Defines the classification of event. All classes are defined in the header file subsys_id.h see
the “Classify Trace Data” section. Four classes are defined for logging messages:

Informative  Normal messages only
Warning  Warning messages
Error  Error condition messages
Disaster  Critical error messages

kget_log_instance()

This call accepts no parameters but returns a unique log instance value. The log instance helps thread log
messages together so the user can easily identify the messages that result from the same event. A change in
the log instance means a new event is being logged.

The log instance value is passed between subsystems through their interface parameter list so each module
can have access to it. If a module encounters a unique event, it will obtain a log instance value. Otherwise, the
module would use the current log instance value it was passed without calling kget_log_instance(). (See
also, klogg_write() for information on log instance values.)

klogg_write()

This routine is used to send log messages to the kernel trace and log facility.

Prefiltering is done at the time of the log call, and unwanted messages are dropped. This routine always
returns a success of zero and is defined as:

klogg_write ( int subsys_id, int class, int device_id,
            int log_instance, caddr_t tl_packet,
            int tl_packet_cnt)

subsys_id  Unique ID (number assigned by Hewlett-Packard) of the calling subsystem.
class  Defines the classification of event. All classes are defined in the header file subsys_id.h
(see the “Classify Trace Data” section). Four classes are defined for logging messages:

Informative  Normal messages only
Warning  Warning messages
Error  Error condition messages
Disaster  Critical error messages

device_id  Device ID number (for example, if_unit) of the calling subsystem message. If this is a
nonapplicable parameter, pass in −1.
log_instance  Unique static number used to identify the thread of events attending an interface. If this is
a nonapplicable parameter, pass in −1.
Either a pointer to an mbuf chain or a pointer to a set of iovec structures as determined by

By tl_packet_cnt. The calling routine passes a pointer (cast to caddr_t) to an mbuf chain or

By an iovec structure. This structure is immediately copied into an mbuf chain owned by

By tracing and logging facilities. So the calling routine need not copy the data and then pass a

By pointer to the data.

If −1, tl_packet points to an mbuf chain. If the value is greater than zero, it is the number

By of the iovec structure (as defined in uio.h) the tl_packet points to.

Logging Code Sample

The following scenarios describe the intrinsic calls of HP-UX logging facilities. They are typical fragments of

code that a subsystem might include to perform logging calls:

```c
#include "netdiag1.h"
#include "net_diag.h"
#include "subsys_id.h"

#define MAX_BUF 1 /* any number of vectors allowed */
#define LOG 1
#define FALSE 0

extern int log_instance;
extern unsigned short kget_log_instance;

int log_disaster()
{
    int class;
    int device_id;
    event_data_type event_data;
    short subsys_id;
    struct iovec tl_buf[MAX_BUF+1];
    int tl_buf_cnt;

    /* Set up variables for call to KLOG_CK() */

    subsys_id = MY_SUBSYS_ID_NUMBER; /* assigned by HP */
    class = DISASTER;
    device_id = -1; /* -1 means not applicable */

    if (KLOG_CK(subsys_id, class))
    {
        /* Logging enabled for this subsystem and class combination. */

        if (log_instance == 0)
        {
            /* There was no previous log instance associated with this event. This is
            * the first module to encounter the problem, so it gets the log instance.
            * Log instance should be available to all modules in the subsystem and to
            * other subsystems.
            */
```
Chapter 8

Tracing and Logging in LAN Drivers

Overview

55 */
56 log_instance = kget_log_instance();
57 }
58
59
60 event_data.event_number = THIS_EVENT_NUMBER;
61 event_data.event_type = THIS_EVENT_TYPE;
62 /*
63 * Additional data about the event can be
64 * placed in the data structure. This
65 * data structure is entirely up to the
66 * local developer to design. The
67 * subformatter for this subsystem must
68 * be able to decode the data structure,
69 * but other than that there are no
70 * restrictions on what gets passed. The
71 * local developer may choose to use a
72 * single mbuf chain to hold all the
73 * event information, or pass a vectored
74 * buffer to the klogg_write() call to
75 * hold individual pieces of information.
76 *
77 *
78 * Callers should NOT pass strings in this
79 * function; the event number as shown in
80 * this example should be used to
81 * generate an NLS string from a message
82 * catalog in the subformatter.
83 */
84
85 tl_buf[0].bufptr = *event_data;
86 tl_buf[0].buflen = sizeof(event_data_type);
87 tl_buf[1].bufptr = NULL;
88 tl_buf[1].buflen = 0;
89 tl_buf_cnt = 1;
90
91 klogg_write(subsys_id,
92 class,
93 device_id,
94 log_instance,
95 &tl_buf,
96 tl_buf_cnt);
97 }
98 return(0);
99 }
Format Networking Trace/Log Messages

The following sections detail facilities and network device driver developer's responsibilities for formatting trace/log output. The netfmt formatter is the HP-UX facility that presents trace and log information in human readable form. It comprises two distinct components:

**Subformatter**
Function(s) provided by the tracing/logging driver subsystem to interpret the subsystem messages and produce human readable form output.

**Formatter Core**
Responsible for file handling, global filtering and dispatching messages to the appropriate subsystem formatter. In addition to these, netfmt also provides many generic formatting utility functions (as listed in the “Formatting Routines” section), to aid the subsystem developer in developing a subformatter and also to achieve consistency in output by various subsystems.

Refer to Figure 8-1, Network Tracing, Logging Elements and Data Flow I, for the interaction among the various components.

Registration of Subformatter with NetTL

The subsystem that provides the subformatter registers the relevant information with NetTL in the configuration database i.e., nettlgen.conf file (see nettlgen.conf(4)) using the nettlconf command (see nettlconf(1M)). For details on the required information, refer to “Configuring Subsystems into the System” in this chapter.

For example the subformatter, for subsystem N, mainly comprises the following functions:

**Subsys_N_format()**
This function provided by the subsystem interprets the trace/log message and presents it in a human readable form. Refer to the *HP-UX 11i v1 Driver Development Reference Guide* for subsys_N_format (NET_DRV) for more details.

**Subsys_N_get_options()**
A subsystem can provide this function if it needs to perform any subsystem specific filtering of its trace/log message. Please note that formatter core provides some common filtering capabilities (see netfmt(1M)) which are applicable to more than one subsystem. Refer to subsys_N_get_options (NET_DRV) reference page for more details.

The previously listed functions have to be provided in a shared library, typically named libNfmt.so. The netfmt formatter uses the subsystem configuration information in the nettlgen.conf database file to identify the shared library which contains the functions listed previously. The netfmt dynamically loads all shared libraries and finds the functions each time it is executed; it calls the functions of a subsystem only when it has data belonging to the subsystem.
Formatter Responsibilities/Features

The formatter core's \( \text{netfmt} \) responsibilities and capabilities are listed:

- Loads subformatter libraries using the subsystem configuration information in \( \text{nettlgen.conf} \) database file.
- Processes filtering and formatting options. The formatter determines filtering and formatting options by processing an auxiliary file.
- Handles global filtering. — The formatter checks each message against the filters that are set up. Only a message that comes past the filter will be sent to the respective subformatter. During filtering, the formatter checks the message to make sure it contains good information. If the formatter finds a corrupted message header, an unknown subsystem, a message that is too long to handle, and so forth. It prints an informative message, formats the message header and discards the remainder of the data and continues with the rest of the file.
- Supports subsystem specific filtering.
- Dispatches data to the correct subformatter.
- Handles common subformatter tasks. — The formatter provides the utility functions that subformatters can call to perform common tasks, such as formatting the message header in a standard fashion, dumping raw data and outputting the formatted data. The formatter also provides routines to format the standard network protocols as described in the Trace “Formatting Routines” section in this chapter.
- Provides various modes of formatting. — The formatter provides three modes of formatting for trace data as explained:
  1. Terse (one-line) — This mode of formatting is the least descriptive and is used to get a summary of trace file contents in a single line. Additional options/flags control the behavior of terse formatting.
  2. Nice (detailed) — This is the most detailed level of formatting available and attempts to decode every piece of data in the trace.
  3. Raw (hex-dump) — This is the default mode of formatting and just prints the trace data in hex format.

The formatter conveys the formatting mode that is active by enabling the appropriate flags when calling subformatter functions.

Formatting Routines

The formatter provides two types of formatting routines to be used by subformatter developers:

1. Utility routines — These are the general utility functions that help the subformatter in obtaining required information and displaying output of the trace/log message in a consistent manner.

2. Trace formatting routines — With the help of these functions, a driver subformatter can completely format trace data by handling just the link header information, not including 802.2 information.

By using the common functions, the underlying implementation of I/O can be changed easily, thus allowing easier porting, further performance enhancements, alternative output schemes, and so forth.

Utility Routines

The utility routines provided by the formatter are listed in Table 8-1, “Utility Routines.” Refer to the routine’s reference page for exact syntax and further details. The following functions are present in the libfmtutil.so library. Subsystems should not link to this library. All externals are resolved during dynamic loading at run time.
Table 8-1 Utility Routines

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tl_banner_char()</td>
<td>Returns the character to be used in the header banner, which is printed using the tl_header_format() function.</td>
</tr>
<tr>
<td>tl_check_cat_version()</td>
<td>Checks that the registered subsystem message catalog is compatible with the subsystem formatter library version.</td>
</tr>
<tr>
<td>tl_format_fprintf()</td>
<td>Converts, formats and prints its arguments. This function behaves like printf.</td>
</tr>
<tr>
<td>tl_format_write()</td>
<td>Writes the decoded buffer to stdout.</td>
</tr>
<tr>
<td>tl_get_line()</td>
<td>Used to obtain a line from the filter file. This function is called by the subsystem’s subsys_N_get_options() function.</td>
</tr>
<tr>
<td>tl_get_parms()</td>
<td>Returns to the caller (typically, the subformatter) a ss_N_fmt_parms_type() data structure (see ss_N_fmt_parms_type (NET3), which contains all the information that a subformatter needs in order to operate. This function can be called anywhere within the subformatter.</td>
</tr>
<tr>
<td>tl_header_format1()</td>
<td>Formats a single trace/log header. The format of the output will conform to the standard HP-UX network tracing and logging recommendations. This function must be called by every subformatter after subsystem filters, if any, have been processed.</td>
</tr>
<tr>
<td>t_log_class()</td>
<td>Returns a text interpretation of a log class. The result of this function is typically used by the subformatter to pass as a parameter to the tl_header_format1() function when printing the header.</td>
</tr>
<tr>
<td>tl_raw_format()</td>
<td>Formats a trace/log message into both hexadecimal and printable ASCII characters. This function is typically used by the subformatter when it can not further decode the information in the trace/log message.</td>
</tr>
<tr>
<td>tl_trace_kind()</td>
<td>Returns a text interpretation of a trace kind. The result of this function is typically used by the subformatter to pass as a parameter to the tl_header_format1() function when printing the header.</td>
</tr>
</tbody>
</table>

Trace Formatting Routines

The formatter core provides the ability to format information from upper layer protocols such as IP, TCP, UDP, ARP, DUX, NFS, from traces taken at link layer. This capability makes it much easier to analyze networking dialogs than examining raw hex data and manually determining what the protocols were sending.

In addition, the formatter also provides the ability to filter the trace output, for example, to display only dialogs taking place with a particular TCP port. The filters include Ethernet type, 802.2 SAPS, IP addresses, UDP ports and RFC information. Refer to the manpages that were sent with the system, for netfmt (1M) for the complete list of filters supported.
This section describes a set of routines available in the netfmt formatter that new link products can and should take advantage of. Link subformatters can use the following decoding routines to format link level packets:

set_up_xxx(): These routines set up global protocol information for the traced packet, to be used later by the filtering and formatting routines.

NOTE   The xxx can be one of “8022”, “link”, “ether” or “ip”.

filter_packet(): This routine determines if the traced packet meets any user-specified filter criteria.

format_link_xxx(): If a packet passes the filters, the packet can be displayed by calling the appropriate format_link_xxx() formatting routine. The exact function called is determined by the mode of formatting that is in effect.

The

NOTE   xxx can be one of “nice”, “raw” or “terse”.

By using these routines, a link product trace subformatter needs to format only the information in its link header, not including 802.2 information. Note also, that the trace subformatter does not directly perform I/O, which is performed through the three formatting routines provided. Using these routines allow future changes to be made to the look of the formatted output without modifying the link subformatter code.

Each of the trace formatting routines provided by the formatter core are listed in Table 8-2, “Trace Formatting Routines.” Refer to the routine’s reference page for exact syntax and further details. The following functions are present in the libnsfmt.so library. Subsystems should not link to this library. All externals are resolved during dynamic loading at run time.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>set_up_8022()</td>
<td>Initializes the global variables (for various protocols) to be used by the filter and formatting routines. This routine handles 802.2 and upper layer protocols and can be called for each PDU_IN/PDU_OUT Ethernet packet.</td>
</tr>
<tr>
<td>set_up_ether()</td>
<td>Initializes the global variables (for various protocols) to be used by the filter and formatting routines. This routine handles upper layer protocols information and can be called for each PDU_IN/PDU_OUT Ethernet packet.</td>
</tr>
<tr>
<td>set_up_link()</td>
<td>Sets up global information only for the data link layer and does not attempt to extract any upper layer information from the traced packet.</td>
</tr>
<tr>
<td>set_up_ip()</td>
<td>Sets up global information for IP and upper layers. Most link products should not use this routine and needs to be used only when no link information is available (for example, when formatting NS_LOOPBACK trace packet.</td>
</tr>
<tr>
<td>filter_packet()</td>
<td>Examines the globals set up by one of the preceding set_up_xxx() routines and returns 0 if the packet should not be displayed.</td>
</tr>
<tr>
<td>format_link_nice()</td>
<td>Formats a packet to display upper layer information in detail. The subformatter should call this routine when nice format is enabled.</td>
</tr>
</tbody>
</table>
Examples

The following code fragment shows how the enet subformatter handles trace data:

```
subsys_enet_format()
{
    tl_msg_hdr_type *hdr_ptr;    /* Pointer to the TL header */
    char * data_ptr;             /* Pointer to start of data (after TL header */
    struct ieee8023_hdr *trace_buffer;
    char *kind_str;                /* Pointer to string containing kind */
    int tl_flag;                /* Is this a trace or log message? */
    int data_len;
    u_short ether_type;

    /* Trace formatting code
     * Extract TL message header and data
     */
    hdr_ptr = (tl_msg_hdr_type *) binary_msg_ptr;
    data_ptr = binary_msg_ptr + sizeof(tl_msg_hdr_type);

    /* Amount of data after the TL header
     */
    data_len = hdr_ptr->data_len;

    /* Set tl_flag; 1 for log and 0 for trace
     */
    tl_flag = hdr_ptr->flags.trace_log_bit;
    if (tl_flag == TRACE) {
        /* Format trace data if any
         */
        if (!data_len)
            return(0);
```

Table 8-2 Trace Formatting Routines (Continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>format_link_terse()</td>
<td>Formats a packet to display upper layer information in a single line. The subformatter should call this routine when terse format is enabled.</td>
</tr>
<tr>
<td>format_link_raw()</td>
<td>Formats a packet to display upper layer information as hex/ASCII data. The subformatter should call this routine when raw format is enabled.</td>
</tr>
</tbody>
</table>
/* Now extract MAC information from trace data */
trace_buffer = (struct ieee8023_hdr *)data_ptr;

/* Make data_ptr point to beginning of 802.2 header. */
/* data_len is length of buffer from where data_ptr points to. */
if (trace_buffer->length < MIN_ETHER_TYPE) { /* IEEE packet */
  ether_type = 0;

  /* both IEEE and ETHER header are 14 bytes long */
  data_ptr = data_ptr + 14;

  /* point to LLC */
  data_len = data_len - 14;

  /* Call setup routine to set up structures reflecting the 802.2 */
  /* and above level headers. Handles SNAP as well. */
  set_up_8022(data_ptr, data_len, trace_buffer->destaddr,
              trace_buffer->destaddr, trace_buffer->sourceaddr);
} else {
  ether_type = 1;

  /* both IEEE and ETHER header are 14 bytes long */
  data_ptr = data_ptr + 14;

  /* point to ether data */
  data_len = data_len - 14;

  /* Call setup routine to set up structures reflecting Ethernet */
  /* information and above level headers. */
  set_up_ether(data_ptr, data_len, trace_buffer->destaddr,
               trace_buffer->sourceaddr, trace_buffer->length);
}

/* filter_packet() will indicate whether the current packet meets the
* user specified filter criteria. This routine uses the global info
* setup by set_up_8022/set_up_ether routines (ie. IP address, Ether
* type, TCP port etc.)
*/
if(!filter_packet()) {
    /* Display no information if filter fails
    */
    return 0;
}
/* Call terse formatter, if terse flag is set
*/
if (flags.terse_mode_bit) {
    if (ether_type)
        /* Determine the link type to pass the string iEi for
         * Ethernet and i8i for 802.3 packet
        */
        format_link_terse(hdr_ptr, data_ptr, data_len, "E", "");
    else
        format_link_terse(hdr_ptr, data_ptr, data_len, "8", "");

    /* Always return after terse formatting, the caller only wants
     * one line of information, so never fall through to other
     * format_link_*() routines.
     */
    return 0;
}
/* Call nice formatter, if the nice flag is set.
* Depending on the link type, pass i802.3i or iEtherneti strings to
* the format_link_nice()/format_link_raw() routines. If there is more
* information to pass about the link header, pass it as the
* iaddl Infoi parameter to format_link_nice()/format_link_raw()
* routines. In this case, just ii is passed
*/
if (flags.nice_mode_bit) {
    if (ether_type) {
        /* If format_link_nice() call fails, call format_link_raw().
         * Return
         */
        if (!format_link_nice(hdr_ptr, data_ptr, data_len,
            "Ethernet", ",", ",", ")
            return 0;
        }
format_link_raw(hdr_ptr, data_ptr, data_len, 0,"Ethernet", "", "", "");
}
else {
    if (!format_link_nice(hdr_ptr, data_ptr, data_len, "802.3", "", "", ""))
        format_link_raw(hdr_ptr, data_ptr, data_len, 0,"802.3", "", "", "");
    return 0;
}
/* Call raw formatter */
if (ether_type)
    format_link_raw(hdr_ptr, data_ptr, data_len, 0, "Ethernet", "", "", "");
else
    format_link_raw(hdr_ptr, data_ptr, data_len, 0, "802.3", "", "", "");
return 0
} /* end if TRACE */
} /* End subsys_enet_format */

The following are some output examples for each mode of format:

**Nice Format Output**

The line1 parameter goes here.-vvvvv

====================== 802.3 ============================

Source : 00-00-00-00-00-31 [I] [Cisco ] LENGTH: 26
Dest : 09-00-00-00-00-01 [M] [HP Probe ] TRACED LENGTH: 60
< The addlinfo parameter info goes here
Date : Mon Dec 02 09:22:04:33390 PST 1991
< The upperinfo parameter info goes here

====================== 802.2 ============================

DSAP : 0xfc SSAP : 0xfc CONTROL : 0x03[U-FORMAT]
DXSAP: 0x503 SXSAP: 0x503

============= PROBE VNA REQ (inbound [ICS]) ======
version: 0 length: 16 seq: 0x6dc1
domain: 1 version: 0 rep len: 8 domrep len: 6
Source: 00-00-0c-00-06-31 Requesting: 15.13.106.63

**Terse Format Output**

m probe vna request for: 15.13.106.63 from: 00-00-0c-00-06-31 seq: 6dc1

| 1. Any addlinfo string would appear beginning here.
+ The linktype is placed here (the second character is placed by
  the function and describes what type of MAC address is used,
  (m)ulticast, (b)roadcast, (l)oopback, (i)ndividual.
Tracing and Logging in LAN Drivers

Formatting Networking Trace/Log Messages

Raw Format Output

vvvvv[The linktype parameter goes here]
Received 60 bytes via 802.3  Mon Dec 02 09:22:04:33390 PST 1991
vvvv-[The interface parameter goes here]
   pid=[ICS] interface=[lan0]
   [The line3 parameter goes here]-vvvvvvvvvvvvv
Dest: 09-00-09-00-00-01 Source: 00-00-0c-00-06-31 Length: 00-1a
< [The addinfo parameter info goes here]
   14: fc fc 03 00 00 00 05 03 05 03 00 11 00 10 6d c1........m.
   30: 00 08 00 06 00 01 0f 6a 3f d8 68 fd f1 0c 20..j?.h...
   46: e3 ff 07 50 18 80 00 00 00 00 00 0c 02...P........

Designing a Subformatter

Subsystems are typically an individual program or set of programs that act in concert. Each subsystem requires an associated subformatter, however several subsystems can use the same subformatter. Subformatter design depends on how logging and tracing are used in the subsystem. Subsystems also have the capability to provide filtering or formatting options.

This section deals with the design of the actual function that is called in response to the formatter reading in a record containing data for a specific subsystem.

The subformatter must be able to handle both trace and log data. These can be separated into separate functions once the subformatter has been invoked, but there is no provision for the formatter to call more than one function for a given specific subsystem ID. Formatting requirements for tracing are often different from logging. A subformatter developer should view the action of tracing or logging as a communication from the subsystem to the user, a user who sees only a message from the subsystem and not the medium that carried the message. The subformatter developer should consider the design of the subformatter in relation to the types of information, that come from the subsystem for logging, providing a few pieces of information, such as logging event and a couple of data items may be adequate. The subformatter can assemble the formatted output from a message in its message catalog based on the event ID, and the additional data can be inserted into the message. This method is employed by the ARPA logging subsystems.

Tracing information can be more of a problem, partly because it usually contains much more data, especially in the case of link-level packet tracing (PDU in or out tracing). The subformatter may have to know how the packet was constructed, which layer sits on top, and so on.

For tracing and logging, the subsystem should pass only as little data as possible to output complete, useful information. The formatter passes the flags to the subformatter which can be used to control the format and degree of detail of the output.

Subformatter Responsibilities

The subformatter has few responsibilities, other than transforming the data, as follows:

1. Perform subsystem filtering or options processing (if this feature is provided).

   The formatter options file contains additional information to control the operation of the subformatter. Each line represents the setting of an option. The lines consist of the identifier, which is the same as the subsystem mnemonic, and the arguments recognized by that subformatter. When the formatter recognizes a subsystem mnemonic, it passes that line to the subsystem options function. The subsystem options function is responsible for parsing and determining the contents of the line. By the time the options function receives the line, the mnemonic has been stripped off and all strings have been converted to lower case. The only restriction on the contents of the line is that it cannot exceed 2048 bytes and must contain only printable characters. The *tl_get_line()* function must always be used to read options lines from the file.
Subsystems can adopt this technique to alter the level of information (beyond terse and verbose), to include extra kinds of data, to provide extra filtering (for events or trace/log data that are not covered by formatter's global filtering functions) and so on.

2. Print the header.

3. If console logging is on, format a terse message; or otherwise, format a message in accordance with the format flags.

4. Write the formatted message.

These can be performed with the help of the utility functions mentioned in Table 8-1, “Utility Routines.”

Subformatter Requirements

1. As mentioned in “Registration of Subformatter with NetTL”, the routines subsys_N_format() and optionally subsys_N_get_options() need to be supplied by the subsystem for the formatter to call when subsystem specific actions need to take place. Subsystem specific actions include parsing filter files and formatting the subsystem’s trace or log data.

2. To make use of the generic formatter utility routines, the subformatter should include the following header files:

   - fmt.h: Contains the necessary data structure for the format support calls.
   - ntl.h: Contains the necessary data structure for the trace and log data.
   - subsys_id.h: Contains subsystem identification information and definitions for log classes and trace kinds.

Build and Installation of Subformatter

The shared library, typically named libNfmt.so, and contains the subformatter routines that are usually created by compiling all modules with the "+z" option to cc and linking them together using the options to ld:

```
-b +e subsys_N_format +e subsys_N_get_options.
```

The "+e" option to ld must be used to prevent symbol collisions among the different subformatter libraries.

The default path for the subformatter shared library is /usr/lib/hpux32 directory. If the subformatter library is placed under a different directory, then the absolute path has to be specified in the nettlgen.conf configuration database.

Internationalization and Message Catalog Support

The formatter provides the subformatters the capability to use the National or Native Language Support (NLS) facilities in HP-UX. When registering the NLS subsystem with the tracing and logging system at installation, the name of the message catalog to be used by the subformatter must also be provided.

The message catalog is called as follows:

1. The netfmt opens and closes the message catalog by using the catopen() and catclose() calls respectively.

2. The file descriptor returned by the catopen() is passed to the subformatter.

3. If no message catalog is registered, or if the message catalog cannot be opened, a special file descriptor of −1 meaning “no file” is passed in.

The subformatter performs the appropriate catgets() calls to retrieve their messages from the message catalog. Subsystems should not open their own message catalogs or use multiple message catalogs.
The commonly accepted method of using message catalogs is to use the `catgets()` call, providing the English language string as the default to the call if the message catalog read fails. This is the same string the call would retrieve from the default message catalog, located typically in `/usr/lib/nls/msg/C/name.cat`, where `name` is the name of the subsystem message catalog registered in the `nettlgen.conf` database.

One recommendation for using message catalogs effectively is to have each logging event correspond to a message number, which makes processing and retrieval simpler. Different message sets or an offset can be used for terse (console) and verbose messages. For further information, refer to `enet` driver's subformatter code.

Because the message catalogs can be altered for a given location, the subformatter should also put some kind of identifying tag (such as “FTAM 489”) on the message that is not localized. Support personnel in a different location will then be able to understand what is being logged without trying to translate the text of the message.

Due to the subformatter's dependency on message catalogs to provide the correct text for a log event, the version of the catalog is highly dependent on the version of the subsystem. The `tl_check_cat_version()` function, refer to the `HP-UX 11i v1 Driver Development Reference Guide (tl_check_cat_version (NET3))` to check message catalog versions.

Configuring Subsystems into the System

The process for getting the tracing and logging facility to know about developed subsystems is somewhat complex. Subsystems must inform the tracing and logging facility of their existence at install/update time.

Each fileset is required to have an Software Depot (SD) configure script. Tracing and logging take advantage of this independence to facilitate the configuration of subsystems into the `nettl` and `netfmt` commands.

The `nettlconf` script has the capability to configure the subsystem (see the `nettlconf (1M)` manpage). `nettlconf` should be called from within the configure script during an SD update or installation. The `nettlconf` configures the subsystem information and puts it into the `/etc/nettlgen.conf` database file.

The `nettl` and `netfmt` commands use the information in the `/etc/nettlgen.conf` database file to configure themselves at run time. For the `netfmt` command, the subsystem's subformatters (in shared libraries) are dynamically loaded so all symbols can be resolved. The `nettl` command uses the subsystem names and initial log classes to build the tables necessary to control subsystem operations.

The information that the subsystems need to configure include:

- **Subsystem ID:** Assigned to the subsystem by Hewlett-Packard, refer to the “Assign Subsystem ID” section.
- **Subsystem Mnemonic:** This is the name by which the subsystem will be identified in `nettl` and in the formatted header printed by `netfmt`. It is a string that can consist of alphanumerics (beginning with a letter) and can contain underscores. Blanks are not allowed.
- **Default Logging Class:** This is a mask containing the level of logging to be enabled when the logging facility starts up. This level can be changed by subsequent calls to `nettl`.
- **Subsystem Space Type:** This is a flag that identifies user-space subsystems, streams subsystems and kernel-space subsystems. The three types of subsystems are handled differently within the `nettl` command.
- **Subsystem Shared Library:** This is the subsystem formatter library that contains the subformatter functions.
Subsystem Formatting Function: This is the C function name used to call the function that supports formatting for the subsystem. This function must be contained in the subsystem formatter shared library.

Subsystem Options Function: This is the C function that is called to process options specified by the user in the netfmt options file. Only the OTS, LAN, ARPA, and X.25 subsystems use this feature. The formatter uses this function to set up global filtering and formatting information. This function must be contained in the subsystem formatter shared library.

Subsystem Group Name: Each subsystem belongs to some logical group, usually a product. This group name is included on the banner printed during formatting. Although this group name can be any ASCII string, it should definitely contain the subsystem product name.

Subsystem Formatter Message Catalog: This is the name of the message catalog used by the subsystem formatter functions. This is typically an unqualified name, that is, the base name of the catalog with no path or .cat extensions. For example, the default message catalog for the formatter is netfmt.cat, and it resides in the default NLS directory, /usr/lib/nls/msg/C. This could be specified simply as netfmt. However, if the message catalog does not reside in the default directory, the message catalog name must contain NLSPATH path constructors described. For example, for product xyz, the abc message catalog, /opt/xyz/lib/nls/msg/C/abc.cat, would be specified as /opt/xyz/lib/nls/%L/abc.cat. This is so the end user of the formatter can use other message catalogs and control them with the LANG and NLSPATH environment variables. This restriction requires subsystems to load their standard English catalog into the C directory under their nls paths (this is the standard place for the shipped message catalogs).

Sample Subformatter Configure Script

The filesset configure script should perform the configuration of all subsystems contained in the filesset. The following fragment is from a SDU control script to perform the configuration for example subsystems A, B and C:

```bash
#!/usr/bin/posix/sh
#
# Product:
# Fileset: NETTL-MIN
# configure
# 0(#) $Revision: 1.2 $
#
# (c) Copyright Hewlett-Packard Company 1993
#
set -a # Export all vars
exitval=0 # Anticipate success
: ${UTILS:="/usr/lbin/sw/control_utils"}
if [ ! -f $UTILS ]
then
    echo "ERROR: Cannot find $UTILS"
    exit 1
fi
set env
```
Tracing and Logging in LAN Drivers

Formatting Networking Trace/Log Messages

```bash
25 : ${FILESET:="NETTL-MIN*"}
26 : ${NETTLCONF:="${SW_ROOT_DIRECTORY}usr/sbin/nettlconf"}
27 : ${NETFMT:="${SW_ROOT_DIRECTORY}usr/sbin/netfmt"}
28
29 if [ ! -x "${NETTLCONF}" ] ; then
30   echo "ERROR: Cannot find ${NETTLCONF}"
31   exit 1
32 fi

34 # Subsystem A
35 $NETTLCONF -S -id 0 -name SUBSYSTEM_A -class 12 -kernel
36   -lib libsubsystem_A.so -msg subsys_A_msg
37   -fmtfn subsys_A_format -optfn subsys_A_get_options
38   -group 'SUBSYSTEM A Product' ||
39   exit 1 # nettlconf reports its own errors
40
42 # Subsystem B
43 $NETTLCONF -S -id 0 -name SUBSYSTEM_B -class 12 -kernel
44   -lib libsubsystem_B.so -msg subsys_B_msg
45   fmtfn subsys_B_format -optfn subsys_B_get_options
46   -group 'SUBSYSTEM B Product' ||
47   exit 1 # nettlconf reports its own errors
48
49 # Subsystem C
50 $NETTLCONF -S -id 0 -name SUBSYSTEM_C -class 12 -kernel
51   -lib libsubsystem_C.so -msg subsys_C_msg
52   fmtfn subsys_C_format -optfn subsys_C_get_options
53   -group 'SUBSYSTEM C Product' ||
54   exit 1 # nettlconf reports its own errors
55
57 ... Other subsystem configurations

59 # Test the configuration file
60 cmd_output=`$NETFMT -pc /dev/null 2>&1`
61 cmd_result=$?
62 if [ $cmd_result -ne 0 ] ; then
63   # The configuration file caused an error
64     echo "ERROR The $NETFMT command produced follwing error"
65     echo * messages while verifying configuration:
66     echo "$cmd_output"
67     exit 1
68 fi
71
72 exit 0
```

Subsystem Installation Testing

Subsystem developers must perform complete installation testing on their subsystems. As described in the previous section, the network trace/log facility is configured at installation time by a registration process that occurs in the subsystems configure script. This process tells the netfmt and nettl commands the IDs of the subsystems that exist on the system and gives information about how the subsystems are to be controlled and formatted. Only those subsystems that are registered are allowed to be turned on for logging and tracing or have their records formatted appropriately.

The nettlconf command does not check the parameters that are passed to it. The subsystem must check that the information to be stored in the configuration database is correct. Subsystems must test their installation for all possible environments, including multi-user systems, workstations, and diskless clusters.

The registration scheme has the potential to break tracing and logging for all subsystems if the configuration becomes corrupt or if the information that is given is invalid. Subsystems should test and review the procedures used to configure their subsystems into the network trace/log facility.
How to Test

The subsystems can perform the following steps to ensure correct installation and configuration. All of the steps can be performed in the target system where the driver subsystem is installed:

1. Install the subformatter library in the appropriate directory (default: /usr/lib/hpux32).
2. Configure subsystem information as explained in the “Configuring Subsystems into the System” section.
3. Stop and start nettl, refer to the manpages that shipped with the system on nettl (1m).
   — Starting nettl doesn’t give any error messages.
   — The “nettl -status” displays appropriate log class enabled for the subsystem as specified in the nettlgen.conf file.
4. Enable tracing, if applicable.
5. Generate log/trace messages.
6. Use netfmt(1m) command to format the log and trace files containing messages from the subsystem. Format the trace file using all three modes of formatting.
7. If console logging is enabled, check if the messages are displayed on the console.

Frequent Installation/Configuration Problems

The problems described in the following list are very common and can cause the configuration file to be unusable. All of these problems are preventable with proper understanding and testing of the subsystem configuration process:

- The subsystem subformatter library or message catalog cannot be found or opened except by superuser.
- A field in the nettlgen.conf subsystem configuration database file is corrupted.
- Symbols in the subformatter library conflict with symbols exported from other subformatter libraries of other subsystems. (This situation cannot occur if the +e option to ld is used when creating the subformatter library.)
- Symbols remain unresolved after netfmt has loaded the subformatter libraries of all configured subsystems.
- The function name of the subsys_N_format() or subsys_N_get_options() functions cannot be found in the specified subformatter library.
- The subsystem name or ID is in use by another subsystem. (This cannot happen if subsystems use the subsystem names and ID numbers assigned by the NetTL Team as described in the “Assign Subsystem ID” section.)
9 SAM Support for LAN Drivers

System Administration Manager (SAM) is an HP-UX system administration tool. For the HP-UX system administrator, it provides both GUI and TUI based interfaces to configure system resources such as file systems, network, and so forth. Starting with HP-UX version 11i, SAM has included support for configuring and managing IHV network interface drivers.

This chapter provides information for driver developers on how to write network interface drivers, such that the drivers are configurable and manageable using SAM. For detailed information on writing a network interface driver for HP-UX 11i v1, refer to Chapter 6, “Creating Networking Device Drivers.”

This chapter contains the following sections:

- “SAM and Network Driver Interface”
- “Network Configuration Parameters Supported by SAM”
- “Driver Configuration File and init Script”
- “Sample Driver Configuration File and init Script”
SAM and Network Driver Interface

This section describes the interface between SAM and network interface drivers. Support components are shown in Figure 9-1, “SAM and Network Driver Interface.”

Figure 9-1  SAM and Network Driver Interface
For SAM to interact with a network interface driver, SAM must know the driver name and the driver special file name. SAM retrieves this information from the driver configuration file. SAM also compiles a list of user configurable parameters by going through the driver configuration file. The driver configuration file is discussed in detail in the “Driver Configuration File and init Script” section and a sample driver configuration file is provided in the “Driver Configuration File” section.

Once a user has modified network parameters for a LAN interface, SAM calls the driver init script to pass the changes to the network interface driver. A driver init script typically calls the lanadmin command to execute the changes. For an IHV network interface driver the init script must call driveradmin. For more information about LAN commands see Chapter 7, “LAN Commands.” The driver init script is described in detail in the “Driver Configuration File and init Script” section and a sample driver init script is provided in with the ENET driver sources.

Once a user has modified network parameters for a LAN interface using SAM, they expect the modified settings to be applicable and restored across reboots. For this to happen, a driver startup script must source the driver configuration file and call the driver init script during system startup.

SAM also sends certain ioctl's and primitives directly to a network interface driver to retrieve some information. SAM uses an internal SAM executable /usr/sam/bin/laninfo command to list all the network interfaces on a system. A network interface driver must implement these ioctl's and primitives for SAM to work with it.
Network Configuration Parameters Supported by SAM

SAM provides a GUI-based configuration of Network Interface Cards. SAM supports the user configurable parameters, shown in Table 9-1, “Network Configuration Parameters.”

Table 9-1  Network Configuration Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP Address</td>
<td>Identifier by which the card is known on the IPv4 network.</td>
</tr>
<tr>
<td>IPv6 Address</td>
<td>Identifier by which the card is known on the IPv6 network.</td>
</tr>
<tr>
<td>Prefix Length</td>
<td>Specifies the length of the Subnet Prefix.</td>
</tr>
<tr>
<td>Subnet Mask</td>
<td>Tells the networking software which part of the IPv4 address to interpret as the subnetwork number.</td>
</tr>
<tr>
<td>Hostname</td>
<td>Current name of the system.</td>
</tr>
<tr>
<td>Aliases</td>
<td>Alternate names by which the system is known on the network.</td>
</tr>
<tr>
<td>Speed</td>
<td>Speed at which the Network Interface Card (NIC) operates. SAM supports 10, 100 and 1000 Mbps.</td>
</tr>
<tr>
<td>Duplex</td>
<td>This mode determines whether data can travel in one direction at a time or in both directions simultaneously. SAM supports HALF and FULL Duplex.</td>
</tr>
<tr>
<td>Autonegotiate</td>
<td>The means of negotiating modes of speed and duplex. SAM supports ON or OFF.</td>
</tr>
<tr>
<td>MAC address</td>
<td>The station address is a unique, 12-digit hexadecimal hardware address supplied with each network interface card.</td>
</tr>
<tr>
<td>Broadcast Address</td>
<td>Internet address with host ID portion consisting of all ones, which refers to all hosts on the network (available only on the IPv4 network).</td>
</tr>
<tr>
<td>MTU Size</td>
<td>Fixed upper limit on the amount of data that can be transferred in one physical frame.</td>
</tr>
</tbody>
</table>

SAM uses the ifconfig utility for setting the IP address, subnet mask and broadcast address. For configuring speed, duplex, autonegotiate, MAC address, and MTU size, SAM calls the driver init script.
Driver Configuration File and init Script

For SAM to configure and manage a network interface card, two files must be supplied by the networking driver developer:

- Driver Configuration File
- Driver init Script

A third file, Driver Startup Script, must also be provided by the network driver developer to restore the network interface card setting across reboots.

Driver Configuration File

The driver configuration file provides information to SAM about the driver. In addition, the driver configuration file is where SAM saves modifications made by a user.

The driver configuration file must be named as `hp<driver_name>.conf` format and it must be placed under the `/etc/rc.config.d` directory. For example, if the network interface driver name is `enet`, the driver configuration file must be named `hpenetconf`. A sample driver configuration file is located at the end of this chapter.

- `HP_<driver_name>_INIT_ARGS="..."`

  This entry specifies user configurable parameters supported by a network interface driver. The keywords for the parameters are described in Table 9-2, “Keywords.”

Table 9-2   Keywords

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>HP_&lt;driver_name&gt;_STATION_ADDRESS</code></td>
<td>Informs SAM to let a user modify a network interface MAC Address using SAM. For an ENET network interface driver, this keyword is <code>HP_ENET_STATION_ADDRESS</code>.</td>
</tr>
<tr>
<td><code>HP_&lt;driver_name&gt;_MTU</code></td>
<td>Informs SAM to let a user modify a network interface MTU using SAM. For an ENET network interface driver, this keyword is <code>HP_ENET_MTU</code>.</td>
</tr>
<tr>
<td><code>HP_&lt;driver_name&gt;_SPEED</code></td>
<td>Informs SAM to let a user modify the network interface Speed parameter. For an ENET network interface, this keyword is <code>HP_ENET_SPEED</code>.</td>
</tr>
</tbody>
</table>

The following is from the ENET driver configuration file. It informs SAM to permit a user to modify the MAC address, MTU and Speed for an ENET network interface using SAM.

```
HP_ENET_INIT_ARGS="HP_ENET_STATION_ADDRESS HP_ENET_MTU HP_ENET_SPEED"
```

For example, the following line is from the ENET driver configuration file:

- `IHV_DLPI_DEVICE_NAME = /dev/<driver_name>`

  This entry directs SAM to the device special filename for the network interface driver. This is required by SAM to be able to send direct ioctl to the network interface driver. The ioctl that SAM sends are listed in the “Driver init Script” section.

```
IHV_DLPI_DEVICE_NAME=/dev/enet
```
Driver init Script

The driver init script acts as a nexus between SAM and the network interface driver. SAM sends requests to modify network configuration parameters to the driver via the driver init script. During the boot-up sequence the driver startup script calls the driver init script to modify network interface parameters.

The driver init script must be named in the format `hp<driver_name>_init` and it must be placed under the `/usr/sbin` directory. For example, `hpenet_init`.

After the user has made modifications to driver network parameters, SAM saves the changes to the driver configuration file and calls the driver init script. The driver init script calling convention is described here:

```
/usr/sbin/hp<driver_name>_init start <major#> <instance#> <interface name>
    <nmid> <station address> <mtu>
    <interface speed>
```

The parameter definitions are:

- **major#**: Major number of the driver. The driver init script should not use this parameter. It is described here only for backward compatibility.
- **instance#**: Instance of the network interface to be modified.
- **interface name**: Network interface name, as shown as `lanscan` output. The driver init script should not use this parameter. It is described here only for backward compatibility.
- **nmid**: Network management ID for the network interface.
- **station address**: New MAC address; set the specified NIC's MAC address with the station address.
- **mtu**: New MTU value; set the specified NIC's MTU with this interface.
- **speed**: New link speed setting for NIC.

The valid values are:

- `10HD`: 10Mbps, half-duplex
- `10FD`: 10Mbps, full-duplex
- `100HD`: 100Mbps, half-duplex
- `100FD`: 100Mbps, full-duplex
- `1000HD`: 1000Mbps, half-duplex
- `1000FD`: 1000Mbps, full-duplex
- `AUTO_ON`: Auto-negotiation ON

For example, to set interface network MAC address to `0x000629BE051C`, MTU size to 1200, and speed settings to 100 full-duplex, SAM can call the ENET driver init script as follows:

```
/usr/sbin/hpenet_init start 119 1 enet1 6 0x000629BE051C 1200 100FD
```

Any message from the driver init script after successfully carrying out the set operation will be redirected to `/dev/null`. If the driver init script fails, the error message provided by the script will be displayed by SAM.

Typically a driver init script will call the `lanadmin` tool to set the values of the driver. The init script for IHV driver must however call its own command, `driveradmin`. For more information about LAN commands, see Chapter 7, “LAN Commands.” Refer to the sample driver init script `hpenet_init` provided with the ENET driver sources.
Sample Driver Configuration File and init Script

This section discusses the driver configuration file and init script.

Driver Configuration File

The driver configuration file must be named in the \texttt{hp<driver_name>conf} format and it must be placed under the \texttt{/etc/rc.config.d} directory. For example, if the driver name is enet, the driver configuration file name will be \texttt{hpenetconf}.

Following is the sample driver configuration file:

\begin{verbatim}
# hpenetconf: contains configuration values for ENET LAN interfaces
#
# HP_ENET_INTERFACE_NAME   Name of interface (lan0, lan1...)  
# HP_ENET_STATION_ADDRESS  Station address of interface 
# HP_ENET_SPEED            Speed and duplex mode 
#   Can be one of : 10HD, 10FD, 100HD, 100FD and AUTO_ON. 
# HP_ENET_MTU              Maximum Transmission Unit (MTU) 
#   Integer value between 257 and 1500, inclusive. 
#
# The station address and duplex are set through the lanadmin(1m) 
# command.
#
# IHV_DLPI_DEVICE_NAME=/dev/enet
HP_ENET_INTERFACE_NAME[]= 
HP_ENET_STATION_ADDRESS[]= 
HP_ENET_SPEED[]= 
HP_ENET_MTU[]= 

# The HP_ENET_INIT_ARGS are reserved. They are NOT user 
# changable.

HP_ENET_INIT_ARGS="HP_ENET_STATION_ADDRESS HP_ENET_SPEED HP_ENET_MTU"

# End of hpenetconf configuration file
\end{verbatim}

Driver init Script

The driver init script must be named in the \texttt{hp<driver_name>_init} format and it must be placed under the \texttt{/usr/sbin} directory. For example, if the driver name is enet, then the driver init script name will be \texttt{hpenet_init}. An example init script is provided with the ENET sample driver.
10 CKO and Transport IOCTLs

This chapter explains the interaction between the HP-UX transport layer and the DLS providers to create an efficient data transfer mechanism between layers in a networking stack. The HP-UX transport layer supports DLS providers that adhere to DLPI 2.0 standard. Certain, HP specific extensions to DLPI 2.0 standard must also implemented when interaction with the HP-UX transport layer.
DLPI IOCTL

The HP-UX transport stack uses a set of IOCTLs for the in-kernel STREAMS Data Link Service (DLS) user (e.g., IP module) to negotiate the driver features and set up a Fastpath. These IOCTLs are HP-UX specific and were originally introduced in HP-UX 11.0. These IOCTLs are listed and described.

DLPI_IOC_DRIVER_OPTIONS

The DLPI_IOC_DRIVER_OPTIONS IOCTL is used by IP to obtain additional information from a DLS provider regarding the driver’s capabilities. The DLS provider communicates to IP of the capabilities which IP may use to enhance performance.

While configuring an interface for IP, the IP sends an M_IOCTL message to the DLS provider. The M_IOCTL message block contains an iocblk structure with ioc_cmd == DLPI_IOC_DRIVER_OPTIONS, ioc_count == size of (driver_ops_t). Following the M_IOCTL message block, pointed by b_cont of the M_IOCTL message block is a M_DATA message block containing a driver_ops_t structure.

For information on the structure, refer to Table 10-1, “driver_ops_t Data Structure.”

Table 10-1  driver_ops_t Data Structure

<table>
<thead>
<tr>
<th>Name</th>
<th>Include File</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>driver_ops_t</td>
<td>&lt;sys/dlpi_ext.h&gt;</td>
<td>DLPI/XPORT options negotiations structure. This structure will be passed as part of the DLPI_IOC_DRIVER_OPTIONS IOCTL from XPORT.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Members</th>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>driver_ops_type</td>
<td>uint32_t</td>
<td>Features supported by XPORT. The features are bit wise flags.</td>
</tr>
<tr>
<td></td>
<td>driver_ops_type_1</td>
<td>uint32_t</td>
<td>Reserved. Must be set to “0”</td>
</tr>
<tr>
<td></td>
<td>driver_ops_type_2</td>
<td>uint32_t</td>
<td>Reserved. Must be set to “0”</td>
</tr>
</tbody>
</table>

For information on possible features supported by transport layer, refer to Table 10-2, “Capabilities Understood by IP.”
The `driver_ops_type` field is used by the IP module to pass a bit mask of flags, indicating the driver capabilities that IP is inquiring about. The driver is expected to look at this bit mask, perform whatever action it needs to enable the options asked for, and then to return an `M_IOCACK` containing the same data structure with `driver_ops_type` set to a bit mask showing the options the driver supports. The transport option `DRIVER_PRI` was added to enable VLAN. Currently, it is not supported for all non-HP network interface drivers, therefore all non-HP drivers must turn this bit off before replying to IP.

The following code snippet was taken from the ENET sample driver for HP-UX version 11i v1. It shows how network drivers must respond to the `DLPI_IOC_DRIVER_OPTIONS` IOCTLs. Note that transport options may be added to HP-UX at any time and therefore recommended that IHV developers make modifications to their network drivers such that it behaves as the following code. This code turns off all the options flags at the beginning. Only the options supported by the driver are turned on.

<< enet code snippet here >>
DL_IOC_HDR_INFO

A DLS provider normally expects to receive datagrams in the form of a DL_UNITDATA_REQ message. The DL_UNITDATA_REQ primitive contains the hardware source and destination addresses from which a DLS provider can construct a packet header and place in front of the IP packet before transmitting the packet. The creation of the packet header is not performance tuned as it has to be constructed for every packet that is sent out. Many connection-oriented protocols like TCP, once the connection has been established, will have identical information in every DL_UNITDATA_REQ. To avoid constructing a packet header every time, a DLS provider can support an IOCTL which permits the transport layer like TCP/IP to request a packet header template during configuration of TCP/IP on a per-interface basis. The packet header template will enable transport layer to add the packet header before sending the packet to a DLS provider. This mechanism is called Fastpath.

The negotiation of Fastpath happens when the transport layer sends a M_IOCTL type message with ioc_cmd set to DLPI_IOC_HDR_INFO. The b_cont of the M_IOCTL message block will link a M_DATA type message block which contains dl_unitdata_req_t structure which is used in the creation of DL_UNITDATA_REQ primitive.

For a driver that does not support Fastpath capability, a M_IOCNAK message should be sent back, in which case the transport layer will use only the DL_UNITDATA_REQ primitive for data transfer.

A Native STREAMS DLPI network interface driver must implement this IOCTL, if it intends to support the Fastpath capability. Refer to Figure 10-1, “Fastpath,” for Fastpath negotiation message formats.

Figure 10-1  Fastpath

a) M_IOCTL message from transport layer to request Fastpath Header

```
M_IOCTL  M_DATA
 DL_IOC_HDR_INFO  DL_UNITDATA_REQ
```

b) Fastpath M_IOCACK response from DLPI with Link Level header appended

```
M_IOCACK  M_IOCTL  M_DATA  M_DATA
 DL_IOC_HDR_INFO  DL_UNITDATA_REQ  CKO header (if supported) + LLC Header
```

In response to the DL_IOC_HDR_INFO IOCTL, a DLS provider must return the packet header template which contains the CKO header and the Link level header template with information filled in. The response will be a M_IOCACK IOCTL as shown in Figure 10-1, “Fastpath.” The CKO header must be present only if the DLS provider supports the Checksum Offload feature. The second message block contains the DL_UNITDATA_REQ passed to DLS provider. The third message contains a M_DATA message block with the LLC Header template created for that particular stream.

The CKO header part of the packet header template is used by the transport layer to communicate checksum offload related information.
Checksum Offload

Checksum offload is a host assistance feature provided by a Network Interface Card (NIC). The NIC accelerates checksum generation and verification by performing this function for the host CPU (offloading it). This network card feature allows checksum calculation (including pseudo headers) for IP, TCP and UDP packets. Enabling checksum offload on adapters leads to a reduction in host CPU utilization and improved performance due to reduced cache misses (avoiding cache pollution with the data to be checksummed). The network card can also support checksum offload of fragmented packets; when, on the transmit side, the host sets up the buffers in such a way that fragmented frames can have their UDP or TCP checksums calculated over the entire datagram.

Three support modules are required for the host to exploit the NIC checksum offload feature:

1. Hardware: The NIC should have the feature to calculate checksum for a TCP/UDP/IP packet.
2. Driver: When checksum offload is enabled, it indicates to the upper layer that it is no longer required to calculate the checksum and places the necessary checksum calculating information in the NIC.
3. Transport Stack: Provides the checksum information (explained later) to the driver.

Adding Checksum Offload Support

There are two kinds of checksums that can be calculated by the NIC:

1. TCP/UDP checksum
2. IP header checksum

The NIC must provide both TCP and UDP checksum offload support. TCP/UDP checksum can be calculated with or without a pseudo header. Therefore, the NIC enables the driver to indicate which checksums need to be calculated and if a TCP/UDP checksum should be calculated with or without a pseudo header.

These options can be set differently for transmit and receive packets.

The behavior of HP-UX transport stack for checksum offload:

- Checksum offload is supported only in IP fast path.
- Transport stack cannot take advantage of IP header checksumming because the IP layer calculates the header checksum for all adapters.
- Transport stack calculates the pseudo-header checksum by default and places it in the TCP/UDP checksum field on outbound. It would be good if the adapter just used this field and did not compute the pseudo-header checksum again.

Initialization

Indication that the checksum offload feature is enabled is kept in a per instance structure, i.e., hwift structure. The DRV_CKO (0x8) flag is set in the features field of the hwift structure.

After binding the stream, the upper layer sends a DLPI_IOC_DRIVER_OPTIONS IOCTL request to the driver to check if the device instance supports checksum offload. The driver_dlpi_ioctl() function (driver routine which handles IOCTL requests) checks the features field of hwift and ACKs the request if DRV_CKO is set.
**driver_dlpi_ioctl()**

Checks if the DRV_CKO flag is set in the features field of the hwift structure and returns back the driver_ops_t structure with DRVCKO flag set or cleared depending on whether checksum offload is enabled or not. This function must implement the DLPI_IOC_DRIVER_OPTIONS IOCTL.

**Transmit Path**

The NIC calculates the IP, TCP, and/or UDP checksum and inserts it into the outgoing frame. The NIC must provide a method for the driver to inform the NIC on a per frame basis. These flags can be provided in a transmit descriptor, where each transmit descriptor refers to a frame. The NIC can also provide the feature of calculating the TCP/UDP checksum for an IP fragmented packet by accumulating the checksums of each IP fragment and inserting the final checksum into the TCP/UDP header. For checksum calculation of fragmented packets, the NIC must provide extra bits to be set in the descriptor to mark the fragments to be considered and the end of the fragment etc. Also there would be a condition that all fragments must be placed in send ring consecutively and in the correct order.

For drivers with DRV_CKO enabled, the upper layer passes the **Message Blocks** (mblks) to the driver with the MSGCKO flag set in the b_flags field; this indicates that the NIC must calculate checksum for the packet. An IP fragment chain comes down to the driver as a series of mblks connected through the b_cont field with MSGCKO bit being set in the b_flag field of each mblk which is the start of the fragment.

For the mblks with the MSGCKO bit flag set, the first 8 bytes denote the Checksum Offload (CKO) information. So the driver_build_hdr() function (driver routine which builds the MAC header) builds the MAC header with an extra 8 bytes at the front of the MAC header before returning the header to the upper layer in fast path. Checksum offload structure (8 bytes) which comes from the upper layer contains the following information:

- Starting location (of the data to be checksummed)
- End location (pointing to the last byte of data)
- Seed information (pseudo-header checksum)
- Insert location (pointing to the location where the checksum is to be inserted)
- Insert flag (indicating whether the checksum is to be inserted in this fragment or not).

Checksum offload information (8 bytes) structure is defined in the header file `<cko.h>`.

```c
struct cko_info {
    u_char      cko_type;               /* Checksum assist control field */
#define CKO_INSERT             0x0001  /* cko_seed value need to be used */
#define CKO_OUTBOUND    0x0002
#define CKO_CMD         0x0004
#define CKO_ALGO_NONE   0x0008
#define CKO_ALGO_UDP      0x0010   /* UDP packet */
#define CKO_ALGO_TCP      0x0020   /* TCP packet */
#define CKO_SUM_FLUSH   0x0040  /* Flush checksum in card */
#define CKO_ETC                    0x0080 /* Embedded Trailer Cksum pkt */
    u_char      cko_start;              /* Checksum starting offset */
    u_short     cko_stop;               /* Checksum stop offset */
    u_short     cko_insert;            /* Checksum insert offset */
    u_short     cko_seed;              /* Checksum seed value */
};
```
The driver analyzes the previous checksum offload information and saves the required information in a transmit descriptor of the packets in device specific manner. Then the driver strips off the first 8 bytes of each mblk with MSGCKO set before passing the packet to the NIC.

The following sections are some of the driver functions in the transmit path and the various purposes they fulfill.

**driver_wput()**

This function is the driver's write side put routine, which receives all the requests from DLPI user (upper layer). For Fastpath, this function calls `driver_dlpi_cko_fast_out()` if checksum offload is enabled (features &DRV_CKO), otherwise it calls `driver_if_resolved_output()`.

**driver_dlpi_build_hdr()**

This function builds the MAC header and if DRV_CKO is enabled it also allocates an extra 8 bytes in front of the MAC header. For fast path, this function is called when the upper layer sends an IOCTL request DLPI_IOC_HDR_INFO during the fast path setup. For regular path, this function is called by the driver to setup MAC header before sending a packet.

**driver_dlpi_cko_fast_out()**

This function is called by `driver_wput()` for fast path and also for an interface for which checksum offload is enabled. The following steps explain what the function does related to checksum offload:

1. IP fragment chain comes down to the driver as a series of mblks connected through the b_cont field with the MSGCKO bit set in the b_flag field of each mblk that is the start of the fragment. Every IP fragment is separated in such a way that the b_next field of the first mblk of each fragment points to the first mblk of the next fragment as shown Figure 10-2, “Fragments.”

2. For the message blocks which have MSGCKO set, the function checks if the CKO_INSERT bit is tuned on in the insert flag of the checksum information. If the CKO_INSERT bit is on, then it inserts the seed value (present in cko_info) which is the pseudo header checksum at the location pointed by cko_insert.

3. Calls `driver_if_resolved_output()`.
**driver_dlpi_unitdata_out()**

This function is called by `driver_wput()` for all regular path data packets (DL_UNITDATA_REQ):

1. Calls `driver_dlpi_build_hdr()` to build the MAC header which allocates an extra 8 bytes at the front of the MAC header if DRV_CKO is enabled.
2. Calls `driver_if_resolved_output()`.

**driver_if_resolved_output()**

This function is called for both regular path and fast path packets:

1. Strips off checksum information (8 bytes) from all message blocks.
2. For any packets with the MSGCKO bit set, it saves the `cko_start` and `cko_insert` fields of the `cko_info` structure for later use.
3. For any packets with the MSGCKO bit set, it calculates the frame length using the `cko_stop` field of the `cko_info` structure.
4. Calls `driver_hw_req` to transmit packets.

**Receive Path**

The NIC calculates the IP, TCP and/or UDP checksum for each incoming frame. The calculated checksum can be kept in a receive descriptor from which the driver reads the values. Transport stack can use the TCP/UDP checksum value directly for unfragmented packets. For fragmented packets, the stack needs to add up the TCP/UDP checksums for all the fragments to obtain the final checksum.
The driver reads the TCP/UDP checksum value from the device registers or receive descriptor (depending on the NIC) and saves the value in `b_quad[3]` of `mblk` for which checksum corresponds. Also MSGCKO is set in the “b_flags” of the `mblk`. Since the IP layer always calculates the IP header checksum, the IP header checksum calculation is not enabled on the device.

If \((mblkp->b_rptr - mblkp->b_datap->db_base) > 8\). In other words, if there are 8 bytes of space between `db_base` and `b_rptr`, then the driver moves `b_rptr` back by 8 bytes to cover checksum information (structure `cko_info`). Otherwise, the driver allocates a new `mblk` for checksum information (`cko_info`). The driver then saves the checksum value (`b_quad[3]`) in the `cko_insert` field of the `cko_info` structure (which will be read by the IP layer). For fragmented packets, the transport stack adds up the TCP/UDP checksums for all the fragments to obtain the final checksum.

The device calculates the checksum for each frame and saves it in the receive descriptor. The `driver_receive_pkts()` function reads the checksum value from receive descriptor and saves it in `b_quad[3]` of the message block and sets the MSGCKO bit in the `b_flags` field of the message block.

**driver_dlpi_fast_in()**


If \((mblkp->b_rptr - mblkp->b_datap->db_base) > 8\), then it moves `b_rptr` back by 8 bytes to cover the checksum information (structure `cko_info`). Otherwise, it allocates a new `mblk` for checksum information. Then it saves the checksum value (`b_quad[3]`) in the `cko_insert` field of `cko_info` structure, which will be read by the IP layer.

**Packet Flow Diagram**

The following diagram shown in Figure 10-3, “Packet Flow Diagram,” shows how the packet traverses the driver.
Figure 10-3    Packet Flow Diagram

![Packet Flow Diagram]

lan_dlpikrn.h:

#define DRV_CKO 0x8 /* Indicates that particular interface has checksum offload enabled */

stream.h:

#define MSGCKO 0x2000 /* Checksum offload is expected for particular packet */

dlpi.h:

#define DL_CLDLS 0x02 /* support connectionless data link service */
#define DL_ETHER 0x4 /* Ethernet Bus, LLI Compatibility */
#define DL_CSMACD 0x0 /* IEEE 802.3 CSMA/CD network, */

if_ether.h:
#define ETHER_HLEN 14 /* header length for ethernet packets */
#define ETHER_PKT 0
#define SNAP8021_PKT 2
#define IEEE8021_PKT 5

if_ieee.h:
#define MIN_IEEE8022_HLEN 3 /* minimum length for 802.2 header */
#define SNAP_802_2_HLEN 8 /* minimum length for 802.2 snap header*/

#define CKO_INFO_SIZE 8

int
driver_wput (queue_t *q, mblk_t *mblkp)
{
    
    /* Get the pointer to the per stream structure */
    driver_dlpi_data_t *hp_dlpi_datap = (driver_dlpi_data_t *)q->q_ptr;

    if (mblkp->b_datap->db_type == M_DATA) { /* Fast Path */
        if ((hp_dlpi_datap->service_mode == DL_CLDLS) &&
            hp_dlpi_datap->fast_path &&
            (hp_dlpi_datap->curr_state == DL_IDLE)) {
            if(hp_dlpi_datap->hwiftp->features & DRV_CKO) {
                
                driver_dlpi_cko_fast_out (hp_dlpi_datap, mblkp);
                
            } else {
                
                driver_if_resolved_output (hp_dlpi_datap, mblkp);
                
            }
        } else if (hp_dlpi_datap->service_mode == DL_CODLS) {
            /* connection oriented part */
            
        } else {
            /* Error : free messages */
            
        }
    }
}
/* The case of M_PROTO and UNITDATA request (Regular path) */
if ((mblkp->b_datap->db_type == M_PROTO) &&
    (*((uint32_t *)mblkp->b_rptr) == DL_UNITDATA_REQ)) {
    ...
    driver_dlpi_unitdata_out (hp_dlpi_datap, mblkp);
    ...
}

/* Handle other messages (M_IOCTL, M_PCPROTO etc) */
...

mblk_t *
driver_dlpi_cko_fast_out (driver_dlpi_data_t *hp_dlpi_datap, mblk_t *mblkp)
{
    hw_ift_t                *hwiftp = hp_dlpi_datap->hwiftp;
    caddr_t                 drv_datap;
    intptr_t                error;
    mblk_t                  *finger_mblkp;
    mblk_t                  *cur_frag_mblkp, *next_frag_mblkp;
    finger_mblkp = cur_frag_mblkp = mblkp;

    /* Break IP fragment chain */
    while (finger_mblkp) {
        if (finger_mblkp->b_cont &&
            (finger_mblkp->b_cont->b_flag & MSGCKO)) {
            /* b_cont here is the beginning of next frag */
            next_frag_mblkp = finger_mblkp->b_cont;
            finger_mblkp->b_cont = NULL;

            /* formulate the train */
            cur_frag_mblkp->b_next = next_frag_mblkp;
            VASSERT(next_frag_mblkp->b_next == NULL);

            /* reset finger_mblkp to the head of the rest */
            finger_mblkp = cur_frag_mblkp = next_frag_mblkp;
            continue;
        }
        finger_mblkp = finger_mblkp->b_cont;
    }
    cur_frag_mblkp = mblkp;

    /* Insert pseudo header checksum into TCP/UDP checksum field in packet */
    while (cur_frag_mblkp) {
        if (!driver_dlpi_add_xport_cksum(hp_dlpi_datap, cur_frag_mblkp)) {
            ...
/* free the entire stream message */
cur_frag_mblkp=mblkp;
while (cur_frag_mblkp) {
    next_frag_mblkp=cur_frag_mblkp->b_next;
    freemsg(cur_frag_mblkp);
    cur_frag_mblkp=next_frag_mblkp;
}
return driver_dlpi_error_ack(NULL, DL_UNITDATA_REQ, EINVAL, DL_SYSERR);
}
cur_frag_mblkp = cur_frag_mblkp->b_next;
}

. . .
driver_if_resolved_output(hp_dlpi_datap, mblk);
}

/* Inserts pseudo header checksum into checksum field for TCP/UDP */

int
driver_dlpi_add_xport_cksum (driver_dlpi_data_t * hp_dlpi_datap, mblk_t *mblkp)
{

    struct cko_info      *cko_info_in, cp_cko_info_in;
    uint16_t             *cksum_addr;
    uint32_t             insert_len, cko_offset;
    mblk_t               *mblkp, *tmp_mblkp;

    mblkp=(mblk_t *)datap;
    cko_info_in = (struct cko_info *)mblkp->b_rptr;
    cs_block = (fddi_xmit_cs_t *)mblkp->b_rptr;

    if (cko_info_in->cko_type & CKO_INSERT) {
        tmp_mblkp = mblkp;
        /* We must adjust the insert length to include the llc header
         * because the insert_len calculated by the transport does
         * not include the llc header length.
         */
        insert_len = cko_info_in->cko_insert + hp_dlpi_datap->fast_path_llc_length;
        while (insert_len > 0) {
/* Find the TCP header’s Checksum field. */
            if (insert_len < (tmp_mblkp->b_wptr - tmp_mblkp->b_rptr)) {
                cksum_addr = (uint16_t *)(tmp_mblkp->b_rptr + insert_len);
                insert_len = 0;
            }else {
                insert_len = insert_len - (tmp_mblkp->b_wptr - tmp_mblkp->b_rptr);
            tmp_mblkp = tmp_mblkp->b_cont;
        }else {
    }
CKO and Transport IOCTLS

Checksum Offload

Chapter 10

```c
*cksum_addr = cko_info_in->cko_seed;
}
}

int
driver_dlpi_build_hdr (driver_dlpi_data_t *hp_dlpi_datap, mblk_t *mblkp,
uint8_t **llc_ptr)
{
    mblk_t *llc_hdr_mblkp, *temp = mblkp;
    caddr_t llc_hdrp, llc_hdr_datap;
    ...

    if (!(llc_hdr_mblkp = allocb(128, BPRI_HI)))
        return (0);
    llc_hdrp = (caddr_t)llc_hdr_mblkp;
    if (hwiftp->features & DRV_CKO) {
        llc_hdr_datap = (caddr_t)llc_hdr_mblkp->b_rptr + 8; /* 8 bytes for cko_info */
    } else {
        llc_hdr_datap = (caddr_t)llc_hdr_mblkp->b_rptr;
    }

    /* Then create MAC header using llc_hdr_datap */
    ...
}

mblk_t *
driver_dlpi_unitdata_out (driver_dlpi_data_t *hp_dlpi_datap, mblk_t *mblkp)
{
    caddr_t llc_hdrp;
    mblk_t *llc_hdr_mblkp;
    ...

    /* driver_dlpi_build_hdr() is called to build MAC header */
    if(!driver_dlpi_build_hdr(hp_dlpi_datap, mblkp, (uint8_t **)&llc_hdrp)) {
        if(llc_hdrp)
            freeb((mblk_t *)&llc_hdrp);
        return hp_dlpi_uderror_ind(hp_dlpi_datap, mblkp, 0, DL_BADADDR);
    }
    llc_hdr_mblkp = (mblk_t *)&llc_hdrp;
    linkb(llc_hdr_mblkp, mblkp->b_cont);
    ...
    driver_if_resolved_output(hp_dlpi_datap, llc_hdr_mblkp);
}
int
driver_if_resolved_output (driver_dlpi_data_t * hp_dlpi_datap, mblk_t *mblkp)
{
    register caddr_t       datap;
    uint32_t               buf_len;
    struct cko_info        *cko_info;
    uint32_t               fast_path_pkt_type, fast_path_llc_length;
    uint8_t                css;            /* Checksum start */
    uint8_t                cso;           /* Checksum insert offset */
    . . .
    datap = (caddr_t) mblkp->b_rptr;
    buf_len = ((int)(mblkp->b_wptr) - (int)(mblkp->b_rptr));
    fast_path_pkt_type = hp_dlpi_datap->fast_path_pkt_type;
    fast_path_llc_length = hp_dlpi_datap->fast_path_llc_length;

    if(buf_len != NULL) {
        if(mblkp->b_flag & MSGCKO){
            kco_info = (struct cko_info *)(datap);
            /*  Move datap by 8 bytes to strip off cko_info (8 bytes) */
            datap += CKO_INFO_SIZE;
            mblkp->b_rptr = (uint8_t *)(datap);
            buf_len -= CKO_INFO_SIZE;    /* decrease buf_len by 8 bytes */
            if (cko_info->cko_type & CKO_INSERT) {
                /* First fragment contains the TCP/UDP header where the checksum
                 * is to be inserted.
                 */
                cso = (int) (fast_path_llc_length + cko_info->cko_insert - CKO_INFO_SIZE);
                css  = (int) (fast_path_llc_length + cko_info->cko_start - CKO_INFO_SIZE);
            } else {
                insert_len = 0;
            }
            /* calculate packet length using cko_info->stop */
            if (buf_len >= ETHER_HLEN) {
                if (fast_path_pkt_type == ETHER_PKT) {
                } else {
                    uint16_t *llc_lengthp;
                    llc_lengthp = (uint16_t *)(datap + LAN_LLC_LEN_OFFSET);
                    if (fast_path_pkt_type == IEEE8023_PKT)
                        *llc_lengthp = kco_info->cko_stop + MIN_IEEE8022_HLEN + 1;
                    else if (fast_path_pkt_type == SNAP8023_PKT)
                        *llc_lengthp = kco_info->cko_stop + SNAP_802_2_HLEN + 1;
                }
            } else {
                . . .
            }
        }
    }
}
status = EINVAL;
cso = css = 0;
}
} else { /* non MSGCKO mblk */
datap += CKO_INFO_SIZE;
        mblkp->b_rptr = (uint8_t *) (datap);
        buf_len -= CKO_INFO_SIZE; /* decrease buf_len by 8 bytes */
} else {
        /* first mblk has a zero length buffer */
        status = EINVAL;
}
/* Use css & cso for filling transmit descriptor with checksum information */

. . .

driver_hw_req (hp_dlpi_datap->enetiftp, LAN_REQ_WRITE, mblkp);

. . .

} void
driver_dlpi_fast_in (driver_dlpi_data_t *hp_dlpi_datap, mblk_t *mblkp, int pkt_fmt)
{
uint8_t   *llc_hdrp;
mblk_t     *tmp_mblkp;
uint32_t   llc_hdr_size = 0;
queue_t    *q;

llc_hdrp = (uint8_t *) mblkp->b_rptr
uint32_t   llc_hdr_size = 0;

if (pkt_fmt == ETHER_PKT) {
        llc_hdr_size = ETHER_HLEN;
} else {
        /* Determine the LLC header size to strip off */
        switch (pkt_fmt) {
        case IEEE8023_PKT:
                llc_hdr_size = IEEE8023_HLEN;
                break;
        case SNAP8023_PKT:
                llc_hdr_size = SNAP8023_HLEN;
                break;
        default:
                /* Unsupported packet type received, drop it. */
                freemsg(mblkp);
                break;
        }
/* Strip off LLC header for both MSGCKO and non-MSGCKO packets */
while(llc_hdr_size) {
    if ( (mblkp->b_wptr - mblkp->b_rptr) > llc_hdr_size ) {
        mblkp->b_rptr += llc_hdr_size;
        llc_hdr_size = 0;
    } else {
        llc_hdr_size -= (uint32_t)(mblkp->b_wptr - mblkp->b_rptr);
        tmp_mblkp = mblkp;
        mblkp = mblkp->b_cont;
        if ( !mblkp ) { /* let go pkts with just llc hdrs*/
            mblkp = tmp_mblkp;
            mblkp->b_rptr = mblkp->b_wptr ;
            break;
        }
        freeb(tmp_mblkp);
    }
}

/* Use the LLC header left over above to store the CKO data!! */
* Driver must guarentee that the LLC header is contained in the same mblk.
*/
/* Determine if this packet contains checksum information. */
if ( ((mblkp->b_flag & MSGCKO) && (hp_dlpi_datap->hwiftp->features & DRV_CKO)) ) {
    mblk_t *cko_mblkp;
    struct cko_info *cko_info_in;
    uint16_t check_sum;

    check_sum = (uint16_t)mblkp->b_quad[3];
    if ( (mblkp->b_rptr - mblkp->b_datap->db_base) >= sizeof(struct cko_info)) {
        mblkp->b_rptr -= sizeof(struct cko_info);
        cko_info_in = (struct cko_info *)mblkp->b_rptr;
        cko_info_in->cko_insert = check_sum;
    } else {
        /* Allocate mblk to hold checksum information. */
        cko_mblkp = allocb(sizeof(struct cko_info), BPRI_HI);
        if (!cko_mblkp) {
            freeb(mblkp);
            return;
        }
    }
}
cko_mblkp->b_datap->db_type = M_DATA;
cko_info_in = (struct cko_info *)cko_mblkp->b_rptr;
cko_info_in->cko_insert = check_sum;

/* put the new cko_mblkp as the head mblk */
cko_mblkp->b_cont = mblkp;
cko_mblkp->b_flag |= MSGCKO;

/* MSGCKO should be set only at the beginning mblk */
mblkp->b_flag &= ~MSGCKO;

mblkp = cko_mblkp;
}
}
/*
 * Send message directly to upstream module.
 */
(void)putnext(hp_dlpi_datap->queue_ptr, mblkp);
return;
}

void
driver_dlp_process_ioctl (driver_dlpi_data_t *hp_dlpi_datap, mblk_t *mblkp)
{
    struct iocblk  *ioctl_mblkp;
    int             cmd, len = 0, retval;

    ioctl_mblkp = (struct iocblk *)mblkp->b_rptr;
    cmd = ioctl_mblkp->ioc_cmd;

    switch(cmd){
    case DLPI_IOC_DRIVER_OPTIONS:
        ops = (driver_ops_t *)mblkp->b_cont->b_rptr;
        if((ops->driver_ops_type & DRIVER_CKO) && !(hwiftp->features & DRV_CKO))
            ops->driver_ops_type &= ~DRIVER_CKO;
            ...
            qreply(q, mblkp);
            return;
    . . .
This chapter provides an overview of the Mass Storage I/O Stack in HP-UX. The Mass Storage I/O Stack includes SCSI Device Drivers (also known as SCSI Class Drivers), SCSI Services, and SCSI Transport Drivers (also known as SCSI Interface Drivers). The SCSI Transport Drivers are responsible for controlling the hardware that transports the SCSI protocol from the host system to the end SCSI device (LUN). Today in HP-UX there are parallel SCSI (pSCSI) and Fibre Channel (FC) transport drivers, which control pSCSI and FC Host Bus Adapters (HBAs). In the future there will be iSCSI transport drivers which control iSCSI HBAs or virtual interfaces.

The following sections are provided in this chapter:

- “Overview of HP-UX Mass Storage I/O Stack”
- “SCSI Addressing Paradigm”

For details on writing a SCSI Class Driver, see Chapter 13, “Writing SCSI Device Drivers.” For details on writing a SCSI Transport Driver, see Chapter 12, “Writing a SCSI Interface Driver.”
Overview of HP-UX Mass Storage I/O Stack

The following Figure 11-1, “Mass Storage Stack,” illustrates the Mass Storage I/O Stack in HP-UX, its key components, and the key modules that it interacts with.

Figure 11-1 Mass Storage Stack

The SCSI subsystem can be broadly divided into three layers:

1. SCSI class drivers (peripheral device drivers)
2. SCSI Services
3. SCSI transport drivers (HBA drivers)

Above the SCSI Subsystem are upper level modules such as volume managers and other pseudo drivers, file systems, and raw I/O access code, which can be used to access the SCSI subsystem. Further up the stack, user applications and I/O commands access the SCSI subsystem via one of these upper level modules.

Off to the side of the SCSI subsystem are WSIO and kernel services which the drivers can call to get themselves configured into the system and attached to their hardware, obtain resources, start and stop timers, map their card’s memory for PIO and setup system memory for DMA, etc. WSIO is the Device Driver Environment (DDE) CDIO which defines and supports the entry points and configuration services used by the SCSI Class and Transport Drivers to get configured and connected to each other and to their hardware. WSIO also provides various I/O tree and configuration database services and access to platform-dependent services such as PIO and DMA services. Currently, drivers also need a number of kernel services such as timer and memory allocation services. Refer to Chapter 1, “Overview of the Driver Environment,” Chapter 2, “HP-UX I/O Subsystem Features,” Chapter 3, “Multiprocessing,” Chapter 4, “Writing a Driver,” and Chapter 5, “Installing Your Driver,” for detailed discussions on WSIO and how a driver fits into the WSIO installation and configuration sequence. Refer to Chapter 1, “Overview of the Driver Environment,” Chapter 2, “HP-UX I/O Subsystem Features,” Chapter 4, “Writing a Driver,” and Chapter 5, “Installing Your Driver,” for detailed discussions on kernel services, WSIO services and SCSI services.
At the bottom of the SCSI subsystem are the transport drivers which typically directly control hardware (HBAs). These drivers directly access their hardware via **Programmed I/O** (PIO) read and write registers on the HBA card, and via the initiation of **Direct Memory Access** (DMA) to cause the HBA to transfer data to and from host memory. Refer to Chapter 12, “Writing a SCSI Interface Driver,” for additional details.

Within the SCSI subsystem, above the transport drivers, is a layer called the SCSI Services, which provides connection interface services between the SCSI Class Drivers and Transport Drivers. The SCSI Services layer defines the entry points and services that are used at the top side of the Transport Drivers to communicate up the stack (to receive requests or send completions) and at the bottom side of the Class Drivers (to send requests or receive completion notifications). In addition, the SCSI Services provides configuration and other services that are SCSI-specific, such as allocation of **qtags**, handling of SCSI queue depth, and probing of LUNs.

At the top of the SCSI subsystem are the class drivers, which control the individual SCSI LUNs for a given device class (disk, tape, autochanger, etc.). A generic class driver, called the SCSI pass-thru driver, can be used by SCSI-specific applications, commands, or upper level modules to build SCSI commands directly and send them to a device for vendor-specific or special control of a device, or for application control of a device class that isn’t supported by a specific class driver. Refer to Chapter 13, “Writing SCSI Device Drivers,” for additional details.

**SCSI Addressing Paradigm**

In the current HP-UX Mass Storage Stack, the SCSI Services and Class Drivers live in a SCSI-2 addressing paradigm (with up to 8 LUNs/target and up to 16 targets/bus). On transports that support larger addressing than this, the transport driver needs to create virtual buses, targets, and LUNs for use in this SCSI-2 paradigm, and translate these addresses to the larger physical target and LUN values for use on the transport. A description of how this can be done is provided.

**Virtual Bus, Target, and LUN Mapping**

Transport drivers can work around the SCSI-2 addressing restriction mentioned previously by creating virtual buses and targets which are attached to physical target I/O Tree nodes. In Fibre Channel, for example, the physical target I/O Tree node can be represented by a series of three I/O Tree nodes containing the 24-bit N-Port address of the actual FC target port (each I/O Tree node can represent up to a maximum of 8 bits of hw_path addressing). In support of this approach, the transport driver must register itself as an interface driver which claims each associated virtual bus (collection of 16 virtual targets) as a separate interface driver instance with its own isc, etc. This “virtual bus driver” must then claim the virtual bus nodes and register an appropriate probe routine such as parallel_scsi_probe, refer to the “Probing Functions” section to probe and create the necessary virtual targets and LUNs, and map them to their corresponding physical targets and LUNs internally in the “bus driver”.

For example, given a storage device attached to a Fibre Channel link containing 1024 LUNs (from 0–1023), the bus driver would create 8 virtual buses. These 8 virtual buses (which need to contain SCSI-2 style target and LUN addressing) would each contain 16 virtual targets, with 8 virtual LUNs per virtual target, for a total of 1024 virtual LUNs across the 8 virtual buses. The mapping can be simple; e.g., virtual bus 0, target 0, LUNs 0–7 maps to physical LUNs 0–7 in the storage device, virtual bus 0 target 0, LUN 1 maps to physical LUNs 8–15; virtual bus 1 maps to physical LUNS 128 to 255, and so on. If the storage device doesn’t have a contiguous LUN range the virtual LUNs corresponding to holes in the range (non-existent LUN addresses) are not created and virtual LUN range is similarly non-contiguous.
**WSIO I/O Tree Management Functions**

In HP-UX 11i v1 WSIO provides functions for transport drivers to use to, create and manage driver-specific I/O Tree nodes. These functions can be used by the transport driver to create child nodes of the HBA node such as the physical target nodes and virtual bus nodes. The virtual bus nodes need to get created by the transport driver as WSIO interface nodes so that they can be subsequently claimed by the transport driver’s virtual bus driver. To facilitate this, the virtual bus node would normally need to be in a subtree rooted at the HBA node, with the physical target nodes between the two as parents of the virtual bus node and children of the HBA node.

These child nodes can be created by calling the `wsio_create_interface` function, passing the HBA’s `isc` as the parent parameter, and the relative hw path of the corresponding physical target or virtual bus node. The type parameter can be `WSIO_TRANS` (transparent node) for the physical target nodes, and would need to be `WSIO_INTERFACE` (interface node) for the virtual bus nodes.

**Probing Functions**

To instantiate the SCSI-2 style buses, targets, and LUNs, the transport driver registers a probe routine with WSIO via a call to `wsio_register_addr_probe` from the driver’s install routine, refer to “Writing a driver_install() Routine” and “Writing Driver Probe Routines” in Chapter 4, “Writing a Driver.” This probe function does probing of LUNs at the SCSI Services level (i.e., using SCSI-2 style bus, target, and LUN values), which are then translated by the transport driver to its physical values, as described.

The SCSI Services routine `parallel_scsi_probe` provides a generic LUN probing routine for use in this paradigm, and therefore currently most of the various SCSI transport drivers, whether pSCSI or Fibre Channel, register this routine for this purpose.
12 Writing a SCSI Interface Driver

This chapter provides information on designing and developing SCSI transport drivers, also known as a **Host Bus Adapter** (HBA) drivers. SCSI transport drivers are WSIO interface drivers. See “Overview of Driver Types” in Chapter 2, “HP-UX I/O Subsystem Features,” for more information. SCSI transport drivers include drivers for parallel SCSI (pSCSI) and **Fibre Channel** (FC) transports.

This chapter details:

- “External Interfaces to a Transport Driver”
- “External Data Structures”
- “I/O Path”
- “Transport Driver Development”
- “Sample Driver”
- “Routines”

Code snippets from a sample HP-UX transport driver for Qlogic's ISP12160 SCSI Ultra160 interface card are provided as an example, during the explanation of transport driver development.
External Interfaces to a Transport Driver

The external interfaces available to a transport driver fall into three general categories:

- Driver exported functions that other modules can use to enter the driver (entry points).
- Service functions exported by other modules for use by drivers (SCSI, WSIO or Kernel Service).
- External data structures needed by the entry points or services.

Entry Points

The driver can only be entered via externally-visible driver routines, called “entry points”. There are five general categories of entry points for a SCSI Transport Driver:

- Configuration entry points.
- Request entry points.
- Interrupt entry points.
- Timeout entry points.
- WSIO Event entry points.

Configuration Entry Points

Include the driver's install routine, attach and init routines, and optionally a probe routine. See “Step 5: Writing Configuration Routines” in Chapter 4, “Writing a Driver.”

Request Entry Points

Defined by the SCSI Services in the scsi_ifsw structure. The driver initializes and registers its scsi_ifsw structure in the driver's init routine (by placing a pointer to it in isc->ifsw). These entry points include optional bus, target, LUN open and LUN close routines, and a mandatory start routine to handle the sending of SCSI Command Blocks (CDBs), and various optional ioctl or SCSI Task Management related entry points (abort, bdr, reset_bus, and ioctl). Optional request entry points can be set to NULL by the driver in its scsi_ifsw structure to indicate that it does not support the entry point. Refer to the “Request Entry Point Details” section for details.

Interrupt Entry Points

Registered from the driver's init routine via WSIO services. For details, refer to “Interrupt Handling” in Chapter 2, “HP-UX I/O Subsystem Features,” and the description of the qlisp_init_interrupts() function in the “Sample Driver” section, in this chapter.

Timeout Entry Points

The timeout callbacks that are registered in the driver when a timer is started. See “Timeout Mechanisms” in Chapter 2, “HP-UX I/O Subsystem Features,” and examples in the Qlisp sample driver code for details.

WSIO Event Entry Points

Registered from the driver's install routine via the WSIO service wsio_install_drv_event_handler(). These entry points currently include suspend and resume entry points for OLA/R with additional entry points to be supported in the future. Refer to Chapter 15, “On-Line Addition/Replacement,” for additional details.
Request Entry Point Details

As previously outlined, the driver's request entry points, which are defined in the scsi_ifsw structure, are composed of the following sets of entry points:

- Bus, target, LUN open and LUN close entry points (optional)
- Start entry point (mandatory)
- Abort, bdr, reset_bus, and ioctl entry points (optional)

Details regarding each of these entry points are provided.

Bus, Target, LUN Open/Close Entry Points

The scsi_ifsw structure contains the following definitions for these entry points:

```
int if_open(dev_t dev) /* if lun_open */
int if_tgt_open (dev_t dev)
int if_bus_open (dev_t dev)
void if_close (dev_t dev)
void if_tgt_close (dev_t dev)
```

Whenever a LUN is opened, the bus, target, and LUN entry points, if non-NULL, are called in the corresponding transport driver, passing the LUN's dev_t as a parameter, to allow the corresponding transport driver to do any bus, target, or LUN-specific processing that may be needed at open time. The transport driver can obtain a pointer to the corresponding scsi_bus, scsi_tgt, and m_scsi_lun structure via the SCSI Services m_scsi_bus, m_scsi_tgt, and m_scsi_lun respectively, refer to their definitions in the HP-UX 11i v1 Driver Development Reference Guide, which can be used to obtain corresponding driver-local structures and to do the processing that is needed for the specific open or close. For example, the driver can ignore non-first opens by checking the open_cnt field in the corresponding scsi_bus/tgt/lun structure.

These entry points are never called under interrupt context and are allowed to sleep.

The SCSI Services provides protection that blocks all other opens and closes to the same bus, target, or LUN, until the transport driver returns from the entry point.

Start Entry Point

The scsi_ifsw structure contains the following definition for this entry point:

```
if_start(struct isc_table_type *isc)
```

The transport driver must specify a non-NULL if_start function. The SCSI Services calls this entry point to inform the transport driver that a bp (pointer to a buf structure) has been queued on the driver's Select Queue for processing. This will be a bp for the SCSI command that needs to be performed.

The if_start entry point can be called on the Interrupt Control Stack (ICS) and is therefore not permitted to sleep.

The SCSI Services can call if_start at any time while the specified LUN is open, but has been optimized to only call it if the Select Queue is not empty. The transport driver must be prepared to process new requests on the Select Queue without receiving additional calls to if_start as long as the select queue has not gone empty while the driver is holding the scsi_bus lock. The Select Queue is synchronized between the SCSI Services and the transport driver via the scsi_bus structure's lock (scsi_bus->lock).
Abort, bdr, reset_bus, ioctl Entry Points

The scsi_ifsw structure contains the following definitions for the entry points:

```c
int if_abort(struct buf *bp)
int if_bdr(struct buf *bp)
int if_reset_bus (dev_t dev)
int if_ioctl(dev_t dev, int cmd, caddr_t data, int_flags)
```

The first three entry points if non-NULL, are called by the SCSI Services upon receipt of an SIOC_ABORT ioctl, an SIOC_RESET_DEV ioctl, or an SIOC_RESET_BUS ioctl. The if_ioctl entry point, if non-NULL is called upon receipt of ioctls that can't be handled at the class driver or SCSI Services level.

The if_reset_bus entry point is provided for use in Parallel SCSI (pSCSI) transport drivers to reset the corresponding pSCSI bus. When if_reset_bus returns, the SCSI bus should have been reset. Outstanding I/O's, whether queued in the transport driver, in the Host Bus Adapter (HBA) or in the device at the time of the reset must be cleaned up in hardware or software and returned to the SCSI Services, see scsi_cbfn.

The if_bdr entry point is intended for use in any of the SCSI transports to cause the transport driver to send a SCSI TARGET RESET to the specified target port.

The if_abort entry point is intended for use in any of the SCSI transports to cause the transport driver to send a SCSI ABORT TASK SET to the specified LUN.

The if_ioctl entry point is intended for processing of transport-specific ioctls. It is called for ioctls that can't be handled at the class driver of SCSI Services level to allow the transport driver the opportunity to process the ioctl. The if_ioctl shall return -1 if the transport driver doesn't support the requested ioctl.

Service Functions

The WSIO and kernel services are described in previous chapters. A description of the transport driver usage of SCSI Services follows.

Categories of SCSI Services functions used by transport drivers:

- I/O Completion Callback Functions
- Queuing Functions
- Dev_t Mapping Functions
- Synchronization Functions
- Miscellaneous Service Functions

In addition, new WSIO Services that can be used for virtual bus, target, LUN management are also discussed:

- I/O Tree Node Management Functions
I/O Completion Callback Functions

When a SCSI transport driver is finished with an I/O, it returns the I/O to the SCSI Services by calling one of the two completion callback functions, `scsi_cbfn` or `scsi_fast_cbfn`, which have the following interface definitions:

```c
void scsi_cbfn(struct buf *bp),
void scsi_fast_cbfn(struct buf *bp),
struct scb *scb,
struct scsi_lun *lp,
struct scsi_bus *busp;
```

The `scsi_cbfn` function can be used by the driver for simplicity in lower performance paths. The `scsi_fast_cbfn` function is intended for use in high performance paths.

The transport driver relinquishes all rights to access the `bp`, `scb` and `*scb->if_scb` once it calls the completion function. The `bp` may be freed or reused by the upper layers later for another I/O, and similarly for the `scb` and `*scb->if_scb`.

If the transport driver has attached a sense buffer to `scb->sense_data`, the `sense_data`, the buffer must remain valid until the completion function returns. The transport driver is forbidden from accessing it, until the completion function returns. The allocation and management of the sense buffer is the responsibility of the transport driver.

The completion functions can be called either in process or interrupt context.

---

**NOTE**

The completion functions must be called with any locks held since the SCSI services may call the transport driver's `if_start` entry point before returning.

---

Queuing Functions

The SCSI Services provides routines for managing queues (circularly-linked lists) of `bp`'s; `scsi_enqueue`, `scsi_dequeue`, and `scsi_dequeue_bp`, which have the following definitions:

```c
void scsi_enqueue(struct buf **qp, struct buf *bp, int where_in_queue);
struct buf *scsi_dequeue(struct buf **qp, int where_in_queue);
struct buf *scsi_dequeue_bp(struct buf **qp, struct buf *bp);
```

The `where_in_queue` parameter can have on the following values:

```c
#define TAIL  0
#define HEAD  1
```

The `qp` parameter points to a list header element whose forward and backward pointers in turn point to the HEAD and TAIL element of the list.

The `scsi_enqueue` enqueues the `bp` at the HEAD or TAIL of the circular list pointed to by `qp`.

The `scsi_dequeue` dequeues the `bp` at the HEAD or TAIL of the list pointed to by `qp` and returns the `bp`. NULL is returned when the queue is empty.
The `scsi_dequeue_bp`:

- Dequeues `bp` from wherever it may be in the queue, pointed to by `qp`.
- Returns `bp` when found on the queue.
- Returns NULL when not found on the queue.

**NOTE** Queuing functions must be called with the `scsi_bus` lock held.

**dev_t Mapping Functions**

The SCSI Services provides routines for mapping a `dev_t` into its corresponding bus, target, or LUN instance structure, or into its `isc` structure:

```c
struct scsi_bus *m_scsi_bus(dev_t dev);
struct scsi_tgt *m_scsi_tgt(dev_t dev);
struct scsi_lun *m_scsi_lun(dev_t dev);
struct isc_table_type *m_scsi_isc(dev_t dev);
```

The `m_scsi_bus`, `m_scsi_tgt`, `m_scsi_lun`, and `m_scsi_isc` return a pointer to the `scsi_bus`, `scsi_tgt`, `scsi_lun`, and `isc` structure, that corresponds to `dev`. They return NULL if `dev` is invalid or references a bus, target, or LUN which is not currently open. The `m_scsi_isc` will return a pointer to the `isc` even if the bus isn't open, as long as `dev` refers to a device on a configured bus.

Spinlocks are the most heavily used synchronization mechanism in the HP-UX kernel, and are the basic locking primitive used by the kernel for short-term locks.

**Synchronization Functions**

The SCSI Services provides functions for transport drivers to use to synchronize their access to SCSI Services data structures:

```c
void scsi_tgt_lock(struct scsi_tgt *tp);
void scsi_tgt_unlock(struct scsi_tgt *tp);
void scsi_bus_lock(struct scsi_bus *busp);
void scsi_bus_unlock(struct scsi_bus *busp);
```

The `scsi_tgt_lock` obtains the `scsi_tgt` structure's spinlock. The `scsi_bus_lock` obtains the `scsi_bus` structure's spinlock. These locks must be held while accessing the `scsi_tgt` or `scsi_bus` structure. The bus lock must also be held while accessing the “select queue”.

**Miscellaneous Service Functions**

The SCSI Services provides the pSCSI-specific function for use in pSCSI transport drivers:

```c
int scsi_frequency(int sdtr_period);
```

The `scsi_frequency` translates the transfer period factor of an **Synchronous Data Transfer Request** (SDTR) or PPR message into its corresponding Hz frequency. For example, a transfer period factor of 8, which is defined in the SCSI specification (SPI-4) to be Ultra320 speed, returns the value 160,000,000 (160 MHz).

SCSI Services also provides a probe function for use by transport drivers called `parallel_scsi_probe`. The transport driver does not call this routine itself, but can register it as the driver's probe routine with WSIO via a call to `wsio_register_addr_probe` from the driver's install routine, see “Writing a `driver_install()` Routine” and “Writing Driver Probe Routines” in Chapter 4, “Writing a Driver.”
Due to the SCSI-2 paradigm that exists at the SCSI Services level and above in the current HP-UX mass storage stack, the `parallel_scsi_probe` routine is commonly registered as the LUN probing routine by most SCSI transport drivers, (pSCSI or Fibre Channel). See Chapter 11, “Mass Storage Stack Architecture.”

### I/O Tree Node Management Functions

The WSIO Services provides functions for a transport driver to create and manage driver-specific I/O Tree nodes. The only expected use for this in HP-UX 11i v1 is for virtual bus, target and LUN management. See Chapter 11, “Mass Storage Stack Architecture,” for additional information on virtual bus, target, and LUN management. See the WSIO Reference Pages in the HP-UX 11i v1 Driver Development Reference Guide, for details on the WSIO I/O Tree Node interfaces, such as `wsio_create_interface` (to register an I/O interface and associated hw path, and return a corresponding isc), `wsio_create_attribute` (to create and attach an attribute to an I/O interface), and `wsio_get_relationship` (to get the parents of children of an interface).

To address more than 8 LUNs (for example a SCSI-3 target), a driver has to create virtual SCSI-2 LUNs (under virtual SCSI-2 targets which in turn is under virtual SCSI-2 buses) to represent the large LUN space behind the target. Once the virtual entities are created, communication between the SCSI-2 stack and the SCSI-3 target happens through a virtual SCSI bus layer.

For this scheme to work, a HBA driver has to have an additional virtual bus driver, which intercepts the open or I/O requests from the SCSI-2 stack and converts it to transport specific open or SCSI-3 I/O. The virtual bus driver also has to be scannable (DRV_SCAN set in `drv_info_t` of the driver) and has to have a target probe interface (registered using `wsio_register_dev_probe()`). This probe interface will be responsible for creating the virtual SCSI buses (also referred to as virtual bus nodes) which can be later scanned using HPUX SCSI services. The virtual bus driver for the HBA driver will have to claim the virtual buses created and HPUX SCSI services (`parallel_scsi_probe()`) can be used to scan those virtual SCSI buses.

An example of addressing 1024 LUNs behind a FC SCSI-3 target using the WSIO Interface management functions is given.

In the Fiber Channel world, a target is addressed using a 24 bit address (`N_PortID`). The addressing method which is described in this section is based on the `N_PortID` of the target; in other words, if the `N_PortID` changes for the target, the device file for the LUN also has to change.

As long as the hardware path to the LUN remains the same, the device file corresponding to it will also remain the same. Thus, the hardware path is used to store information. In this scheme, `N_PortID` and `FCP_LUN` is embedded into a hardware path. A LUN hardware path looks like this:

```
<HBA-hw-path>/<domain>.<area>.<port>.<vbus>.<vtgt>.<vlun>
```

where `vbus`, `vtgt` and `vlun` are taken from the SCC first level LUN ID.

```
->| |<- 2 bits addressing
-------------
| AD |<------- 14 bit FCP_LUN_0 ----->|
-------------
```

```
->| |<- 2 bits addressing
| 9<->7 6<->3 2<->0 Bit position
-------------
| AD | vbus | vtgt | vlun |
-------------
```

To achieve the previous hardware path for a LUN, HBA driver has to do this.
For each FC target, 3 tiers of nodes are created representing:

- Domain node (upper 8-bits) under the HBA port
- Area node (middle 8-bits) under the domain
- Port node (lower 8-bits) under the area.

Under the port node one or more virtual SCSI bus nodes are created depending upon the number of LUNs behind the target.

Nodes can be created using `wsio_create_interface()`.

Once the virtual bus nodes are created and after `vbus` driver claims it, `parallel_scsi_probe()` takes over the scanning and creates targets and LUNs beneath it.

The HBA driver probe interface is responsible for creating the virtual SCSI buses. Virtual bus driver is responsible for claiming those virtual SCSI bus nodes, intercepting I/Os from SCSI services and converting it to SCSI-3 packets.

In the following code snippets, the interface driver is represented as "`scsi_xport`" and the virtual SCSI bus driver as "`scsi_xport_vbus`".

Once the virtual SCSI buses are created, the `scsi_xport_vbus` module would claim those nodes. The `scsi_xport_vbus` driver registers an address probe routine `parallel_scsi_probe()` with WSIO, and the rest of the scanning/discovery (scanning of the virtual SCSI bus) of SCSI targets/LUNs (in this case, virtual targets and virtual LUNS) behind the virtual bus is done by `parallel_scsi_probe()`.

The `scsi_xport` transport driver registers a device probe routine that scans for all the SCSI-3 targets and create virtual SCSI buses (one or more) for all the targets to address the large number of LUNs behind them.

```c
scsi_xport_install()
{
    
    if(wsio_register_dev_probe(DRV_NAME, scsi_xport_probe,
        SCSI_XPORT_NAME) != WSIO_OK) {
        return(0);
    }
}
```

The following, is a code snippet of the `scsi_xport_probe()` routine that creates virtual buses. The `scsi_xport_vbus` driver manages an attach chain used to claim the virtual bus nodes created by the `scsi_xport_probe` routine.

`---------
 Probe logic
 ---------`

1. Find the number of devices attached to HBA (e.g.: from name server query in fabric topologies)
2. For each target
   - Build and send SCSI INQUIRY command.
   - For disk array type of devices,
     . Build and send REPORT_LUNS command
. Compute the number of virtual SCSI buses that needs to be created from the response
. Create domain, area and port nodes, if not already created.
. Create virtual buses as needed

------------

static int (*scsi_xport_attach_chain)() = scsi_xport_vbus_attach;

/*
 * Compute the number of virtual SCSI buses that need to be created (by probing the targets/LUNs which can be seen from this HBA port) and return them for every call into this routine; one PROBE_FIRST and multiple PROBE_NEXT calls
 *
 * One possible way of addressing 1024 luns for the SCSI first level LUN (first two bytes of the 64 bit FCP_LUN) is this:
 *
 * ->|    |<- 2 bits addressing
 * ------------------------
 * | AD |<-------- 14 bit FCP_LUN_0 ------>
 * ------------------------
 *
 * ->|    |<- 2 bits addressing
 *    9<->7  6<->3  2<->0 Bit position
 * ------------------------
 * | AD |          | VBUS | VTGT | VLUN |
 * ------------------------
 *
 * where VBUS is the hardware path address for the virtual bus node
 * VTGT is the hardware path address for the virtual target node
 * VLUN is the hardware path address for the virtual lun node
 *
 * VBUS and VLUN nodes are created by parallel_scsi_probe() (the scsi_xport_vbus_if_open() need to map the tgt_id (in struct scsi_tgt) and lun_id (in struct scsi_lun) to bits 3-6 and 0-2 respectively of FCP_LUN_0)
 *
 * The node passed to scsi_xport_probe is the node of the HBA. Initially (PROBE_FIRST) hw_path->last_index is set to point to HBA node element
* (hw_path->addr[hw_path->last_index] == HBA's address) and hw_path->
* first_index is set to hw_path->last_index+1 by the caller.
*/

scsi_xport_probe(
    void                   *node,
    drv_info_t             *drv_info,
    void                   *probe_id,
    hw_path_t              *hw_path,
    struct isc_table_type  *isc,
    int                    probe_type,
    char                   *name,
    char                   *desc
)
{

    struct scb              *scb;
    struct buf              *bp;
    union inquiry_data      *inq_data;
    wsio_ret_code_t         w_ret;
    struct isc_table_type   *isc_ret;

    /*
    * The isc gotten here is the same isc claimed in scsi_xport_attach().
    * Use isc->if_isc to pass any information from scsi_xport_attach()
    * to scsi_xport_probe()
    */

    :  

    /* Do the actual scanning of FC targets here */

    /* Send SCSI commands INQUIRY to identify the type of device and
    * REPORT_LUNS to find out the number of LUNs behind the target
    */

    /* sample code for sending SCSI command INQUIRY follows */
    bp  = kmalloc(sizeof(struct buf), M_IOSYS, M_WAITOK);
    scb = kmalloc(sizeof(struct scb), M_IOSYS, M_WAITOK);
    inq_data = kmalloc(sizeof(union inquiry_data), M_IOSYS, M_WAITOK);
    /* Allocate and intialize if_scb, if needed */
 rsp_len = sizeof(union inquiry_data);
 scb->max_msecs = 10*1000; /* 10 seconds timeout */
 scb->cdb[0] = CMDinquiry;
 scb->cdb[4] = rsp_len;
 scb->cdb_len = 6;

 bp->b_scb = scb;
 bp->b_flags = B_SCSI_CMD | B_READ;
 bp->b_bcount = bp->b_resid = rsp_len;
 bp->b_un.b_addr = (void *)inq_data;

 /* Call interface drivers IO start routine here to send this IO */

 /*
 * Once all the targets and LUNs behind the targets are scanned,
 * build an array/tree of data structures containing the N_PortID
 * of all FC targets and for each N_PortID, the number of virtual
 * buses (with its hardware path address) under it.
 * 
 * Remember, the objective of this probe routine is to create
 * DOMAIN, AREA, PORT and VBUS nodes only. Once the VBUS node is
 * created, parallel_scsi_probe() takes over the node, once
 * xx_vbus module claims it, and do the rest of the scanning.
 * 
 * The order in which the nodes created are as follows:
 * 
 * parallel_scsi_probe kicks
 * .-- | off as soon as ext_bus
 * | | nodes are created, which
 * V | is responsible for the
 * DOMAIN-1, AREA-1, PORT-1, EXT_BUS-1 | creation of "target" and
 *     :                        \ "lun" nodes
 *             EXT_BUS-N
 *              :
 * PORT-N, EXT_BUS-1
 *              :
 * EXT_BUS-N
 *              :
 * AREA-N, PORT-1, EXT_BUS-1
 *              :
Writing a SCSI Interface Driver
External Interfaces to a Transport Driver

*                          EXT_BUS-N
*                          PORT-N, EXT_BUS-1
*                           :
*                          EXT_BUS-N
*                           :
*                         DOMAIN-N, AREA-1, PORT-1, EXT_BUS-1
*                           :
*                          EXT_BUS-N
*                           :
*                          PORT-N, EXT_BUS-1
*                           :
*                          EXT_BUS-N
*                           :
*                        AREA-N, PORT-1, EXT_BUS-1
*                           :
*                          EXT_BUS-N
*                           :
*                          PORT-N, EXT_BUS-1
*                           :
*                          EXT_BUS-N
*

/*
 * Assuming a routine scsi_xport_get_next() returns the elements
 * in the above mentioned order, the IO tree nodes corresponding
 * to the DOMAIN, AREA, PORT and VBUS (or ext_bus) can be
 * created like this.
 *
 * #define DOMAIN  1
 * #define AREA    2
 * #define PORT    3
 * #define VBUS    4
 *
 * Also assume scan_elem is of the following structure type
 *
 * struct scan_element {
 *     int     nport_id;   /* 24 bit nport id */
 *     int     type;       /* DOMAIN, AREA, PORT or VBUS */
 *     uint8_t addr;       /* hardware path address for this element */
 * } scan_elem;
 */
while(scsi_xport_get_next(&scan_elem)) != NULL) {
    switch(scan_elem.type) {
    case DOMAIN:
        hw_path->last_index = hw_path->first_index;
        hw_path->addr[hw_path->last_index] = scan_elem.addr;

        w_ret = wsio_create_interface(isc, hw_path, WSIO_TRANS,
                                       NULL, NULL, NULL, "", &isc_ret);

        if (w_ret != WSIO_OK) {
            return(PROBE_UNSUCCESSFUL);
        }

        return(PROBE_SUCCESS);
        break;

    case AREA:
        hw_path->last_index = hw_path->first_index + 1;
        hw_path->addr[hw_path->last_index] = scan_elem.addr;
        /* create transparent node as in DOMAIN */

        return(PROBE_SUCCESS);
        break;

    case PORT:
        hw_path->last_index = hw_path->first_index + 2;
        hw_path->addr[hw_path->last_index] = scan_elem.addr;
        /* create transparent node as in DOMAIN */

        return(PROBE_SUCCESS);
        break;

    case VBUS:
        hw_path->last_index = hw_path->first_index + 3;
        hw_path->addr[hw_path->last_index] = scan_elem.addr;
        /*
         * set
         * probe_id as : a 16(max) byte identifier
         * (on subsequent ioscans, if this
* differs, ioscan output shows the
* element as DIFF_HW).
* name : a name for the io tree node
* desc : description for the node (this appears under
*        (Description in ioscan output)
*/

w_ret = wsio_create_interface(isc, hw_path,
   WSIO_INTERFACE, probe_id, name, desc, "", &isc_ret);
w_ret = wsio_sizeof_attribute(isc_ret, "__wsio_card_parms",
     &size);

if (w_ret != WSIO_OK) {
    /* create card_parms property for the node */

    bus_info = kmalloc(sizeof(xx_bus_info_t),
                       FCD_WAITOK);

    bzero((caddr_t)bus_info, sizeof(fcd_pb_bus_info_t));
    bzero((caddr_t)&card_parms,
         sizeof(struct wsio_card_parms));

    card_parms.slot_num     = -1;
    card_parms.if_flags     = 0;
    card_parms.bus_info     = (caddr_t)bus_info;
    card_parms.bus_type     = PCI_BUS;
    /* slot_id is passed to scsi_xport_vbus_attach as id */
    card_parms.slot_id      = SCSI_XPORT_VBUS;
    card_parms.attach       = &scsi_xport_attach_chain;

    /*
    * Initialize scsi_xport_bus_info_t here (to pass
    * information to scsi_xport_vbus module
    */
    w_ret = wsio_create_attribute(isc_ret,
        "__wsio_card_parms",
        (uintptr_t)&card_parms,
        sizeof(struct wsio_card_parms),
        WSIO_COPYDATA);

    if (w_ret != WSIO_OK) {
return(PROBE_UNSUCCESSFUL);

}

return(PROBE_SUCCESS);
break;
}
}
}

The following code snippets represent the `scsi_xport_vbus()` module that manages the virtual bus nodes.

/* This structure contains pointers to all driver entry points */
static drv_ops_t xx_vbus_ops = /* no interface via [bc]devsw */
{
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    0,
};

/* This structure is used by GIO & WSIO to display properties. */
/* The driver here is scsi_xport_vbus driver with class as ext_bus */
/* and the device flags are that the scsi_xport_vbus driver supports */
/* scanning, defered scanning and is MP SAFE. */

static drv_info_t scsi_xport_vbus_info =
{
    "scsi_xport_vbus",
    "ext_bus", /* the SCSI-2 bus class */
    DRV_SAVE_CONF | DRV_SCAN | DRV_DEFER_SCAN | DRV_MP_SAFE,
The following routine registers the `scsi_xport_vbus()` driver with WSIO. It also registers an address probe routine that enumerates the SCSI LUNs behind a target.
scsi_xport_vbus_install(void)
{
    int ret;

    /* Call wsio_install driver to register with wsio */
    install_ret = wsio_install_driver(&scsi_xport_vbus_wsio_info);
    if (install_ret != WSIO_OK) {
        return 0;
    }

    /* Register the parallel_scsi_probe as the probe routine. */
    wsio_register_addr_probe(parallel_scsi_probe, 'scsi_xport_vbus');
}

The scsi_xport_vbus() module's attach routine is used to claim the virtual bus nodes created by the scsi_xport driver.

int
scsi_xport_vbus_attach(int id, struct isc_table_type *isc)
{
    /*
     * id is the same as slot_id programmed in struct wsio_card_parms when
     * creating this vbus node. This can be used to identify the type of
     * the virtual scsi_bus node which we have created before (this is
     * needed only if we create multiple types of virtual scsi_buses
     */
    /* do "id" validation here if needed */
    :
    /* isc->bus_info is the same pointer, which was programmed as
     * wsio_card_parms's bus_info pointer, when this node was created
     */
    scsi_xport_bus_info = (scsi_xport_bus_info_t *) isc->bus_info;

    /* Initialize out scsi_xport_vbus_init routine with the isc's gfsw */
    *(isc->gfsw) = scsi_xport_vbus_gfsw;

    /* bind this driver to this node by calling isc_claim */
    isc_claim(isc, &scsi_xport_vbus_wsio_info);

    /* Set the bus_max_width to 16 for SCSI probe */
    isc->bus_max_width = 16;
/ for scsi_frequency() calc */
isc->bus_min_sdtr_period = 2;

/* Call the next driver in the chain, if there are multiple drivers
* which can possibly claim this node. If there is only one driver
* which can claim this node, call wsio_last_attach() directly.
*/
return (wsio_last_attach(id, isc));
}

The following routine shows the initialization steps for the enumerated virtual bus nodes.

static int
scsi_xport_vbus_init(struct isc_table_type *isc)
{

/*
 * So long as the hardware path to this virtual SCSI bus remains
 * the same, the instance number also would remain the same (actually,
 * the bus part (cXXX) of the instance number) guaranteeing the same
 * device file for the lun.
 */
int bus_id = wsio_isc_to_instance(isc, NULL);

if (bus_id > SCSI_MAX_BUS_ID) {
    printf("WARNING: Maximum possible scsi_bus limit has reached\n",
           "No more LUNs will be seen from scsi_xport driver\n");
    return WSIO_ERROR;
}
scsi_isc[bus_id] = isc;

scsi_xport_ifsw = kmalloc(sizeof(struct scsi_ifsw), M_IOSYS, M_WAITOK);
/* initialize our ifsw */
bzero((caddr_t) scsi_xport_ifsw, sizeof(struct scsi_ifsw));

/* set IF_B2_LIST too if capable of disk sort merge buffer request. */
scsi_xport_ifsw->if_flags = IF_BUS_TAGS;

scsi_xport_ifsw->if_max_tag = 254;

scsi_xport_ifsw->if_scb_size = sizeof(scsi_xport_io_ccb_t);
scsi_xport_ifsw->if_lun_size = sizeof(scsi_xport_scsi_lun_t);
Writing a SCSI Interface Driver

External Interfaces to a Transport Driver

```c
scsi_xport_ifsw->if_open = scsi_xport_vbus_if_open;
scsi_xport_ifsw->if_close = scsi_xport_vbus_if_close;
scsi_xport_ifsw->if_start = scsi_xport_vbus_if_start;
scsi_xport_ifsw->if_max_io_size = 1024 * 1024; /* 1MB is the max */

/* Set our scsi_xport_ifsw to isc's ifsw */
isc->ifsw = (caddr_t)scsi_xport_ifsw;

return(WSIO_OK);
}
```

The following routines show the `if_open`, `if_start` and `if_close` entry points in the `if_sw` table.

```c
scsi_xport_vbus_if_open(dev_t dev)
{
    /*
     * Call interface driver's open routine to establish nexus, if
     * it is not already established
     */

    /*
     * Make use of isc->bus_info to pass any information (like per
     * instance interface data structure etc) from probe
     * isc can be retrieved as follows:
     *
     * struct scsi_bus     *busp = m_scsi_bus(dev);
     * struct isc_table_type *isc  = busp->isc;
     */

    /*
     * Additionaly, can convert the SCSI-2 tgt_id and SCSI-3 lun_id to
     * FCP_LUN (the reverse mapping) and could store the information in
     * if_lun and later retrieved for I/Os
     */
}
```

```c
scsi_xport_vbus_if_start(struct isc_table_type * isc)
{
    struct scsi_bus *busp = (struct scsi_bus *) isc->if_drv_data;

    /* Dequeue request packets (bp) from busp->select_q */
```
/* Find the HBA port instance from isc->bus_info */

/* retrieve the FCP_LUN from if_lun */

/* Issue the IO request to the hardware */

}

scsi_xport_vbus_if_close(dev_t dev)
{
    /* Close the nexus on last close to the device */
}

External Data Structures

In addition to the functions, SCSI subsystem also provides data structures to pass data across different layers of the SCSI stack. Some WSIO data structures are also used to pass data across the SCSI subsystem. Of the specified data structures and fields, some are owned by the transport driver, i.e., maintained only by the transport driver (after being initialized to zero by services). These exist because some part of the SCSI subsystem other than the transport driver needs access to the information, and source of this information is the transport driver. It could be services or a device driver that needs access to the information, but that is irrelevant to the transport driver.

Other specified data structures and fields are owned by some other part of the SCSI subsystem and are available for use by the transport driver. Those remaining are owned by neither the transport driver nor some other part of the SCSI subsystem, but may be accessed and modified by either.

The *bp, scb, *scb->if_scb and the data buffer for an I/O are available for use by the transport driver only while the I/O is active.

In this section data structures are present in C syntax and only fields that are relevant to the current discussion are shown.

SCSI Control Block

The buf structure is not large enough to hold all state information associated with a SCSI I/O attempt. An scb is attached to a buf by SCSI services to hold the temporary state information until the I/O is completed. The *bp/scb association does persist for retries. The struct buf is described later in the section.

Fields that are of interest to a transport driver writer are:

```c
struct scb {
    void *if_scb;
    struct scsi_lun *lp;
    ubit32 flags;
    ubit32 max_msecs;
    ubit8 cdb[SCSI_MAX_CDB_LEN];
    ubit8 cdb_len;
    ubit32 io_id;
    ubit8 tag;
    ubit32 cdb_status;
    ubit32 data_resid;
    ubit32 sense_status;
    ubit8 sense_bytes;
    ubit8 *sense_data;
} *scb;
```

The transport driver is not allowed to set the following fields:

- lp
- flags
- max_msecs
Writing a SCSI Interface Driver

External Data Structures

- cdb
- cdb_len
- io_id
- tag

**NOTE** If a field is already described earlier, it will only be mentioned here. For more details on such a field, refer to earlier sections.

**scb->if_scb**

Is a pointer to ifsw->if_scb_size bytes allocated by SCSI services and reserved for use by the transport driver. The pointer is initialized at scb creation time by services and the data area is bzero'ed by services for each I/O attempt prior to putting the I/O on the select queue. It is not touched by services at any other time. The if_scb area is later freed by the SCSI services along with the scb.

**scb->lp**

Is a pointer to the scsi_lun structure in the open device tree with which this scb is associated. If the scb belongs to a per-LUN pool of scb's as opposed to a per-bus pool, then scb->lp is initialized at scb creation time by SCSI services and never changed. Otherwise, the scb belongs to a per-bus pool, and scp->lp is only valid while the scb is associated with a bp. Then scb->lp points to the scsi_lun structure associated with bp->b_dev.

**scb->flags**

The transport driver may check these bits in the flag for proper functionality. Bits in scb->flags that are relevant to a transport driver are:

- **SCB_NO_DISC**
  - This bit indicates that the disconnect privilege should not be granted in the identify message.

- **SCB_SDTR**
  - If this bit is set and SCB_WDTR is not set, the transport driver should initiate SDTR negotiation immediately following the Selection, Identify or tag message, whichever comes last, and before sending the **Command Description Block** (CDB) for the I/O.

- **SCB_WDTR**
  - This bit directs the transport driver that a wide negotiation should be initiated immediately following the Selection, Identify or tag message, whichever comes last, and before sending any CDB for the I/O. If (tp->state & T_ENABLE_SDTR) or (scb->flags & SCB_SDTR) is also set, the transport driver should initiate SDTR negotiation immediately following the **Wide Data Transport Request** (WDTR) negotiation. The wide negotiation should always precede the synchronous negotiation, since a wide negotiation resets the link to asynchronous.

- **SCB_4BYTE**
  - This bit informs the transport driver that the target will never change phase while in data phase on other than a 4-byte boundary at the beginning of the data transfer without subsequently restoring the data pointer (implicitly or explicitly) to a previously aligned value and re-transferring data up to and beyond the point of
disconnection to an aligned boundary. The phase change at the end of the I/O needs to be considered only if the amount of data transferred may be less than that requested in bp->b_bcount. Note that SCB_4BYTE does not imply that bp->b_count is a multiple of four or that bp->b_un.b_addr is 4-byte aligned. Note also, that the phase change out of data phase if all bp->b_bcount bytes have been transferred is not subject to the alignment restructuring.

SCB_2BYTE
This bit is the same as SCB_4BYTE except that phase changes are only restricted to even boundaries.

SCB_ORDERED_TAG
Denotes that ordered tags are intended to be used for this device.

scb->max_msecs
Minimum number of milliseconds the transport driver is to allow for this I/O from the time of Selection until Command Complete in parallel SCSI or as close to that as possible within a given HBA architecture. If scb->max_msecs milliseconds elapses and the I/O has not completed, the transport driver is encouraged to abort the I/O with Abort or Abort Tag as appropriate. The transport driver can run a timer routine periodically to watch for the I/O’s that are timed out. A value of zero indicates the transport driver should never abort this I/O based solely on the amount of time since Selection.

scb->cdb
Holds the SCSI command bytes for this I/O.

scb->cdb_len
The number of bytes in the cdb. This can be a maximum of SCSI_MAX_CDB_LEN.

scb->io_id
Is a unique identifier for a SCSI I/O. It is initialized when the scb is associated with a bp and is unique across all SCSI buses.

scb->tag
The tag value allocated for this I/O by the SCSI subsystem in accordance with the transport driver’s direction via ifsw->if_max_tag. It is recommended that the transport driver use this value as the tag value for the I/O if the I/O will be tagged, but not required. Currently there can only be 256 tags per bus. The tag value may not remain the same for retried I/O’s.

scb->cdb_status
Indicates the status of the I/O command. If the I/O attempt completes with no phase sequencing errors and without being aborted or timed out, the transport driver sets scb->cdb_status to S_GOOD. If the selection phase times out, the transport driver sets cdb_status to SCTL_SELECT_TIMEOUT. If the I/O is not even attempted because of bogus data in the bp or scb, the transport driver sets cdb_status to SCTL_INVALID_REQUEST. If the I/O is not attempted or does not complete for any other reason, cdb_status is set to SCTL_INCOMPLETE. If there is a Contingent Allegiance condition, the cdb_status is set to S_CHECK_CONDITION to request an auto-sense request. scb->cdb_status must be set by the transport driver prior to returning the bp via scsi_cbfm.

Refer to scsi.h for the valid values of cdb_status.
If the I/O attempt completes with no phase sequencing errors and without being aborted or timed out, the transport driver sets `scb->data_resid` so that `bp->b_count - scb->data_resid` is the offset from `bp->b_un.b_addr` of the first byte not transferred by the target, i.e., number of bytes transferred = `bp->b_count - scb->data_resid`. Even if the I/O attempt is failed for some reason, set the `scb->data_resid` to indicate the number of bytes that are not yet transferred. Setting this field will have no adverse affect. `scb->data_resid` must be set by the transport driver prior to returning the `bp` via `scsi_cbfn`.

Represents the status of the automatic request sense that is performed if `scb->cdb_status` is `S_CHECK_CONDITION`. If the Request Sense completes with no phase sequencing errors and without being aborted or timed out, the transport driver sets `scb->sense_status`. Otherwise, `scb->sense_status` is undefined and will not be referenced by the SCSI subsystem on callback. The possible values for `scb->sense_status` are the same as those for `scb->cdb_status` except `SCTL_INVALID_REQUEST` cannot be used. `scb->sense_status` represents the result of the automatic Request Sense in the same way that `scb->cdb_status` represents the result of attempting `scb->cdb`. It must be set by the transport driver before returning the `bp` via `scsi_cbfn`. If there is any sense data, the `sense_status` has to be set to `S_GOOD`.

Number of bytes of data received in response to the automatic request sense, if one was performed. It is valid only if `sense_status` is valid and is neither `SCTL_SELECT_TIMEOUT` nor `SCTL_INCOMPLETE`. `scb->sense_bytes` is the offset from `scb->sense_data` of the first byte of sense data not transferred by the target. It must be set by the transport driver prior to returning the `bp` via `scsi_cbfn`.

If `scb->cdb_status` is Check Condition and the resulting Request Sense completes with no phase sequencing errors and without being aborted or timed out, and if `scb->sense_status` is not zero, the transport driver sets `scb->sense_data`. Otherwise, `scb->sense_data` is undefined and will not be referenced by the SCSI subsystem on callback. The transport driver sets `scb->sense_data` to point to a KERNELSPACE buffer containing the sense data; its size must be at least `scb->sense_bytes`. It must be set prior to returning the `bp` via `scsi_cbfn` and the transport driver must not modify the buffer for the duration of `scsi_cbfn`. When `scsi_cbfn` returns, and not until, the transport driver can reuse the buffer.
ISC

Each instance of an interface card has an Interface Select Code (ISC) entry that the system maintains in an internal table. Each ISC entry, defined as an isc_table_type structure, is used by WSIO to maintain transport driver information. Refer to the Chapter 2, “HP-UX I/O Subsystem Features,” for a detailed discussion on this structure. It is not part of the SCSI subsystem.

Some of the fields of the isc_table_type structure are reserved for use by the transport driver as specified by the SCSI subsystem. They are described in this section. Others are reserved for use by the transport driver at its discretion. These reserved fields are - ppoll_flag, ppoll_mask and ppoll_sense. However, these fields are not typically used by the transport driver. A number of these fields have been renamed using #defines to more accurately reflect their meaning within the context of the SCSI subsystem. Fields specific to a SCSI transport driver are:

```c
struct isc_table_type {  
    char my_address;  
    struct gfsw *gfsw;  
    caddr_t *ifsw; 
    
    unsigned char int_enabled;  
    unsigned char spoll_byte;  
    unsigned char tfr_control;  
    struct buf *ppoll_f;  
    struct buf *ppoll_l;  
    
    int lock_count;  
    struct buf *event_f;  
    struct buf *event_l;  
    struct buf *status_f;  
    struct buf *status_l;  
    
    char ppoll_flag;  
    unsigned char ppoll_mask;  
    unsigned char ppoll_sense;  
    struct buf *owner;  
    unsigned int state;  
    int *card_ptr;  
    
    unsigned char my_isc;  
    char bus_type;  
    caddr_t if_reg_ptr;  
    caddr_t if_drv_data;  
    void *if_isc;  
    
    int if_id;  
} *isc;  
```

#define bus_max_width int_enabled
#define bus_min_sdtr_period spoll_byte
#define bus_max_reqack_offset tfr_control
#define tgt_wdtr_done ppoll_f
#define tgt_wdtr_width ppoll_l
#define tgt_sdtr_done lock_count
#define tgt_sdtr_period event_f
NOTE The transport driver is *not allowed* to set `if_drv_data`.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>isc-&gt;myaddress</code></td>
<td>SCSI bus address of the initiator. It is a binary value from zero to fifteen, it is not a power of two, representing the data bit used by the initiator for selection and re-selection. <code>isc-&gt;my_address</code> is initialized by the transport driver's attach routine.</td>
</tr>
<tr>
<td><code>isc-&gt;gfw</code></td>
<td>Pointer to the transport driver's <code>gfsw</code> structure. The SCSI subsystem does not require that the transport driver provide a <code>gfsw</code> structure.</td>
</tr>
<tr>
<td><code>isc-&gt;ifsw</code></td>
<td>Pointer to the transport driver's <code>scsi_ifsw</code> structure. It is initialized by the transport driver's attach routine.</td>
</tr>
<tr>
<td><code>isc-&gt;bus_max_width</code></td>
<td>Width of the SCSI data bus. Currently, reasonable values are 8 and 16. It is initialized by the transport driver during the driver's attach routine. This field is later used by the SCSI services while probing for SCSI devices. Not setting this field results in not seeing devices on this SCSI bus.</td>
</tr>
<tr>
<td><code>isc-&gt;bus_min_sdtr_period</code></td>
<td>Minimum synchronous data transfer period supported by the hardware. This field is expressed in units of 4 ns. It is initialized during the driver's attach routine.</td>
</tr>
<tr>
<td><code>isc-&gt;bus_max_reqsck_offset</code></td>
<td>Maximum synchronous data transfer REQ/ACK offset supported by the hardware. It is initialized during the driver’s attach routine.</td>
</tr>
<tr>
<td><code>isc-&gt;tgt_wdtr_done</code></td>
<td>Indicates whether or not a WDTR negotiation has occurred since the most recently detected event which resets the data transfer width to eight bits, i.e., bus reset or BDR. There is one bit for each target, the least significant bit is for target zero, the next least significant bit is for target one, and so on. If the bit is set, a negotiation has occurred, otherwise, no negotiation has occurred.</td>
</tr>
<tr>
<td><code>isc-&gt;tgt_wdtr_width</code></td>
<td>Indicates the current width of data transfers. There is one bit for each target; the least significant bit is for target zero, the next least significant bit is for target one, and so on. If the bit is set, sixteen-bit data transfers are in effect, otherwise eight-bit transfers are being used.</td>
</tr>
<tr>
<td><code>isc-&gt;tgt_sdtr_done</code></td>
<td>Indicates whether or not an SDTR negotiation has occurred since the most recently detected event which resets the data transfer parameters to asynchronous; that is, bus reset, BDR or WDTR. There is one bit for each target, the least significant bit is for target zero, the next least significant bit is for target one, and so on. If the bit is set, a negotiation has occurred, otherwise, no negotiation has occurred.</td>
</tr>
<tr>
<td><code>isc-&gt;tgt_sdtr_period</code></td>
<td>Represents the location of an array of bytes indicating the current synchronous data transfer period as represented in an SDTR message. The address of <code>isc-&gt;tgt_sdtr_period</code> is the start of the array. There is one byte for each target. The byte at offset zero is for target zero, the next byte is for target one, and so on.</td>
</tr>
<tr>
<td><code>isc-&gt;my_isc</code></td>
<td>Index into the <code>isc_table</code> array that will yield a pointer this structure.</td>
</tr>
</tbody>
</table>
isc->bus_type
Bus type of the interface card.

isc->if_reg_ptr
This is a virtual address corresponding to a base address register that has already had this virtual mapping done to make it usable by the driver and system.

isc->if_drv_data
Pointer to the scsi_bus structure. It is NULL if the bus is not open. SCSI services set this field and a transport driver is not allowed to write to this field.

isc->if_isc
Pointer to private data structure for use by transport driver.

isc->if_id
Unique ID (device and vendor ID) of the transport.

The SCSI subsystem maintains an array of isc_table_type structures in scsi_isc[]. It is the driver's responsibility to assign its isc structure to an element in the scsi_isc[] array indexed by its instance number.

Transport Driver Switch

This structure defines SCSI transport driver entry points and parameters as required by SCSI services. The transport driver's attach routine must initialize the ifsw field of the isc_table_type entry to point to a scsi_ifsw_structure. The contents of the scsi_ifsw structure specify the transport driver entry points and operational parameters to the SCSI subsystem. A detailed description of the fields are:

```c
struct scsi_ifsw {
    ubit8 if_flags;
    ubit8 if_max_tag;
    unsigned int if_scb_size;
    unsigned int if_lun_size;
    unsigned int if_tgt_size;
    unsigned int if_bus_size;
    int (*if_open)(dev_t dev);
    void (*if_close)(dev_t dev);
    void (*if_start)(struct isc_table_type *isc);
    int (*if_abort)(struct buf *bp);
    int (*if_bdr)(struct buf *bp);
    int (*if_reset_bus)(dev_t dev);
} *ifsw;
```

NOTE
If a field is already described earlier, it will only be mentioned here. For more details on such a field, please refer to earlier sections.

ifsw->if_flags
Transport driver flags convey information to the SCSI services on what it supports and what not. The possible flags are:

- **IF_BUS_TAGS**: This is a default tag.
- **IF_NO_TAGS**: Transport driver does not support tags.
- **IF_B2_LIST**: If set, it indicates the transport driver supports handling of disk sort merge buffers.
- **IF_OWNS_TAGS**: Transport driver owns tagged queueing.

ifsw->if_max_tag
One less than the number of per-bus tags supported by the transport driver. A tag is used to differentiate I/O requests. The SCSI subsystem will use tags from zero through ifsw->if_max_tag, inclusive. Actually, the transport driver is not required to use the tags allocated by the SCSI.
subsystem, but the SCSI subsystem will not allow more than ifsw->if_max_tag+1 active I/Os to the bus at any given time (this includes untagged I/Os).

ifsw->if_scb_size
The number of bytes the SCSI subsystem shall allocate and attach to each scb for use by the transport driver. The if_scb field of the scb structure is initialized at scb creation time by services and the data area is bzero'ed by services for each I/O attempt prior to putting the I/O on the select queue. It is not touched by services at any other time.

ifsw->if_lun_size
The number of bytes the SCSI subsystem allocates and attaches to each scsi_lun structure for use by the transport driver. The if_lun field of scsi_lun structure is a pointer to ifsw->if_lun_size bytes for the use of the transport driver.

ifsw->if_tgt_size
The number of bytes the SCSI subsystem allocates and attaches to each scsi_tgt structure for use by the transport driver. The if_tgt field of scsi_tgt structure is a pointer to ifsw->if_tgt_size bytes for the use of the transport driver.

ifsw->if_bus_size
The number of bytes the SCSI subsystem allocates and attaches to each scsi_bus structure for use by the transport driver. The if_bus field of scsi_bus structure is a pointer to ifsw->if_bus_size bytes for the use of the transport driver.

ifsw->if_open
Pointer to the transport driver’s logical unit close function. This is optional for a transport driver.

ifsw->if_start
Pointer to the transport driver’s start function.

ifsw->if_reset_bus
Pointer to the transport driver’s Bus Reset function. This is optional for a transport driver.

ifsw->if_bdr
Pointer to the transport driver’s Bus Device Reset function. This is optional for a transport driver.

ifsw->if_abort
Pointer to the transport driver’s Abort function. This is optional for a transport driver.

ifsw->if_io_max_size
Maximum size of I/O request supported by the transport driver. A value of zero specifies no limit. If set, I/O requests for more than the supported size will be erred back by the SCSI services.

ifsw->if_beg_align, ifsw->if_end_align
Transport driver data buffer alignment requirement. These fields must be set to (n -1) where n is a power of two (2). SCSI services will ensure that data buffer (bp->b_un.b_addr) is n-byte aligned. The maximum of both the fields is used for buffer alignment.

**struct buf Structure**

This structure is the header for buffers in the buffer pool and otherwise used to describe a block I/O request. I/O requests are passed to the transport driver in the form of buf structure. Some of the fields that are of interest to a transport driver writer are explained here:

```c
struct buf {
    int32_t b_flags;
    struct buf *av_forw;
    struct buf *av_back;
};
```
union { caddr_t b_addr;
    } b_un;
int32_t b_bcount;
dev_t b_dev;
uint16_t b2_flags;
struct buf * b_merge;
uint16_t b_merge_cnt;
space_t b_spaddr;
    long b_s2;
} *bp;
#define b_scb b_s2

The transport driver is *not allowed* to set the following fields:

- b_flags
- b_un.b_addr
- b_bcount
- b_dev
- b2_flags
- b_merge_cnt
- b_spaddr
- b_s2

bp->b_flags

B_READ is the only bit bp->b_flags that is of interest to the transport driver and only if bp->b_bcount is not zero. If B_READ is set, the I/O has data in phase; if clear, the I/O has data out of phase.

bp->av_forw

Position on free list of buffers if not busy. This field is used to save a pointer to the buf structure which is passed to scsi_cbfn() after an I/O request is completed.

bp->av_back

Position on free list of buffers if not busy.

bp->b_un.b_addr

Kernel virtual address of the data buffer for the I/O. This is passed to the DMA mapping routines in the transport driver.

**NOTE**

This address may not be cache aligned. This has implications for a read request when part of the cache line is modified by a processor write. The data after the I/O completion will be stale if cache flush occurs after inbound DMA.

Transport drivers must do I/O's to a temporary location for non-cache aligned portions and copy data from temporary buffers to actual data buffers after DMA completions. This differs from the buffer alignment requirement of the transport driver.

bp->b_count

Maximum number of bytes that should be transferred for the I/O.

bp->b_dev

Device number of the destination for the I/O. This is used to obtain a pointer to the scsi_bus structure, a pointer to the scsi_tgt structure, a pointer to scsi_lun structure and a pointer to the isc structure when only the buf structure is available.
SCSI services provide the following services to work with the device number:

- m_scsi_dev()
- m_scsi_tgt
- m_susi_lun
- m_scsi_isc

bp->b2_flags

Additional flags to support B2_LIST buffers. If B2_LIST flag is set in bp->b2_flags, bp represents a disksort merge buffer. transport driver specifies its capability of handling such buffers by setting IP_B2_LIST in ifsw->if_flags.

bp->b_merge

The b_merge field of the first bp represents a linked list of buf structures containing the actual data. The list itself is chained using the b_merge fields of subsequent buf structures.

The b_count field of the first bp represents the total data length in all buffers.

If this is a merge buffer, all data buffers (bp->b_un.addr) are page aligned and bp->b_count will be a multiple of page size Number of Bytes per Page (NBPG).

bp->b_merge_cnt

A count of merged requests. If this field is non-zero, DMA mapping is done via bp_dma_setup() instead of dma_setup().

bp->b_spaddr

Space address of b_un.b_addr. This is passed to the DMA mapping routines in the transport driver.

bp->b_scb (or b_s2)

This field is a pointer to the scb associated with this bp. It is used in a transport driver to obtain the scb struct which will have additional information on the I/O. Refer to the earlier discussion on struct scb for additional details.

Logical Unit Structure

A SCSI Logical Unit (LUN) structure is created per SCSI LUN. This structure is allocated and initialized when the LUN is first opened, and is deallocated on the last close by the SCSI services. It is owned by the SCSI services, but can be accessed by a transport driver. Only the fields relevant to a transport driver are described:

```c
struct scsi_lun {
    struct scsi_tgt *tgt;
    void *if_lun;
    ubit8 lun_id;
    dev_t dev_minor;
    ubit32 open_cnt;
    ubit32 state;
} *lp;
#endif
#define L_TAGS 0x20
```
The transport driver is *not allowed* to set the following fields:

- `tgt`
- `lun_id`
- `dev_minor`
- `open_cnt`
- `state`

`lp->tgt`  
Pointer to the `scsi_tgt` structure to which this logical unit structure belongs, i.e., it is a pointer to the logical unit structure’s parent in the open device tree.

`lp->if_lun`  
Pointer to the data area allocated and zeroed on first open, by the SCSI subsystem, for use by the transport driver. Its size is specified by the transport driver in `isc->ifsw->if_lun_size`. This memory is allocated during the first open of the LUN and is freed during the last close of the LUN.

`lp->lun_id`  
The logical unit’s identification number.

`lp->dev_minor`  
Minor number of the device minus the volume bits i.e., the device number independent portion of the minor number.

`lp->open_cnt`  
The number of successfully completed opens that have not had a corresponding close. It includes block, character and pass-through driver opens and closes.

`lp->state`  
`L_TAGS` is the only bit of `lp->state` that is defined for the transport driver; all other bits are undefined. The transport driver will send a simple queue tag message when initiating an I/O to the logical unit represented by `lp` if (`lp->state` & `L_TAGS`) is set. The (`lp->state` & `L_TAGS`) bit will never change when there are active I/O's on the logical unit.

**Target Structure**

A SCSI target structure is created per SCSI target. This structure is allocated and initialized when a LUN connected to the target is first opened, and is deallocated on the last close of a LUN connected to the target by the SCSI services. This is owned by the SCSI services, but can be accessed by a transport driver:

```c
struct scsi_tgt {
    uint32 open_cnt;
    uint32 state;
    uint8 tgt_id;
    u_char min_sdtr_period;
    struct scsi_bus *bus;
    void *if_tgt;
    struct scsi_lun *lun[SCSI_MAX_LUN_ID+1];
} *tp;
```

#define T_ENABLE_WDTR 0x40
#define T_ENABLE_SDTR 0x20
The transport driver is not allowed to set the following fields:

- `open_cnt`
The number of successfully completed opens that have not had a corresponding close. It includes block, character and pass-through driver opens and closes for all logical units of this target.

- `state`
  Two bits are defined. `T_ENABLE_WDTR` directs the transport driver to initiate SDTR negotiation after any event other than WDTR negotiation that causes the data transfer width to be reset to eight bits that is, bus reset or BDR. If `T_ENABLE_WDTR` is not set, the transport driver is forbidden to initiate WDTR negotiation, but not to respond.
  `T_ENABLE_SDTR` directs the transport driver to initiate SDTR negotiation after any event other than SDTR negotiation that causes the synchronous data transfer parameters to be reset to asynchronous that is, bus reset, BDR or WDTR negotiation. If `T_ENABLE_SDTR` is not set, the transport driver is forbidden to initiate SDTR negotiation, but not to respond.

- `tgt_id`
  SCSI ID of this target.

- `min_sdtr_period`
  Lower limit on the transfer period that should be used for synchronous transfers to this target. It may or may not be smaller than what is supported by the hardware. This value may change at any time and the transport driver is responsible for checking it before initiating every I/O to the target. If the currently negotiated transfer period is smaller than what is supported by the hardware. This value may change at any time and the transport driver is responsible for checking it before initiating every I/O to the target. If the currently negotiated transfer period is smaller than `tp->min_sdtr_period` and `(tp->state & T_ENABLE_SDTR)` is set, the transport driver should initiate SDTR negotiation to correct the situation.

- `bus`
  Pointer to the target’s parent `scsi_bus` structure in the open device tree.

- `if_tgt`
  Pointer to the data area allocated and zeroed on first open by the SCSI subsystem for use by the transport driver. Its size is specified by the transport driver in `isc->ifsw->if_tgt_size`. This memory is allocated during the first open of a LUN on this target and is freed during the last close of a LUN on the target.

- `lun`
  Array of pointers to the `scsi_lun` structures for open logical units of the target. `tp->lun[x]` is NULL if and only if logical unit x is not open.
Bus Structure

A SCSI bus structure is created for each SCSI bus. This structure is allocated and initialized when a LUN connected to a target on the SCSI bus is first opened, and is deallocated on the last close if a LUN connected to the target on the SCSI bus by the SCSI services. This is owned by the SCSI services, but can be accessed by a transport driver:

```c
struct scsi_bus {
    u_int open_cnt;
    struct isc_table_type *isc;
    void *if_bus;
    u_char bus_id;
    scsi_lock_t *lock;
    struct scsi_tgt *tgt[SCSI_MAX_TGT_ID+1];
    struct buf *select_q;
} *busp;
```

The transport driver is not allowed to set the following fields:

- `open_cnt`
- `bus_id`
- `lock`
- `tgt`

`busp->open_cnt` The number of successfully completed opens that have not had a corresponding close. This includes block, character and pass-through driver opens and closes for all logical units of all targets of this bus.

`busp->isc` Pointer to the bus’ parent `isc_table_type` structure in the open device tree.

`busp->if_bus` Pointer to the data area allocated and zeroed on first open by the SCSI subsystem for use by the transport driver. Its size is specified by the transport driver in `isc->ifsw->if_bus_size`. This memory is allocated during the first open of a LUN on this bus and is freed during the last close of a LUN on this bus.

`busp->bus_id` Index into the `scsi_isc` array that will yield a pointer to `busp->isc`.

`busp->lock` SCSI bus spinlock. This lock needs to be held by the transport driver when calling `scsi_enqueue/scsi_dequeue/scsi_dequeue_bp`. The `scsi_bus_lock()` and `scsi_bus_unlock()` services are used to acquire and release the lock respectively.

`busp->tgt` Array of pointers to the `scsi_tgt` structures for open targets of this bus. `busp->tgt[x]` is NULL if and only if target x is not open.

`busp->select_q` A doubly linked list of `buf` structures onto which services places I/O’s ready for selection. The transport driver picks up I/O requests from this queue.
**select_q Structure**

This is the only data structure that is shared between the SCSI subsystem and the transport driver — both may change it. It is the per-bus select queue as shown:

```c
struct scsi_bus {
    struct buf *select_q;
} *busp;
```

After initializing the `bp` and its `scb` and zeroing `*scb->if_scb`, the SCSI subsystem enqueues the `bp` onto the select queue using `scsi_enqueue(&busp->select_q, bp, TAIL)`. Once the SCSI subsystem enqueues a `bp` onto the select queue, it will not modify the `bp`, its `scb` or the `if_scb` until the transport driver returns the `bp` via `scsi_cbfn`, nor will the SCSI subsystem remove a `bp` from the queue once it has enqueued it.

The transport driver must treat the select queue as ordered for any one logical unit and execute a Request Sense during Contingent Allegiance in response to any Check Condition; that is, the transport driver must initiate all I/O's for any logical unit in the order they were enqueued by the SCSI subsystem, and it must not initiate any other command to the logical unit after a Check Condition until the associated Request Sense clears the Contingent Allegiance.

The transport driver may dequeue from and enqueue to the head of the select queue with `scsi_dequeue(&busp->select_q, HEAD)` and `scsi_enqueue(&busp->select_q, bp, TAIL)` provided it observes the ordering requirement. It may also dequeue a specific `bp` using `scsi_dequeue_bp(&busp->select_q, bp)` with the same restriction.

The transport driver does not access the queue in any other way than through the access functions provided by the SCSI subsystem as mentioned in this section.

The transport driver needs to hold the SCSI bus lock while calling `scsi_enqueue/scsi_dequeue/scsi_dequeue_bp, scsi_bus_lock()` and `scsi_bus_unlock()` services are used to acquire and release the SCSI bus lock respectively.

**Data Structure Diagram**

To complete the discussion on the various data structures that are of relevance to a transport driver, the diagram in *Figure 12-1, “SCSI Data Structures,”* illustrates the inter-relationship between different data structures in the SCSI subsystem and other kernel data structures.
Figure 12-1  SCSI Data Structures
I/O Path

An I/O request typically passes through different queues in the SCSI subsystem when it passes from one layer to another. The following is a brief description of these queues.

**scb queue**  
Per-LUN queue defined in the device driver that contains requests that are waiting for an `scb`. Requests are enqueued by the device driver's `dd_strategy` routine and dequeued by the PD's `dd_start` routine.

**tag queue**  
Per-bus queue that contains requests waiting for a qtag. Requests ready to be retried (that is, requests on the retry queues that have hit their “time-to-retry”) are placed on the tag queue to be restarted; they already have an `scb` (from before) but need a new qtag assignment.

**nexus queue**  
Per-LUN queue that contains requests that have to wait to be started to avoid exceeding the LUN's queue depth (`lp->max_active`).

**select queue**  
Per-bus queue that contains requests that are ready to be handed off to the SCSI HBA Driver.

**retry queue**  
Per-bus queue that contains requests that are needed to be retried at some point in the future. The requests are ordered in the queue in “timeout order” (that is, they're ordered by the time when the request will be retried).

The following *Figure 12-2, “I/O Flow Between Various Queues,”* illustrates all the queues involved in a typical I/O path.
The SCSI device driver is responsible for enqueueing the incoming I/O requests in its queue. Device drivers typically call `scsi_strategy()` of SCSI services for this purpose. The device driver specifies a `dd_strategy` entry point in `scsi_ddsw` structure for SCSI services to call the actual device driver routine that does the queueing.

SCSI services, in addition to calling the device driver strategy routine, provide:

- I/O buffer alignment as required by the transport driver.
- I/O forwarding, to do further processing on the CPU slated to handle the I/O completion.

The device driver’s strategy routine should minimally enqueue the buffer on `lp->scb_q`. The I/O start time should be recorded in `bp->b_qstart` if the I/O request is to be timed.

A SCSI device driver owns the following queues:

- `lp->scb_q`: Device driver I/O queue
- `dd_lun->lun_disk_queue`: `sdisk` driver’s queue of sorted I/O requests
- `lp->special_scb_q`: `sctl` pass-through driver I/O request queue
- `lp->priority_scb_q`: Priority mode I/O queue
The device driver strategy routine may return an error if an I/O request encounters any errors while queueing the I/O request. `dd_strategy` returns nonzero status to indicate an error. If I/O is queued successfully, SCSI services act on it for further processing.

**I/O Within SCSI Service Layer**

The SCSI service layer is responsible for:

- Passing I/O requests from device driver to transport driver.
- Implementing flow control policies to honor device I/O queue depth.
- Ensuring fair distribution of shared bus resources between different LUNs (tags, scbs).
- Keeping track of I/O time.
- Handling I/O completion.
- Retrying I/O requests if needed.

SCSI services allocate the required resources for the I/O. If tag resources are not available the buffer is queued to `busp->tag_q`. If nexus resources are not available the buffer is queued to `lp->nexus_q`. If all the resources are allocated SCSI services place the buffers to `busp->select_q`. The transport driver is called through the `ifsw.if_start` entry point for processing all the I/O requests queued on its `select_q`.

Transport drivers return completed I/O requests to SCSI services by calling `scsi_cbfn()`. If I/O is not completed successfully, depending on the I/O return status, a device driver can enqueue the I/O to `busp->retry_q`.

The SCSI services layer owns the following queues:

- `busp->tag_q`: SCB is allocated and I/O is initialized; waiting for per bus resource.
- `lp->nexus_q`: I/O request got SCB and tag resources; waiting for nexus resource.
- `busp->retry_q`: I/O is being retried on failure; tag and nexus resources are fixed. Waiting for the timeout period to be queued.
- `busp->select_q`: I/O is enqueued to the interface layer. I/O is owned by the transport driver until it is returned to SCSI services by calling `scsi_cbfn()`.

**I/O Within SCSI Interface Layer**

The transport driver’s `if_start` routine must dequeue each I/O on its `select_q` and do whatever is required to execute the I/O request. The SCSI transport driver is responsible for:

- Executing the I/O request.
- Timing the I/O request as requested by upper layers.
- Returning I/O to SCSI services by calling `scsi_cbfn()`.
Transport Driver Development

Developing a transport driver includes the following steps:

1. Driver installation and initialization
2. Register mapping and DMA
3. Protection and synchronization
4. Sending SCSI I/O requests to the target
5. Processing Completed I/O requests
6. Interrupt Handling
7. Special Routines

Driver Installation and Initialization

A SCSI transport driver must have a `driver_install` routine and a `driver_attach` routine. If the transport driver controls and interacts with a hardware device, the driver is also required to have an interrupt service routine to handle the device interrupts. Typically a `driver_init()` routine is assigned to the `isc->gfsw->init` field which will be called by the WSIO.

For a detailed discussion on the `driver_install()`, `driver_attach()`, `driver_init()` and `driver_isr()` routines, refer to Chapter 4, “Writing a Driver.”

The following sections explain the specific transport driver installation and initialization routines.

Calling `driver_install()`

When the HP-UX system is configured through the `config` command, a table of `driver_install()` entry points is created from information in `/stand/system`. When `driver_install()` is called by the I/O system configuration process through the `driver_install()` entry point configured in the system, the `driver_install()` routine places the `driver_attach()` entry point in the table of drivers to be called at configuration time. `driver_install()` then calls the `wsio_install_driver()` routine to register the driver with the I/O subsystem and returns any error.

SCSI Specific Part

Every transport driver has to register a probe function with the WSIO to identify the targets connected to the interface card. This is typically the next step in attaching the `driver_attach()` routine. The SCSI subsystem provides the probe function `parallel_scsi_probe` to perform the probe and the WSIO provides `wsio_register_addr_probe()` to register a probe function with the WSIO. The SCSI transport driver sets the “class” field of the `drv_info_t` to “ext_bus”.

The following example shows a call to `driver_install()`:

```c
static drv_ops_t qlisp_drv_ops = {
    NULL, /* d_open */
    NULL, /* d_close */
    NULL, /* d_strategy */
    NULL, /* d_dump */
    NULL, /* d_psize */
    NULL, /* d_mount */
    NULL, /* d_read */
    NULL, /* d_write */
    NULL, /* d_ioctl */
    NULL, /* d_select */
};
```
Writing a SCSI Interface Driver

Transport Driver Development

```c
static drv_info_t qlisp_drv_info = {
    "qlisp", /* name */
    "ext_bus",/* class */
    DRV_SAVE_CONF | DRV_MP_SAFE | DRV_SCAN, /* flags */
    NULL,  /* b_major */
    NULL,  /* c_major */
    NULL,  /* cdio */
    NULL,  /* gio_private */
    NULL,  /* cdio_private */
};

static wsio_drv_data_t qlisp_wsio_data = {
    "scsi", /* drv_path */
    T_INTERFACE, /* drv_type */
    DRV_CONVERGED, /* drv_flags */
    NULL, /* dvr_minor_build - field not used */
    NULL, /* dvr_minor_decode - field not used */
};

static wsio_drv_info_t qlisp_wsio_info = {
    &qlisp_drv_info, /* driver info */
    &qlisp_drv_ops, /* driver ops */
    &qlisp_wsio_data, /* driver data */
    WSIO_DRV_CURRENT_VERSION
};
extern int (*pci_attach)(uint32_t id,
struct isc_table_type *isc);
/* To attach PCI driver to system */
static int (*qlisp_saved_pci_attach)();

int qlisp_install(void) {
    /* Link driver attach function into chain */
    qlisp_saved_pci_attach = pci_attach;
    pci_attach = qlisp_pci_attach;

    /* Register the driver probe function to identify 
    * the targets */
    wsio_register_addr_probe(parallel_scsi_probe, "qlsip" transitional
    wsio_unregister_probe(WSIO_ADDR_TYPE, "qlsip" transitional
    /* Register driver with WSIO */
    if (!wsio_install_driver(&qlisp_wsio_info)) {
        (void)wsio_unregister_probe(WSIO_ADDR_TYPE, "qlsip" transitional

        /* Remove driver from attach chain as cleanup */
        pci_attach = qlisp_saved_pci_attach;
        return 0; /* WSIO driver install failed */
    }
    return (1); /* 1 = WSIO driver successfully installed */
}
```

WSIO allows a driver to register its own probe function used by WSIO to scan for devices underneath an interface card. This is usually required if the ioscan can’t reach the device in question on its own. If a device is behind a bus for which there is no bus nexus CDIO in the OS, it is the responsibility of the driver to let WSIO notify it can scan by setting the DRV_SCAN flag in drv_info_t structure and provide a handle to the WSIO by registering a probe function with the WSIO. For a detailed discussion on the driver probe function, refer to Chapter 4, “Writing a Driver.”
Calling driver_attach()

Use the driver_attach() routine to determine whether the product ID passed in matches the driver_attach device and vendor ID. If the IDs do not match, the driver_attach() routine calls the next attach routine in the chain by calling the *driver_saved_attach() routine.

The driver_attach() routine may be called many times before a match is found. For the device in the first slot, the associated driver_attach() routine is called by the number of devices in the PCI backplane. For the device in the last slot of the PCI backplane, the associated driver_attach() routine is called only once.

When the driver_attach() routine recognizes the device ID, it takes actions such as allocating and initializing its driver control blocks and PCI I/O registers. driver_attach() also sets up a driver initialization routine and finally calls isc_claim() to claim the device.

SCSI Specific Part

A SCSI transport driver has additional requirements for hardware that it controls. It must inform the SCSI subsystem of this SCSI hardware by assigning the appropriate pointer in the global scsi_isc array to isc. The index of the appropriate pointer in scsi_isc is the bus ID for the SCSI bus. The index into this array can be obtained by calling the WSIO routine wsio_isc_to_instance(isc, NULL).

Another responsibility of the SCSI transport driver is to attach a fully initialized scsi_ifsw structure to the isc structure via isc->ifsw field. This is typically performed in the transport driver init routine. Each bus could have its own scsi_ifsw if necessary, but it’s more likely that a single scsi_ifsw could be shared by all buses being managed by the same transport driver.

The transport driver init routine also sets the initiator ID, SCSI transfer rates and bus width fields in the isc structure by calling SCSI_GET_INITIATOR_PARAMS macro. The bus_max_width field in the isc structure is important. The parallel_scsi_probe() uses this field to actually scan the SCSI bus. If this field is not set, the SCSI services do not scan on the SCSI bus behind the interface card. More information is provided later in this chapter.

Device initialization, allocating and mapping SCSI command queues, registering device interrupts, and so forth, can be done in the driver_init() routine. In the attach routine, isc->gfsw->init field is assigned with a pointer to the driver_init() routine. This can also be done by calling the macro CONNECT_INIT_ROUTINE(isc, <driver>_init). The HP-UX configuration continues the initialization by calling the driver init routines after calling the driver attach routine.

An example driver_attach() routine and driver_init() routine are shown:

```c
#define QLISP_ID_SUPPORTED(x) ((x) == QLISP_ID_1020 || (x) == QLISP_ID_1040 || (x) == QLISP_ID_1240 || (x) == QLISP_ID_1080 || (x) == QLISP_ID_1280 || (x) == QLISP_ID_12160 )

static int qlisp_pci_attach(uint32_t id, struct isc_table_type *isc)
{
    /* Check if it is the correct device */
    if (QLISP_ID_SUPPORTED(id)) {
        /* Check that the PCI register set is correctly * mapped */
        if(isc->if_reg_ptr == NULL) {
            msg_printf("qlisp:
                      Mapping QLISP registers failed!\n");
            goto recover1;
        }
    }
    /* Allocate driver control structures */
```
/* Initialize the fields in the driver control */
/* Connect the driver init routine */
CONNECT_INIT_ROUTINE(isc, qlisp_init);

/* Claim the device */
isc_claim(isc, &qlisp_wsio_info);
return qlisp_saved_pci_attach(id, isc);

recover1:
/* Error processing */

}  

static int
qlisp_init(struct isc_table_type *isc) {

/* Perform SCSI specific steps */
/* 1. Assign the scsi_isc array with the
   * interface's isc. SCSI interface drivers must
   * check the instance number returned by
   * wsio_isc_to_instance(). If the returned value
   * is greater than SCSI_MAX_BUS_ID, the driver
   * should return WSIO_ERROR. */
if((instance = wsio_isc_to_instance(isc, NULL)) >
   SCSI_MAX_BUS_ID) return WSIO_ERROR;
scsi_isc[instance] = isc;

/* 2. Attach the interface driver switch */
/* Allocate memory for scsi_ifsw used by SCSI services */
/* Initialize the ifsw fields */
/* Attach the struct ifsw to isc */
isc->ifsw = (caddr_t)qlisp_ifsw;

/* 3. Call SCSI_GET_INITIATOR_PARAMS macro to get the
   * initiator ID and the SCSI rate. */
/* Set the following fields of the isc with the correct
   values: isc->my_address
   *isc->bus_max_width
   *isc->bus_min_sdtr_period
   *isc->bus_max_reqack_offset
   */
/* Configure PCI config space on the interface card */
/* Allocate and map memory for SCSI command queues */
/* Reset the interface card */
/* Allocate interrupt objects and activate the interrupt */
return 0;
Setting SCSI Parameters

Some of the state information about the SCSI bus and targets on the bus must persist across opens, but the data structures are deallocated when a device is closed. This information is kept to the following fields of the isc structure:

- `isc->my_address`
- `isc->bus_max_width`
- `isc->bus_min_sdtr_period`
- `isc->bus_max_reqack_offset`

For additional details on these fields, refer to the discussion on the `isc` structure earlier in this document.

The driver has to set these fields in the driver's `init` routine. In the current implementation of the SCSI subsystem it is required to interact with the processor dependent code on PA-RISC platforms to obtain the SCSI parameter values. However, this may change in future platforms. To protect drivers from these changes, the DDK includes the `SCSI_GET_INITIATOR_PARAMS` macro. It wraps the processor dependent calls and passes back the values of the SCSI parameters. For correct operation of the macro, do not make any modifications to it.

**NOTE** Include the file `scsi_params_macro.h` distributed with the *Driver Development Kit* to access this macro.

**SCSI_GET_INITIATOR_PARAMS**

This macro is called with four arguments: a pointer to the `isc` structure, and placeholders for the initiator ID, SDTR period, and REQ/ACK offset. The last three parameters to the macro correspond to the following fields in the `isc`:

- `my_address`
- `bus_min_sdtr_period`
- `bus_max_reqack_offset`

**NOTE** `SCSI_GET_INITIATOR_PARAMS` does not set the fields in the `isc` structure; it just returns the values of the corresponding fields.

The following pseudo code of a transport driver's `init` routine illustrates the calling convention of the `SCSI_GET_INITIATOR_PARAMS` macro and how to set the fields in the `isc`.

```c
int driver_init()
{
    ...
    /* Declare local variables to pass to the macro */
    int initiator_id, sdtr_period, reqack;
    ...

    /* Call the macro to obtain the SCSI parameters */
    SCSI_GET_INITIATOR_PARAMS(isc, initiator_id,
        sdtr_period, reqack);

    /* Set the corresponding fields in the isc with the *
    * values obtained above. */
    */
```
isc->my_address = initiator_id;
isc->bus_min_sdtr_period = sdtr_period;
isc->reqack_offset = reqack;

/* Set the SCSI bus width */
isc->bus_max_width = width;/* where width equals
* to 8 or 16 */

SDTR/WDTR Negotiation

SCSI host bus adapters and targets negotiate for synchronous data transfer rates and wide data transfer widths. **Synchronous Data Transfer Request** (SDTR) and **Wide Data Transfer Request** (WDTR) commands are used for this. A transport driver need not do anything for this. However, in some scenarios like bus reset, or when a slower rate device is connected on the bus, the transport driver may force a negotiation on SDTR and WDTR.

Current Negotiated Information

The currently negotiated synchronous data transfer period per target is stored in one byte of the tgt_sdtr_period array. A pointer to the byte associated with a target is kept in the scsi_tgt structure while the device is open for easy access, and consistent with other target specific state information. A value of 0xff implies asynchronous data transfers otherwise, the value is the minimum transfer period from the SDTR agreement.

The currently negotiated bus width for each target is stored in one bit of the tgt_wdtr_width bit field. As with tgt_sdtr_period, a pointer to the bit field is kept in the scsi_tgt structure. If the bit is set, 16-bit wide transfers have been agreed upon.

Two other bit fields in the isc_table_type structure associated with SDTR and WDTR are the tgt_sdtr_done and the tgt_wdtr_done bit fields. There is one bit per target in each, indicating whether or not SDTR or WDTR negotiation has been done since the most recently recognized event, which reset the device parameters to their default value.

When to Negotiate

The transport driver clears the appropriate tgt_sdtr_done and tgt_wdtr_done bits, sets the appropriate tgt_sdtr_period bytes to 0xff, and clears the appropriate tgt_wdtr_width bits in the isc_table_type structure when it recognizes a device reset. The transport driver recognizes a device reset when it initiates or detects a SCSI bus reset or sends a **Bus Device Reset** (BDR) message to the device. The transport driver also clears the appropriate tgt_sdtr_done bit when WDTR is negotiated. The transport driver always initiates SDTR or WDTR if the corresponding “done” bit in the isc_table_type structure is clear, and the T_ENABLE_SDTR or T_ENABLE_WDTR bit is set in tp->state. It also initiates WDTR or SDTR whenever SCB_WDTR or SCB_SDTR is set in the scb independently of the bits in tp->state. It initiates WDTR and/or SDTR negotiation on every auto-sense if enabled by tp->state. The transport driver initiates negotiation at any other time it deems appropriate.

What to Negotiate

Three parameters are used in determining the minimum transfer period for which to negotiate:

1. tp->min_sdtr_period
2. isc->bus_min_sdtr_period
3. The lowest period at which the hardware can operate.
When initiating SDTR, the transport driver sends a period equal to the maximum of \texttt{tp->min_sdtr_period} and the minimum period supported by the hardware. The transport driver always negotiates for the maximum REQ/ACK offset allowed by the hardware.

Unlike for SDTR, there is no way to control bus width negotiation; it is all or nothing. Either negotiation is enabled or not. If negotiation is initiated, it will be for the maximum supported by the bus.

**Register Mapping and DMA**

The following is a brief discussion of PCI register mapping. See Chapter 14, “Writing PCI Device Drivers,” for a detailed description.

For WSIO drivers, the \texttt{if_reg_ptr} member of the \texttt{isc} structure is a **Precision Architecture** (PA) virtual address, corresponding to a base address register that has already had this virtual mapping done to make it usable by the driver and system. If \texttt{if_reg_ptr} is NULL, the driver maps the range itself.

WSIO provides a number of services for DMA mapping. Refer to the manpages in the *HP-UX 11i v1 Driver Development Reference Guide* on WSIO services for a detailed discussion of each. In a transport driver, \texttt{dma_setup()} or \texttt{bp_dma_setup()} is called to set up DMA mapping, depending on whether one \texttt{buf} structure or multiple \texttt{buf} structures merged together.

Fields relevant only to a transport driver:

```c
struct dma_parms {
  int channel;
  int dma_options;
  int flags;
  struct iovec *chain_ptr;
  int chain_count;
  caddr_t addr;
  space_t spaddr;
  int count;
};
```

There is a derived structure from \texttt{dma_parms}.

```c
struct bp_dma_parms {
  struct dma_parms dma_parms;
  struct dma_parms *merge_dma_parms;
};
```

The following steps set up the DMA:

1. Initialize the DMA options.
   - If the device can bus master for DMA, the channel field has to be set to \texttt{BUS_MASTER_DMA}.
     ```c
dma_parms->channel = BUS_MASTER_DMA;
```
   - If you don’t want to wait until the \texttt{dma_setup()} succeeds, \texttt{dma_parms->flags = NO_WAIT}.
   - Set the DMA options.
     ```c
dma_parms->dma_options = ((bp->b_flags & B_READ) ?
                               (DMA_8BYTE|DMA_READ) :
                               (DMA_8BYTE|DMA_WRITE));
```

2. If \texttt{bp->bp_merge_cnt} equal zero, call \texttt{dma_setup()}.
   - Set the address of the buffer to be mapped. \texttt{dms_parms->addr =}
   - Set the space address of the buffer to be mapped. \texttt{dms_parms->spaddr = bp->b_spaddr}.
   - Set the buffer length to be mapped. \texttt{dms_parms->count = bp->b_count}.
     ```c
     ret_code = dma_setup(isc, dma_parms);
     ```
On a successful call to `dma_setup()`, `chain_ptr` in the `dma_parms` structure points to an `iovec` structure that contains the mapped address and the length of the mapping. If there are multiple elements involved, the `chain_count` field is used to obtain the number of DMA elements.

3. If `bp->b_merge_cnt` > 0, call `bp_dma_setup()`. Note that `bp_dma_setup()` takes `bp_dma_parms` as one argument, not `dma_parms`.

On a successful call to `bp_dma_setup()`, `bp_dma_parms->merge_dma_parms` will point to a chain of the `dma_parms` structures, and for each of these `dma_parms` structures, the fields explained previously for `dma_setup()` are applicable.

```c
ret_code = bp_dma_setup(isc, bp, bp_dma_parms);
```

### Protection and Synchronization

The major synchronization issue with transport drivers is to avoid data corruption and race conditions when shared structures are accessed by multiple threads in MP systems. Driver data structures also need protection against interrupts. A spinlock scheme may be chosen to provide finer granularity locks, protecting data structures at finer levels. More information on spinlocks refer to Chapter 3, “Multiprocessing.”

Whenever a transport driver enqueues or dequeues `bp`'s from the `select_q`, it has to acquire `scsi_bus_lock()` and release the lock after the completion of `scsi_enqueue()`, `scsi_dequeue()` or `scsi_dequeue_bp()` by calling `scsi_bus_unlock()`.

For example:

```c
scsi_bus_lock(busp);
scsi_enqueue(&busp->select_q, bp, HEAD);
scsi_bus_unlock(busp);
```

Similarly,

```c
scsi_bus_lock(busp);
bp = scsi_dequeue(&busp->select_q, HEAD);
scsi_bus_unlock(busp);
```

Protect driver control structures with a spinlock to protect across MP access.

A lock order of `(SCSI_LOCK_ORDER_BASE + 2)` can be used by the transport driver while allocating new spinlocks.

For example:

```c
typedef struct {
    uint32_t state;
    ubit8 chip_rev;
    ...

    lock_t *mp_lock;
    ...
} qlisp_shared_isc_t;
```

The driver does not hold spinlocks across calls to different subsystems. Before calling the SCSI services, the driver should release the spinlocks and acquire them if necessary when the transport driver reestablishes control.
Sending SCSI I/O Requests to the Target

The SCSI subsystem requires the transport driver to specify a start function. This is the entry point for an I/O request at the transport driver. The following pseudo code is for a typical transport driver start routine.

```c
This routine is called with a pointer to the isc structure.
Obtain the scsi_bus structure from the isc.
(NOTE: isc->if_drv_data field points to the scsi_bus structure)

do
    acquire scsi_bus lock
    Get a bp from the select_q
    release scsi_bus lock
    if bp is not NULL
        get SCSI Control Block from the bp.
        if there are bytes to transfer
            (bp->b_bcount != 0)
                DMA map the buffer(s)
                Get the hardware I/O control block(s)
                If the I/O request can't be satisfied
                because of insufficient resources at the hardware, set the scb->cdb_status to
                SCTL_INVALID_REQUEST and call scsi_chfn().
                If the I/O request can't be satisfied
                because of temporary resource shortage,
                then put the bp back at the HEAD
                of the select_q.
                Build the SCSI command
                Inform the hardware of a pending I/O request
done

until there are bp's in the select_q
```
**Processing Completed I/O Requests**

I/O requests which are completed are processed when the I/O card either generates an interrupt or informs by some other means.

The transport driver has to inform the SCSI subsystem that an I/O request is completed so the resources allocated for that I/O can be freed. The SCSI subsystem provides the transport driver with the service `scsi_cbfn()` to accomplish this. However before calling the SCSI callback services, fields in the SCB structure have to be set to indicate the result of an I/O request. An I/O might have completed successfully, or it might have resulted in a Check Condition and an auto-sense message is returned. An I/O might have failed because of lack of resources, an invalid command, or an I/O might have timed out. In any of these scenarios, the SCSI subsystem should know the result of the I/O for further processing. It is the responsibility of the transport driver to set the fields in the SCB structure to provide the correct result to the SCSI subsystem.

For details of the fields in the SCB structure, refer to the “SCSI Control Block” section. The following pseudo code defines the steps required to set up the fields of the transport driver. In the examples, `scb` is `bp->scb`.

1. If the I/O request is completed successfully, set:
   
   ```c
   scb->sense_bytes = 0;
   scb->data_resid = 0;
   scb->sense_status = SCTL_INCOMPLETE
   scb->cdb_status = S_GOOD;
   ```

2. If there is a select timeout, set:

   ```c
   scb->sense_status = scb->cdb_status = SCTL_SELECT_TIMEOUT;
   scb->data_resid = <value returned from the device.>
   ```

   If this generates an auto-sense, follow step 3 to set other fields. Otherwise, set:

   ```c
   scb->sense_status = SCTL_INCOMPLETE:
   scb->sense_bytes = 0;
   ```

3. If the I/O request resulted in an auto-sense, set:

   ```c
   scb->sense_status = S_GOOD;
   scb->sense_bytes
   scb->sense_bytes = iocb->stat.req_sense_length;
   scb->sense_data = iocb->stat.req_sense_data;
   /*
   ** If data transferred and the sense data has at least the info byte (resid) field, check resid count from ther, else set the field with the value returned from the device.
   */
   ```

   **Code example:**

   ```c
   ....
   scb->data_resid = iocb->stat.residual;
   if (iocb->stat.state_flags & QLISP_STATE_GOT_SENSE) {
     scb->sense_status = S_GOOD;
     scb->sense_bytes = iocb->stat.req_sense_length;
     scb->sense_data = iocb->stat.req_sense_data;
     /*
     ** If data transferred and the sense data has at least the info byte (resid) field, check resid
     ** count from there. Some devices report NO_SENSE
     ** check condition with the resid count in the
     ** sense data and the QLogic card reports zero
   ```
Writing a SCSI Interface Driver
Transport Driver Development

4. If SCSI command transport failed completely, set:

   /* Transport failed entirely, residual = requested length */
   scb->data_resid = bp->b_bcount;
   scb->cdb_status = SCTL_INCOMPLETE;
   scb->sense_status = SCTL_INCOMPLETE;
   scb->sense_bytes = 0;

5. If the I/O request is invalid, set:

   scb->cdb_status = S_BUSY;

6. If a call to dma_setup()/bp_dma_setup() fails, set:

   scb->cdb_status = S_BUSY;

When the control returns to the transport driver after calling scsi_cbfn(), the transport driver does not touch any of the resources of the completed I/O. The transport driver forms a chain of bp's whose I/O has completed. To build such a chain, the transport driver uses the av_forw field of the buf structure. The following code example demonstrates this point:

- bp->av_forw = bp_chain;
- bp_chain = bp;

If the bp_chain initially points to NULL, as bp's become available the buf_chain points to the most recent bp, and the chain can be traversed through the bp->av_forw field.

When calling scsi_cbfn(), bp's from this list are passed to the SCSI service.
Interrupt Processing

Refer to Chapter 4, “Writing a Driver,” as it discusses the issues involved in writing an Interrupt Service Routine (ISR). An ISR is a device specific routine and different drivers do different things when handling a device interrupt. Some generic steps are provided for processing a device interrupt:

1. Acquire a spinlock for protection on management processor systems.
2. Read a device status register.
3. Check for a spurious interrupt; if so, release the lock and return.
4. If an I/O request is completed without any errors, process it.
5. If there is an error, look for the specific event that caused the interrupt.
6. Build a bp chain of the completed I/O’s.
7. Set the device register to indicate if an I/O response is processed.
8. Loop if there are more completed commands.
9. Wake up any I/O requests waiting for device resources.
10. Release the spinlock.
11. Call SCSI callback services on completed I/O’s.
12. Return INTR_SERVICED

Special Routines

A SCSI transport driver can optionally implement the following routines to handle special events such as bus reset, I/O abort, and so forth. A brief description and pseudo code is provided here for each of the routines.

Bus Device Reset

The SCSI subsystem calls the transport driver’s bus device reset (bdr) function in response to an SIOC_RESET_DEV ioctl request.

This function is passed with a buf structure bp as its sole argument. The bp->b_dev field contains the device number of the target. The SCSI subsystem provides m_* services to obtain the required SCSI data structures for executing a BDR.

Pseudo code for a typical BDR routine:

1. Obtain the device number from bp->b_dev.
2. Obtain the scsi_bus structure from m_scsi_bus(bp->b_dev).
3. Obtain the scsi_tgt structure from m_scsi_tgt(bp->b_dev).
4. Obtain the isc structure from the bp->isc.
5. If the device is in reset state, reject the request.
6. Obtain the driver control structures.
7. Issue a BDR command to the device.
8. Set the device to a known state.
Aborting an I/O Request

The SCSI subsystem calls the transport driver’s abort function in response to an `SIOC_ABORT` ioctl request. This function is passed with a `buf` structure `bp` as its sole argument. The `bp->b_dev` field contains the device number of the destination I/O. The SCSI subsystem provides `m_*` services to obtain required SCSI data structures to execute an abort of the specified I/O request.

Pseudo code for a typical abort routine:

1. Obtain the device number from `bp->bp_dev`.
2. Obtain the `scsi_bus` structure from `m_scsi_bus(bp->b_dev)`.
3. Obtain the `scsi_tgt` structure from `m_scsi_tgt(bp->b_dev)`.
4. Obtain the `scsi_lun` structure from `m_scsi_lun(bp->b_dev)`.
5. Obtain the `isc` structure from the `bp->isc`.
6. If the device is in reset state, reject the request.
7. Acquire the driver control structures.
8. Issue a command to abort the LUN device.
9. Set the device to a known state.

Reset Bus

The SCSI subsystem calls the transport driver’s `reset_bus` function to reset the SCSI bus. This function is passed with an `isc` structure as its sole argument.

Pseudo code for a typical `reset_bus` routine:

1. Obtain the driver control structures.
2. Acquire a driver spinlock.
3. Check if the device is under RESET state. If so, release the spinlock and return. Otherwise, set the device state to RESET state.
4. Release the spinlock.
5. Issue a command to the device to reset SCSI bus.
6. Acquire a driver spinlock.
7. Set the device state to `RESET_DELAY` state.
8. Set a `timeout` after 3 seconds.
9. Release the driver lock.
10. After the `timeout` sets, reset the driver state.
11. Call the driver’s `if_start()` function to process I/O requests.

Handling Timeouts

It is a common practice in driver development to use time-outs to schedule events after a pre-determined time interval. The `timeout()` kernel service can be used to schedule an event to occur after a specified amount of time. An `untimeout()` kernel service is also provided to unschedule a time-out. It is particularly important to `untimeout()` any pending time-outs before unloading a DLKM driver. Refer to Chapter 2, “HP-UX I/O Subsystem Features,” for more information on the `timeout()` and `untimeout()` services.
Sleep and Wakeup Mechanism

At times, a transport driver may need to wait for an event such as availability of resources or device establishment of a known state. The transport driver typically calls “sleep”, if it can, from the current context, instead of busy waiting and monopolizing the CPU. When a corresponding event occurs, the driver calls “wakeup” to notify any sleeping threads of this event.

To avoid any race conditions, the kernel spinlock() services provide the get_sleep_lock() call. A get_sleep_lock() obtains a lock before calling sleep and releases the lock after it calls sleep. Similarly, a wakeup() call is also protected with the locks obtained from get_spin_lock(). For a more detailed explanation, refer to Chapter 3, “Multiprocessing.”

Example code for a typical sleep/wakeup pair routine:

```c
/*
 * qlisp_sleep - 
 * Sleep a process on an address 
 */
static void
qlisp_sleep(lock_t *wlock, caddr_t wchan)
{
    vassert(owns_spinlock(wlock));

    (void) get_sleep_lock(wchan);
    spinunlock(wlock); /* wlock is a driver spinlock */
    (void) sleep(wchan, PRIBIO);
    spinlock(wlock); /* wlock is a driver spinlock */
}

/*
 * qlisp_wakeup - 
 * Wakeup processes sleeping on an address 
 */
static int
qlisp_wakeup(lock_t *wlock, caddr_t wchan)
{
    vassert(owns_spinlock(wlock)); /* wlock is a driver spinlock */

    return wakeup(wchan);
}
```
Sample Driver

A sample SCSI transport driver for Qlogic's ISP12160 is provided along with this document as part of the Driver Development Kit. A review of the earlier sections of this chapter will ensure a better understanding of this driver.

Description of the Device

A brief description of the device and its functionality is included to keep the driver in perspective. However, this is not a complete description of the device. For that, refer to Qlogic's hardware reference manual.

The ISP12160A supports dual channel, Ultra3 (Fast-80) SCSI functionality. It interfaces the PCI bus to two Ultra3 SCSI buses and contains an on-board RISC processor. The interface between the ISP12160A firmware and drivers consists of two queues, request and response. The queues are located in host memory and are organized as circular fixed length lists of 64-byte entries First In First Out (FIFOs). They are I/O mapped and their physical addresses are stored in the device registers.

This section presents the code flow of the sample Qlogic driver. Refer to the sample driver code for details. This section is organized as follows:

- Driver Architecture
- Driver Data Structure
- qlisp_multi Driver

Driver Architecture

The sample driver is two drivers; one for claiming the PCI interface card and probing the SCSI channels underneath the PCI card, and another for claiming the SCSI controllers. The first one is "qlisp_multi" and the second driver is "qlisp". This kind of two-driver architecture is required only for special devices with multiple SCSI channels behind a PCI function. Since the HP-UX ioscan can't recognize the SCSI controllers that are underneath a PCI interface, two drivers are required.

The qlisp_multi driver registers a driver specific probe function to scan for SCSI controllers during the driver installation. After the driver claims the card, it sets an WSIO_IDENTIFY_CHILD routine to correctly identify the child node. When the driver's scan finds a SCSI controller, the second driver's attach routine claims the SCSI channels. If an interface card has each SCSI channel on a separate PCI function, this kind of two driver approach is not required.

As far as each driver's functionality is concerned, the qlisp_multi driver is involved only in building the I/O tree with the SCSI controllers. The real work is done by the qlisp driver. If a two driver approach is not needed, the qlisp driver is the one which is of interest.
Data Structures

The following are the main data structures used in the driver:

- **qlisp_isc_t**: Local interface specific control structure (lisc); this is allocated per device instance.
- **qlisp_shared_isc_t**: Common device control structure; this is allocated when the master port is initialized.
- **qlisp_scb_t**: Interface specific SCSI Control Block structure; this is used to keep track of the associated bp and scb.
- **qlisp_bus_t**: Interface specific SCSI bus structure; this is used to locally associate an I/O (based on the tag) with an I/O request.
- **qlisp_q_ent_t**: This is a union of different 64-byte structures which can be entries in the request and response queues. These entries can be: Command, Command A64, Extended Command, Continuation, Continuation A64, Marker, Status and Extended Status.

The driver design facilitates traversing from one data structure to another. The inter-relationship among data structures is shown in Figure 12-3, "Driver Data Structure Inter-Relationship."

Figure 12-3  Driver Data Structure Inter-Relationship

qlisp_multi Driver

This section briefly describes the qlisp_multi driver. The objective is to present the code flow of the qlisp_multi driver.

The qlisp_multi driver is a typical WSIO transport driver. The installation and initialization routines of the qlisp_multi driver are similar to the routines explained in Chapter 4, “Writing a Driver,” of this manual. There are some differences that are specific to the functionality of this driver and they are explained in the following sections.

The qlisp_multi driver claims the interface card and scans for SCSI controller chips underneath it. The driver should be able to scan the hardware. This requires three things:

1. It has to inform the WSIO that it can scan for the hardware.
2. It has to provide a device specific probe function to the WSIO to scan the hardware.
3. It has to inform the WSIO of a child driver identifying routine.
The last two steps are performed in the `qlisp_multi_install()` and `qlisp_multi_attach()` routines. The `DRV_SCAN` flag is set in the `drv_info_t` structure to inform the WSIO that the driver can scan.

```
static drv_info_t qlisp_multi_drv_info = {
    "QLISP_NAME_MULTI",    /* name */
    "qlispha",              /* class */
    DRV_SAVE_CONF | DRV_MP_SAFE | DRV_SCAN, /* flags */
    NULL,                   /* b_major */
    NULL,                   /* c_major */
    NULL,                   /* cdio */
    NULL,                   /* gio_private */
    NULL,                   /* cdio_private */
};
```

Fragments from the `qlisp_multi` driver's install and attach routines are shown here to illustrate the driver steps.

```
int qlisp_multi_install()
{
    ....

    wsio_register_dev_probe(DRV_NAME, &qlisp_multi_dev_probe,
        "QLISP_NAME_MULTI");

    wsio_install_driver(&qlisp_multi_wsio_info);

    ....
}
```

The calling semantics for the device probe routine are:

```
qlisp_multi_dev_probe(this node,
    drv_info,
    probe_id,
    hw_path, isc,
    probe_type,
    name, desc);
```

In the driver attach routine `qlisp_multi_pci_attach`,

```
int qlisp_multi_pci_attach()
{
    ....

    isc_claim(isc, &qlisp_multi_wsio_info);

    wsio_set_parm(isc, WSTO_IDENTIFY_CHILD,
        (void *)&qlisp_multi_identify_child);

    ....
}
```

The calling semantics for the driver identify child routine are:

```
qlisp_multi_identify_child(child_name);
```
Writing a SCSI Interface Driver

Routines

The driver's probe function is called by WSIO during the hardware scan, and if a device is found WSIO calls the driver's child identify routine to verify the driver name.

These routines are required only for a multi-port interface card.

qlisp_multi_dev_probe

To understand a device probe routine, it is necessary to have an understanding of the hardware path.

A hardware path is a numerical string of hardware components, notated sequentially from the bus address to the device address. Typically, the initial number is appended by slash (/), to represent a bus converter (if required by the machine), and subsequent numbers are separated by periods (.). Each number represents the location of a hardware component on the path to the device. The hardware path is defined by the following structure:

```c
/* HW path structure (hw_path(GIO4) - This structure is used
 * to define a hardware path. The first/last indices define a
 * window of address elements which are meaningful. If
 * last_index == (first_index-1) then the path is NULL.
 */
#define MAX_ELEMENTS 14
typedef struct hw_path {
    char first_index;
    char last_index; unsigned
    char addr[MAX_ELEMENTS];
} hw_path_t;
```

The device probe routine is called with the following arguments:

- Pointer to the calling I/O tree node
- Pointer to the driver info structure
- probe_id, which is set by the transport driver
- Hardware path of the calling device
- Pointer to the isc structure
- probe_type
- Pointer to name of the child node.
- Pointer to description of the child node.

When this routine is called for the first time by the WSIO, a probe type of PROBE_FIRST is passed, and the hardware path structure has the fields set to satisfy the following condition:

```
hw_path->first_index == hw_path->last_index + 1
```

When a device is found, the driver increments the last_index, and it becomes equal to the first_index; the probe function places the hardware address of the first device found in the hw_path->addr array at an index of last_index. Later the probe function is called with the probe type of PROBE_NEXT and with first_index == last_index, both of which point to the hardware address of the last found device. When probe type is PROBE_ADDRESS, the probe routine should retrieve the last element of the hardware path and probe the device at that address. The last element is referenced by last_index.
If a device is found, three pieces of information are retained from the device: an “ID” (the product and vendor ID strings), a “description” (driver description), and a “name” (the expected interface/device path to this device, for example, “scsi_disk”). The WSIO CDIO will use the “name” property to recognize a corresponding driver and match the device when the node is later claimed.

The driver should sense when the address range is out of range and should return an error.

If a device is found the routine returns PROBE_SUCCESS or PROBE_UNSUCCESSFUL. The following is a sample device probe routine which illustrates the concepts.

```c
static int qlisp_multi_dev_probe(
    void *this_node,
    drv_info_t *drv_info,
    void *probe_id,
    hw_path_t *hw_path,
    struct isc_table_type *isc,
    int probe_type,
    char *name,
    char *desc)
{
    int found = NO_DEV;
    int target;

    do {
        switch (probe_type) {
            case PROBE_FIRST:
                target = 0;
                probe_type = PROBE_NEXT;
                hw_path->last_index++;
                break;
            case PROBE_NEXT:
                target = hw_path->addr[hw_path->last_index] + 1;
                break;
            case PROBE_ADDRESS:
                target = hw_path->addr[hw_path->last_index];
                break;
            default:
                found = INVAL_TGT;
        }

        if (((target < 0) || (target >= QLISP_MULTI_NUM_PORTS)) ||
            (hw_path->last_index != hw_path->first_index))
            found = INVAL_TGT;

        if (found == NO_DEV) {
            strcpy(name, QLISP_NAME);
            strcpy(desc, QLISP_DESC_BUS);
            *(int *)probe_id = isc->if_id;
            found = VALID_TGT;
            hw_path->addr[hw_path->last_index] = target;
        }

        GENLOG (probe_type, target, found, hw_path->last_index, 0);
    } while ((probe_type != PROBE_ADDRESS) && (found == NO_DEV));

    return ((found == VALID_TGT) ? PROBE_SUCCESS :
            PROBE_UNSUCCESSFUL);
}
```
qlisp Driver

This section presents the code flow of the qlisp driver. The qlisp_multi driver is a typical WSIO transport driver. The installation and initialization routines of the driver of the qlisp_multi driver are similar to the routines explained in Chapter 4, "Writing a Driver." However, qlisp driver registers an address probe function to identify all the LUNs connected to the port.

The driver should be able to scan the hardware; this requires two things:

1. To inform the WSIO that it can scan for the hardware.
2. To provide a device specific probe function to the WSIO to scan the hardware.

To inform the WSIO that the driver can scan, DRV_SCAN flag is set in the drv_info_t structure

```c
static drv_info_t qlisp_drv_info = {
    "QLISP_NAME",    /* name */
    "ext_bus",       /* class */
    DRV_SAVE_CONF | DRV_MP_SAFE | DRV_SCAN, /* flags */
    NULL,            /* b_major */
    NULL,            /* c_major */
    NULL,            /* cdio */
    NULL,            /* gio_private */
    NULL,            /* cdio_private */
};
```

Fragments from the qlisp driver’s install routine illustrate the previously mentioned steps in the driver.

```c
int qlisp_install(void)
{
    ....

    wsio_register_addr_probe(parallel_scsi_probe, "QLISP_NAME");

    wsio_install_driver(&qlisp_wsio_info);

    ....
}
```

The parallel_scsi_probe is the address probe function for the parallel SCSI drivers. It determines the next address to be probed. Registering this probe function is required to determine the targets and LUNs attached to each port. An explanation of this probe function is given in Chapter 4, “Writing a Driver.”

All routines internal to the qlisp driver are given a brief description and for some, pseudo code is provided to illustrate the code concepts.

qlisp_init_pci_cfg

Turn on the flags in the config space control registers that we care about.

qlisp_init_reset_card

- Soft reset the card
- Write into some registers
- Set SHARE_DMA64 flag
qlisp_init_attributes
Set DMA attributes like:

- Pre-fetch length (in this case, 0)
- DMA width
- Coherency protocol etc.

qlisp_init_alloc_queues
This allocates and maps memory for request & response queues.

NOTE
The “share” is a pointer to qlisp_shared_isc_t data structure.
The “iova’s” are stored in: share->dma_req_queue_p & share->dma_res_queue_p. share->req_max_iocb is set.

qlisp_init_interrupts
This allocates an interrupt object and activates the interrupt. It also enables device interrupts by writing into the corresponding device registers.

qlisp_init_firmware
- Load the firmware in case of PA-RISC
- Set firmware features.

qlisp_pdk_load_nvram_values()
- Retrieves 256 byte NVRAM region from ATMEL 93C56/66 serial EEPROM on Qlogic card.

qlisp_pdk_nvram_read()
- Programs serial EEPROM to read a 16-bit value from NVRAM.

qlisp_pdk_nvram_send_bits()
- Send bits to serial EEPROM.

qlisp_pdk_init_common()
- Initialization routine. Common to master and slave ports.

qlisp_init_common
- Save the isc in scsi_isc() maintained by the system
- Call SCSI_GET_INITIATOR_PARAMS macro to obtain SCSI parameter values
- Set the initiator ID
• Set the correct transfer period and req/ack offset
• Call \texttt{init_ifsw()}

\texttt{qlisp\_init\_ifsw}

• Allocate memory for \texttt{scsi\_ifsw} structure used by \texttt{scsi\_ctl}
• Initialize fields in the \texttt{if\_sw} structure
• Assign \texttt{isc->ifsw} with a pointer to the \texttt{ifsw} structure allocated

\texttt{qlisp\_init\_start\_queues}

• Initialize the request and response queues. These queues hold entries of \texttt{I/O Control Blocks} (IOC\texttt{B})
• Change the card state to (\texttt{SHARE\_RSP\_QUEUE} | \texttt{SHARE\_REQ\_QUEUE})

\texttt{qlisp\_init\_master}

Call routines \texttt{qlisp\_init\_pci\_cfg} through \texttt{qlisp\_init\_start\_queues} in the order set in this section of the manual.

\texttt{qlisp\_init\_slave}

• Save \texttt{slave\_isc} in \texttt{master\_lisc}
• Call \texttt{qlisp\_init\_common()}

\texttt{qlisp\_pci\_attach}

• Allocate memory for interface specific control structure
• Assign \texttt{isc->if\_isc} with the driver control structure (\texttt{lisp}) allocated
• Compare the device ID to verify whether the card can be claimed or not
• Allocate memory for the shared control structure for a dual port card
• Initialize PCI config space values
• Attach the \texttt{init} routines to the list of \texttt{init} routines
• Claim the card instance

\texttt{qlisp\_mailbox\_cmd}

Mailbox commands are not done in performance path, and usually only during the driver initialization time. Send a mailbox command. (details later).

\texttt{qlisp\_build\_cmd} (and \texttt{cmd 64 for \_LP64\_})

• Set the \texttt{iocb} fields
• Get the \texttt{scsi\_lun} structure associated with the \texttt{scb}
• Get the \texttt{scsi\_tgt} structure associated with the \texttt{lp}
• Set the `target_id` and `lun_id` fields of the command
• Set the control flags of the command
• Copy the cdb from `scb` into `iocb`
• Take the DMA parameters and sync the memory

**qlisp_iocb_cnt**
Calculate number of DMA command entries from DMA scatter/gather list.

**qlisp_get_req_iocb**
• Checks request queue for enough available `iocb`'s for command
• Updates `share->req_free` and `share->req_in`

**qlisp_if_start**
Interface entry point called by SCSI services where I/O's are started. This entry point is set in the ID `ifsw` structure in `qlisp_init_ifsw()`.

• Get the ID specific control structure from `isc->if_isc`
• Get `scsi_bus` structure from `isc->if_drv_data` (this field is set by the `scsi_services`)
• Get the shared control structure from the `lisc`

```c
do {
    Acquire scsi bus lock.

    **NOTE**: Once an I/O has been put on the select queue it is off limits to services until the interface driver calls scsi_cbfn with the I/O.

    Get a bp waiting for interface driver
    (Call scsi_dequeue, a SCSI service to get a bp from the select queue.

    Release scsi bus lock

    if bp is not NULL
        Get SCSI control block.
        Get a pointer to driver’s local scb structure.
        Get a pointer to driver’s local bus structure.

        if there are bytes to transfer (bp->b_bcount != 0)
            Perform mapping for dma pages
            Call get_iocb_cnt (cnt64 for LP64)
            Get required iocbs (this may not always succeed)
            if didn’t get reqd iocbs, then
                if the request can’t be satisfied,
                    Set the scb status to invalid
                    (scb->cdb_status = SCTL_INVALID_REQUEST)
                    and call scsi_cbfn() (SCSI service)
                else (if request can be satisfied later)
                    Requeue the bp (scsi_enqueue(&busp->select_q, bp, HEAD))
            /* everything is fine */
```
Call build cmd (cmd64 for LP64)
Increase the activecnt in the interface driver local bus structure.

/* scb->tag is used as an index into the NexusTable. The scb->tag is assigned immediately following scb association in scsi_start and is retained for all retries. */
Save the interface driver local scb in interface driver local bus structure.
Inform the device by moving in the req_in pointer.
}

} until bp != NULL

quilsp_fast_complete

Used by performance status completion mechanism when status is good and only handle is returned.

Get local scb from the local bus structure’s NexusTable field. Get bp from local scb.
Get scb from local scb.

if bp->b_count != 0, cleanup the DMA

The next few fields indicate the result of an I/O attempt. The appropriate fields are set by the interface driver prior to calling scsi_cbfn().
Set the following fields (as reqd.)
scb->cdb_status
scb->data_resid - number of data bytes that were transferred in response to scb->cdb, i.e.
number of bytes transferred = bp->b_bcount - scb->data_resid
scb->sense_status
scb->sense_bytes
scb->sense_data

Decrement activecnt of the outstanding I/O requests
Set the local scb structure in the NexusTable to NULL
Put the bp in the free list (bp->av_forw)
Assign the bp to the list of bps waiting for scsi_cbfn.

quilsp_post_reset_delay

• Change the lisc state to ~LISC_RESET
• Call quilsp_if_start() if reset completes.

quilsp_call_cbfns

Invoke callback functions for chain of bp’s.

Get the scsi_bus structure from the isc->if_drv_ata

while there is a bp in the cbfns list
Get scb from the bp
Set cbfns to bp->av_forw
Call scsi_cbfn()
done with while
qlisp_process_rsp

Loop through response blocks and process.

Get scsi_bus structure from isc->if_drv_data.
Get the local bus structure from the scsi_bus structure.

while queue is not empty

Get the iocb from the response queue.
Move the rsp_out to the next one with a circular wrap-up if reqd.

NOTE: Performance status completion for more than 5 I/O’s at one time.

Get the number of I/O handles in done_cnt.

while the done_cnt is not 0

Call qlisp_fast_complete on the corresponding handle

done with while

Signal chip that responses have been processed by moving out pointer

qlisp_isr

The qlisp_isr is the driver interrupt routine. It handles interrupts on the Qlisp 12160 card.

Acquire the driver spinlock.
Get the driver local control structure (lisc).

Read the device register.

while reg val is SEMA_LOCK

Check the register for command status
if the command is completed,
   Read the other mailbox commands
   if the command is completed with no errors,
      just break.
   else print an error message and break.
else /* An async event */
   if the mb_status is one of the port RIO_POST statuses,
      then
      Get the number of I/Os done
      Call qlisp_fast_complete() on each of them
      else check for possible errors

done with while

if the mailbox command is not complete, get the rsp_in.
else
   Check the share state if it is in SHARE_RSP_QUEUE and if the response Q is not empty.
   Call qlisp_process_rsp().

if there was a reset earlier, call qlisp_post_reset_delay()
Check for other share->state status values.

if there are any bps whose i/o is completed,
Call qlisp_call_cbfns().
qlisp_if_lun_open
LUN open interface entry point called by SCSI services.

qlisp_if_abort
Interface entry point called by SCSI services where an I/O is aborted.

- From the buf structure, get the device number, scsi_bus structure, scsi_target structure and scsi_lun structure.
- Submit a mailbox command to abort device LUN.
- If the card is not in reset state, call qlisp_send_marker.

qlisp_if_bdr
Interface entry point called by SCSI services where a device is reset.

- Submit a mailbox command BDR Target
- If the card is not in reset state, call qlisp_send_marker

qlisp_reset_bus
Causes a SCSI bus reset with a mp_timeout delay before I/O's are scheduled.

- Set lisc state to LISC_RESET
- Submit a mailbox command to Reset SCSI bus
- Call qlisp_post_reset_delay after 3 secs. (via a time-out)

qlisp_if_reset_bus
Interface entry point called by SCSI services where the SCSI bus is reset.

- Get scsi_bus structure from the device number
- Get isc from the scsi_bus structure
- Call qlisp_reset_bus

qlisp_send_marker
This routine is used to bring the device to a known state. This re-enables the adapter, target or LUN request queue.
This chapter presents routines and conceptual material specifically for drivers of SCSI devices. Chapter 4, "Writing a Driver," describes the general configuration and entry point driver routines, such as `driver_open` and `driver_write`. When writing a SCSI driver, provide routines from both Chapter 4, "Writing a Driver," and this chapter.

The *HP-UX 11i v1 Driver Development Reference Guide* describes the SCSI Services routines.

SCSI devices can be controlled in two ways, through Kernel drivers and using the `scsi-ctl` driver, both are supported by the SCSI Services routines. Kernel drivers, following the `scsi_disk` model, are the traditional method and are described in this chapter and in the `scsi_ctl` (7) manpage. Many SCSI devices do not need a special driver. Instead, user programs pass ioctl commands to the pass-through driver, `scsi_ctl`. The pass-through driver is described in *HP-UX 11i v1 Driver Development Reference Guide* manpage `scsi_ctl` (7). Refer to `mt` (7) manpage for discussion of ioctl, device files and driver behavior for tape devices.

The following sections provide the suggested steps for developing a SCSI driver:

- “Step 1: Include Header Files”
- “Step 2: Set Up Structures”
- “Step 3: Create the `driver_install` Routine”
- “Step 4: Create the `driver_dev_init()` Routine”
- “Step 5: Analyze Multiprocessor Implications”
- “Step 6: Create the Entry Point Routines”
- “Step 7: Error Handling”
- “Step 8: Underlying Routines”

The examples in this chapter assume the name of the driver is `mydriver` and are following the routine naming conventions described in Chapter 4, "Writing a Driver."
Step 1: Include Header Files

See reference pages for each kernel call and data structure the driver uses to find which headers the driver requires.

NOTE
HP recommends that header files the driver doesn't need are not included. It increases compile time and the likelihood of encountering portability problems. It is not recommended.

General Header Files

/usr/include/sys/buf.h    I/O buf structure, buf.
/usr/include/sys/errno.h  Defines errors returned to applications.
/usr/include/sys/file.h  Defines open flags
/usr/include/sys/io.h    isc table structure.
/usr/include/sys/conf.h  drv_ops_t structure
/usr/include/sys/kern_svcs.h Kernel services for synchronization
/usr/include/sys/malloc.h Necessary for acquiring and releasing memory.
/usr/include/sys/wsio.h  WSIO context data and macro definitions.

Header Files for SCSI Drivers

/usr/include/sys/scsi.h   SCSI-specific data definitions and ioctl commands.
/usr/include/sys/scsi_ctl.h SCSI subsystem data and macro definitions.

Header Files for Device Classes

In addition to the header file created for the specific driver, the driver may need other device class specific files.

/usr/include/sys/diskio.h Data definitions for disk ioctl commands (DIOC_xxx). Includes /usr/include/sys/types.h and /usr/include/sys/ioctl.h.
/usr/include/sys/floppy.h Data definitions for floppy ioctl commands.
/usr/include/sys/mtio.h  Data definitions for magnetic tape ioctl commands.
Step 2: Set Up Structures

Depending on the characteristics of the driver, it can be set up as a character driver, a block driver, or (as in the case of disk drivers) both.

NOTE

SCSI Services use the locking facilities regardless of whether the driver operates on an Multiprocessor (MP) platform. All drivers using SCSI Services must use the provided data protection routines. It is essential that the C_ALLCLOSES and C_MGR_IS_MP flags in the drv_ops_t structure and the DRV_MP_SAFE flag in the drv_info_t structure are included. See “Step 5: Analyze Multiprocessor Implications” for more information.

Determine the driver’s name and device class, and put this information in the appropriate structures. (See Chapter 4, “Writing a Driver,” for information about these data structures.)

First, declare the driver’s routines that can be called by the kernel. These are used in the following structure:

```c
int mydriver_open();
int mydriver_close();
int mydriver_strategy();
int mydriver_psize();
int mydriver_read();
int mydriver_write();
int mydriver_ioctl();
```

The drv_ops_t structure specifies the “external” driver routines to the kernel. The C_ALLCLOSES and C_MGR_IS_MP flags are required by SCSI Services. See Chapter 4, “Writing a Driver,” for further details.

```c
static drv_ops_t mydriver_ops =
{
     mydriver_open,
     mydriver_close,
     mydriver_strategy,
     NULL,
     mydriver_psize,
     NULL,
     mydriver_read,
     mydriver_write,
     mydriver_ioctl,
     NULL,
     NULL,
     NULL,
     NULL,
     NULL,
     NULL,
     C_ALLCLOSES | C_MGR_IS_MP
};
```
The `drv_info_t` structure specifies the driver’s name (*mydriver*) and class (*disk*). Flags define the driver type. The `DRV_MP_SAFE` flag is required by SCSI Services. See *Chapter 4, “Writing a Driver,”* for further details.

```c
static drv_info_t mydriver_info =
{
    "mydriver",
    "disk",
    DRV_CHAR | DRV_BLOCK | DRV_SAVE_CONF | DRV_MP_SAFE,
    -1,
    -1,
    NULL,
    NULL,
    NULL,
};
```

The `wsio_drv_data_t` structure specifies additional information for the WSIO CDIO. The first field should be “scsi <class>” for SCSI class drivers (for example, “scsi_disk” for the disk class) and “scsi” for SCSI interface drivers. See *Chapter 4, “Writing a Driver,”* for further details.

```c
static wsio_drv_data_t mydriver_data =
{
    "scsi_disk",    /* "scsi_tape" for tape devices */
    T_DEVICE,
    DRV_CONVERGED,
    NULL,
    NULL,
};
```

The `wsio_drv_info_t` structure ties the preceding three together. See *Chapter 4, “Writing a Driver,”* for further details.

```c
static wsio_drv_info_t mydriver_wsio_info =
{
    &mydriver_info,
    &mydriver_ops,
    &mydriver_data,
    WSIO_DRV_CURRENT_VERSION
};
```
Step 3: Create the driver_install Routine

The `driver_install` routine causes the information that was previously created to be installed into the I/O subsystem, specifically into the WSIO CDIO. External definition of `dev_init` is in `<sys/wsio.h>.

```c
int (*mydriver_saved_dev_init)();

int mydriver_install()
{
    /* The content of dev_init (a function pointer) needs to be
       saved here and then must be called by mydriver_dev_init().
       This is done to create a linked list of driver_init routines.
       It is the responsibility of the driver’s init routine to call
       the init routine of the next driver using this method.
    */
    mydriver_saved_dev_init = dev_init;
    dev_init = mydriver_dev_init;

    /* register driver with WSIO and return any error */
    return(wsio_install_driver(&mydriver_wsio_info));
}
```
Step 4: Create the driver_dev_init() Routine

Specify the `driver_dev_init` routine from the `driver_install()` routine. The `driver_dev_init` routine calls `scsi_ddsw_init()`, which initializes some fields in the SCSI driver's device switch table (`scsi_ddsw`). This table is independent of the kernel's device switch tables.

```c
mydriver_dev_init()
{
    /*
     * Initialize mydriver_ddsw.blk_major and
     * mydriver_ddsw.raw_major.
     */
    scsi_ddsw_init(mydriver_open, &mydriver_ddsw);

    (*mydriver_saved_dev_init)();
}
```

Setting up the Device Switch Table (scsi_ddsw)

To use SCSI Services effectively, a SCSI driver must define its `scsi_ddsw` device switch structure. This structure contains pointers to special dd routines, some of which are executed indirectly by the standard driver routines, such as `driver_read`. The structure is passed to SCSI Services routines from the `driver_open` routine, which calls the `scsi_lun_open()` SCSI Services routine.

SCSI Services has been set up to control the housekeeping and other processing in the SCSI interface. The standard driver routines restrict their operation to calling the appropriate SCSI Services routine, as shown in the examples in “Step 6: Create the Entry Point Routines”. Special processing and customization should all be handled in the special dd routines.

For a summary of SCSI Services, see the “SCSI Services Summary” section. For more detailed information, see the HP-UX 11i v1 Driver Development Reference Guide.

The `scsi_ddsw_init()` should be used only for backward compatibility. Do not use it for new drivers.

```c
struct scsi_ddsw
{
    u_char blk_major;
    u_char raw_major;
    int dd_lun_size;
    int (*dd_open)();
    void (*dd_close)();
    int (*dd_strategy)();
    int (*dd_read)();
    int (*dd_write)();
    int (*dd_ioctl)();
    struct buf (*dd_start)();
    int (*dd_done)();
    int (*dd_pass_thru_okay)();
    int (*dd_pass_thru_done)();
    int (*dd_ioctl_okay)();
    struct status_action *dd_status_list;
    int dd_status_cnt;
    ubit32 dd_flags;
    wsio_drv_info_t *wsio_drv;
};
```
The entries are described as follows:

- **blk_major, raw_major**: Block and character major numbers; specify them as NODEV. They are initialized by `scsi_ddsw_init()` when it is called from the `driver_dev_init()` routine.

- **dd_lun_size**: The number of bytes to be allocated and attached to the `scsi_lun` structure, see Chapter 13, “Writing SCSI Device Drivers,” for the class driver's private data structure. This data structure will be allocated at the first open and deallocated at the last one.

- **dd_open(), dd_close(), dd_strategy(), dd_read(), dd_write(), dd_ioctl()**: Pointers to underlying driver specific routines. When the corresponding `driver_routine` is called by the kernel and transfers control to SCSI Services, SCSI Services performs certain overhead operations and calls these routines for driver specific operations.

- **dd_start()**: Driver specific start routine.

- **dd_done()**: Driver specific post I/O processing.

- **dd_pass_thru_okay()**: Driver specific control of pass through I/O.

- **dd_pass_thru_done()**: Driver specific notation of pass through I/O.

- **dd_ioctl_okay()**: Disallow `ioctl` commands through the pass through driver.

- **dd_status_list**: Action list for handling various status/error conditions.

- **dd_status_cnt**: Number of entries in `dd_status_list`.

- **dd_flags**: Flag bits, currently only DD_DD defined.

- **wsio_drv**: Pointer to standard WSIO driver info data structure.

Here is an example of an initialized declaration of the `scsi_ddsw`:

```c
int mydriver_dd_open(); /* dd_open */
void mydriver_dd_close(); /* dd_close */
int mydriver_dd_strategy(); /* dd_strategy */
int mydriver_dd_read(); /* dd_read */
int mydriver_dd_write(); /* dd_write */
int mydriver_dd_ioctl(); /* dd_ioctl */
struct buf mydriver_dd_start(); /* dd_start */
int mydriver_dd_done(); /* dd_done */
int mydriver_dd_pass_thru_okay(); /* dd_pass_thru_okay */
int mydriver_dd_pass_thru_done(); /* dd_pass_thru_done */
int mydriver_dd_ioctl_okay(); /* dd_ioctl */
```
The following example shows the `scsi_ddsw` structure. Specify NULL for routines that are not defined (not using). The first two fields specify the block and character major numbers; they are filled in by the call in `driver_dev_init()` to the SCSI Services routine `scsi_ddsw_init()`. The last field points to the `wsio_drv_info_t` structure that was defined in “Step 2: Set Up Structures”. The first name in each comment is the field name of the `scsi_ddsw` structure element.

```c
struct scsi_ddsw mydriver_ddsw =
{
    NODEV,    /* blk_major - mydriver_dev_init sets */
    NODEV,    /* raw_major - mydriver_dev_init sets */
    sizeof(struct mydriver_lun), /* dd_lun_size */
    mydriver_dd_open,  /* dd_open */
    mydriver_dd_close, /* dd_close */
    mydriver_dd_strategy, /* dd_strategy */
    NULL,      /* dd_read */
    NULL,      /* dd_write */
    mydriver_dd_ioctl, /* dd_ioctl */
    mydriver_dd_start, /* dd_start */
    mydriver_dd_done, /* dd_done */
    mydriver_dd_pass_thru_okay, /* dd_pass_thru_okay */
    mydriver_dd_pass_thru_done, /* dd_pass_thru_done */
    mydriver_dd_ioctl_okay, /* dd_ioctl_okay */
    mydriver_dd_status_list, /* dd_status_list */
    sizeof(mydriver_dd_status_list), /* dd_status_cnt */
    mydriver_dd_flags,  /* dd_flag bits DD_DDG */
    &mydriver_wsio_info /* For Diagnostics Logging; NULL means errors print in dmesg */
};
```
Step 5: Analyze Multiprocessor Implications

Make the class driver **Multiprocessor** (MP) safe, regardless of whether it is to operate an MP platform or not. SCSI Services make use of the kernel's locking facilities, so all drivers that use SCSI Services must use the synchronization routines the kernel provides.

Drivers must do the following:

- Set the `C_MGR_IS_MP` flag in the `d_flags` field of the driver's `drv_ops_t` structure.
- Set the `DRV_MP_SAFE` flag in the `flags` field of the `drv_info_t` structure.
- Use the driver semaphore, driver lock, **Logical Unit Number** (LUN) lock, and target lock as necessary to provide MP protection. Refer to the defines and structures in `/usr/include/sys/scs_ctl.h` for details. This is the largest task, and involves looking at the code and determining whether there are data references that must be protected and which locks and semaphores must be used to protect the references.
- Build a kernel with the driver.
- Test the driver on a **single processor** (UP) system and a **Multiprocessor** (MP) system with a debug kernel if available.
- See Chapter 3, “Multiprocessing,” and the “Synchronization in SCSI Drivers” section of this chapter for additional details.
Step 6: Create the Entry Point Routines

SCSI services provides helper routines to support the various driver entry point routines, which must be called in the driver entry point path.

For example, on open, the SCSI service routine, `scsi_lun_open()` must be called to instantiate the `scsi_lun` structure and other SCSI services data structure. `scsi_lun_open()`, in turn will call back to the driver `dd_open` routine, if not `NULL`, to perform any subsequent driver-specific actions if needed. The `scsi_lun_open()` routine will simply initialize the inquiry data if no `dd_open` was specified. Similarly, the other entry points have corresponding SCSI services functions (as shown in the following examples) which will call back to the corresponding `dd_xxx` routines if not `NULL`.

The `scsi_read()` and `scsi_write()` routines will call `physio()` if no corresponding `dd_read/dd_write` routines specified (NULL) to complete the request down the stack.

If `dd_strategy` is `NULL`, `scsi_strategy` will enqueue the `bp` on the `scsi_lun scb` queue for processing down the stack. The `dd_strategy` routine needs to enqueue the passed `bp` on a queue which can be retrieved later during processing of the I/O down the stack.

If `dd_strategy` is provided, one should also provide a `dd_start` routine to dequeue the I/O requests, enqueued by the `dd_strategy` routine, for processing down the stack.

The `dd_ioctl` routine is needed for implementing driver specific `ioctl()` commands.

Examples of each of the entry point routines:

```c

mydriver_open(dev_t dev, int oflags)
{
    ......
    return (scsi_lun_open(dev, &mydriver_ddsw, oflags));
    ......
}

mydriver_close(dev_t dev)
{
    return scsi_lun_close(dev);
}

mydriver_read(dev_t dev, struct uio *uio)
{
    return scsi_read(dev, uio);
}

mydriver_write(dev_t dev, struct uio *uio)
{
    return scsi_write(dev, uio);
}
```

```c

Chapter 13
```
**driver_strategy()**

The `driver_strategy()` routine does not return anything. It records errors in `bp->b_error`.

```c
mydriver_strategy(struct buf *bp)
{
    scsi_strategy(bp);
}
```

**driver_psize()**

This example assumes that `driver_psize()` is never called when the device is closed. Note the use of the SCSI Services `m_scsi_lun()` function. This routine returns the number of 1K blocks on the device.

```c
mydriver_psize(dev_t dev)
{
    struct scsi_lun  *lp = m_scsi_lun(dev);
    struct mydriver_lun  *llp = lp->dd_lun;
    int rshift, nblks, size;

    /* Example private fields in llp */
    nblks = llp->nblks;
    rshift = llp->devb_lshift;

    /* Example usage of the files */
    size = rshift > 0 ? nblks >> rshift : nblks << -rshift;

    return size;
}
```

**driver_ioctl()**

```c
mydriver_ioctl(dev_t dev, int cmd, int *data, int flags)
{
    return scsi_ioctl(dev, cmd, data, flags);
}
```
Step 7: Error Handling

The class driver specifies what actions it wants to take on various I/O status conditions via its `dd_status_list` field in `scsi_ddsw` structure. If `dd_status_list` is not specified (`dd_status_cnt` equals 0), then the failed I/O will be completed immediately without any retry.

This status list specifies the retry policy (whether to retry and how long to delay between retrying), an action routine to be called to handle the condition, and an `errno` value to be used on ultimate completion of the I/O in case of failure. The status list keys off of the `cdb_status` field in the `scb` data structure, whose low-order byte is the SCSI status byte from the device and whose high-order byte encodes additional software on the hardware condition.

The SCSI service routine, `scsi_cmdx()`, also allows for a specific status list to be passed in as a parameter.

Data structure definitions for `struct status_action` and `struct sense_action` are available in `<sys/scsi_ctl.h>`.

Structure for each element in the array of status/action pairs attached to `ddsw->dd_status_action_list` and traversed by SCSI services follows.

```c
struct status_action
{
    int status;
    int (*action)();
    intptr_t arg1, arg2, arg3, arg4;
};
```

- **status**: The value to be compared against the `cdb_status` field in `scb` data structure.
- **action**: The action routine to be called if `cdb` status matches the value specified in the “status” field.
- **arg1, arg2, arg3, arg4**: The arguments to be passed to the action routine when called. The first argument to action routine is always the pointer to the `buf (bp)`. The values specified in arg1, arg2, arg3, arg4 would be passed as 2nd, 3rd, 4th and 5th arguments.

For `cdb_status` of CHECK_CONDITION, an additional status list of sense data from the device can be provided to match the sense key, additional sense code, and additional sense code qualifier, to take appropriate actions. The following data structure is used to provide the action list for `cdb_status` of CHECK_CONDITION.

```c
struct sense_action
{
    int status; /* status of the automatic request sense */
    short error_code;
    short sense_key;
    short sense_code;
    short sense_qualifier;
    int (*action)();
    int arg1, arg2, arg3, arg4;
};
```

- **status**: Request sense status byte. The value to be compared against the `sense_status` field in `scb` data structure.
- **error_code**: The value which is to be compared against the “Error code” byte of the request sense data.
sense_key The value which is to be compared against the “sense key” field of the request sense data.
sense_code The value which is to be compared against the “Additional sense code” field of the request sense data.
sense_qualifier The value which is to be compared against the “Additional sense code qualifier” field of the request sense data.
action The action routine to be called if sense_key, sense_code, and sense_qualifier fields match with the request sense data obtained from the device.
arg1, arg2, arg3, arg4 The arguments to be passed to the action routine when called. Please note the first argument to action routine is always the pointer to the buf (bp).

For cdb_status of S_CHECK_CONDITION, the action routine can be SCSI service function, scsi_sense_action(), with the parameters specifying a driver-specific “sense action list”.

In the mydriver_status_list structure, argument one is used as flags, argument two as error no, and argument three is milliseconds between retries (msecs). See the following examples:

```
struct sense_action mydriver_sense_list[] = {
   { S_GOOD, S_CURRENT_ERROR, S_RECOVERRED_ERROR,
     SA_ANY, SA_ANY, mydriver_check_residue, SA_DONE |
     SA_LOG_IT_NEVER, 0 },
   { SA_ANY, SA_ANY, SA_ANY, SA_ANY, SA_ANY, scsi_action,
     SA_DONE + SA_LOG_IT_NEVER, EIO }
};
```

```
struct status_action mydriver_status_list[] = {
   { S_GOOD, scsi_action, SA_DONE +
     SA_LOG_IT_NEVER, 0 },
   { S_CHECK_CONDITION, scsi_sense_action, (int)
     mydriver_sense_list, sizeof(mydriver_sense_list) |
     sizeof(mydriver_sense_list[0]) },
   { S_CONDITION_MET, scsi_action, SA_DONE +
     SA_LOG_IT_NEVER, 0 },
   { S_CONDITION_MET, scsi_action, SA_DONE +
     SA_LOG_IT_NEVER, 0 },
   { S_I_CONDITION_MET, scsi_action, SA_DONE +
     SA_LOG_IT_NEVER, 0 },
   { SA_ANY, scsi_action, SA_DONE + SA_LOG_IT_ALWAYS, EIO }
};
```

The driver can specify its own routines for handling errors and can break down errors for more granularity. Access the Pass-Thru Driver status using the driver’s dd_pass_thru_done() routine, described in “Step 8: Underlying Routines”.

For tape specific error handling, the head position data returned in MTIOCGET must reflect any changes in head position indicated by sense data. When a Device or Bus reset is done, an error with EPERM is returned on any subsequent media motion commands (for example, read, write, space or write file marker) until the user closes or reopens the device.
Step 8: Underlying Routines

This is where the driver can be as complex as desired, or as the device requires. The `scsi_lun_open()` routine ensures that the bus, target, and LUN of the driver’s device are open and able to handle I/O. Specific requirements for the device itself should be addressed in the driver’s `ddsw->dd_open()` routine. The same principle applies for `close`, `read`, `write`, and so on.

The call graph in Figure 13-1, “Call Graph of SCSI Routines and Services,” shows how these underlying routines and SCSI services call each other. For a summary list of SCSI Services, see “SCSI Services Summary”. Detailed information on SCSI Services is provided in the *HP-UX 11i v1 Driver Development Reference Guide*.

Figure 13-1 Call Graph of SCSI Routines and Services
**dd_close**

The `dd_close()` SCSI function, is used to provide driver specific processing during close is provided by the driver writer. It can have any unique name. Pass the name to SCSI Services by specifying it in the `dd_close` field of the `scsi_ddsw` structure.

If this routine is defined in the `scsi_ddsw` structure, it is called to perform the actual device close processing. For example, for the `scsi_disk` driver, the `sd_close()` function performs the Test Unit Ready and Allow Media Removal commands.

The tape driver supports the following device file attributes:

- No rewind-on-close: Do not rewind media during driver close routine.
- ATT style close: If the device is opened as read only, the driver will space the next file on close if not already at Beginning of Tape (BOT) or End of File (EOF). If the user has written End of Data (EOD) but has not explicitly written an EOD mark, write an EOD in the driver close routine.

Refer to `mt (7)` manpage for more device file attributes supported by a tape driver.

**Conditions**

The `dd_close()` is called from `scsi_lun_close()` in a process context. The open or close LUN semaphore is held when the `dd_close()` function is called. `dd_close()` is not called from within a critical section; it may block.

**Declaration**

```c
void dd_close (dev_t dev);
```

**Parameters**

- `dev` The device number.

**Return Values**

`dd_close()` does not return a value.

**Example**

```c
#include <sys/scsi_ctl.h>
#define ST_GEOM_LOCKED 0x00000002

void mydriver_dd_close(dev_t dev);
{
    struct scsi_lun *lp = m_scsi_lun(dev);
    struct mydriver_lun *llp = lp->dd_lun;

    /*
     * dd_blk_open_cnt() is defined in <sys/scsi_ctl.h>
     * It returns the count of current block opens
     */
    if (dd_blk_open_cnt(lp) == 1) {
        scsi_lun_lock(lp);
        llp->state &= ~ST_GEOM_LOCKED;
        scsi_lun_unlock(lp);
    }
}
```
**dd_ioctl**

The `dd_ioctl()` routine is provided by the driver writer. It can have any unique name. Pass the name to SCSI Services by specifying it in the `dd_ioctl` field of the `scsi_ddsw` structure.

If this routine exists in the `scsi_ddsw` structure it is called by `scsi_ioctl()` if the ioctl command remains unsatisfied by the choices provided within that SCSI Services procedure. If `dd_ioctl()` does not exist when called, `scsi_ioctl()` returns an error.

Examine the ioctl commands provided by SCSI Services in `scsi_ioctl()` and implement any additional commands needed in the `dd_ioctl()` routine.

If `dd_ioctl()` and `dd_open()` are implemented, some of the more specialized features of SCSI Services may be useful, refer to the following list:

- `scsi_cmd()`
- `scsi_init_inquiry_data()`
- `scsi_mode_sense()`
- `scsi_mode_fix()`
- `scsi_mode_select()`
- `scsi_wr_protect()`

*Table 13-1, “IOCTL Usage by LVM Summary,” shows a list of IOCTLs that the LVM uses to interface with the class driver.*

### Table 13-1 IOCTL Usage by LVM Summary

<table>
<thead>
<tr>
<th>IOCTL</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIOC_RSTCLR</td>
<td>The LVM uses each of these ioctl's on all bus paths.</td>
</tr>
<tr>
<td>SIOC_RESET_BUS</td>
<td></td>
</tr>
<tr>
<td>DIOC_DESCRIBE</td>
<td>The LVM uses each of these ioctl's for device accessibility, device type, block size, and control of immediate reporting, and setting the pftimeout interval on a per LUN path basis.</td>
</tr>
<tr>
<td>DIOC_BLKLIST_REMAP</td>
<td></td>
</tr>
<tr>
<td>DIOC_SET_PF</td>
<td></td>
</tr>
<tr>
<td>TIMEOUTSIOC_GET_IR</td>
<td></td>
</tr>
<tr>
<td>SIOC_INQUIRY</td>
<td></td>
</tr>
<tr>
<td>SIOC_IO</td>
<td></td>
</tr>
<tr>
<td>SIOC_SET_IR</td>
<td></td>
</tr>
</tbody>
</table>

The tape driver must support the ioctl's as described in `mt (7)` manpage in order to interoperate with HP-UX commands, such as `tar`, `dd`, `cpio`, `mt`, `st`, `mc`, `fbackup` and `frecover`. The ioctl specific to the tape driver is given in *Table 13-2, “IOCTL Usage by Tape Driver.”*
Table 13-2  IOCTL Usage by Tape Driver

<table>
<thead>
<tr>
<th>IOCTL</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTIOCGET</td>
<td>The fbbackup and frecover commands require that the mt_type field in the MTIOCGET data be filled in with one of the following tape types; MT_ISLTO, MT_ISDLT, MT_IS3480, MT_IS3590, MT_ISQIC, MT_ISHPFIB_REEL, MT_ISSCSI_REEL, MT_ISSDDS, MT_ISSMM.</td>
</tr>
<tr>
<td>MTIOCTOP</td>
<td>This ioctl is for media motion operations (for example, space, write file mark, rewind, and so on.) Refer to mt (7) manpage for detailed description on operations. The header file &lt;sys/mtio.h&gt; defines the operations.</td>
</tr>
<tr>
<td>SIOC_SET_BLOCK_SIZE</td>
<td>This ioctl is used for block size settings.</td>
</tr>
<tr>
<td>SIOC_GET_BLOCK_SIZE</td>
<td>This ioctl is used for reading the set block size.</td>
</tr>
</tbody>
</table>

Conditions

The dd_ioctl() is called from scsi_ioctl() in a process context. It is not called from within a critical section; it may block.

Declaration

```c
int dd_ioctl (  
    dev_t dev,  
    int cmd,  
    caddr_t data,  
    int flags  
);  
```

Parameters

- `cmd` The command word.
- `data` Pointer to the commands arguments.
- `dev` The device number.
- `flags` The file access flags.

Return Values

The dd_ioctl() is expected to return the following values:

- `0` Successful completion.
- `<>0` Error. Value is expected to be an `errno`. 
Example

```c
#include <sys/scsi.h>
#include <sys/scsi_ctl.h>
mydriver_dd_ioctl (dev_t dev,
    int cmd,
    int *data,
    int flags
)
{
    struct scsi_lun *lp = m_scsi_lun(dev);
    struct mydriver_lun *llp = lp->dd_lun;
    struct scsi_tgt *tp = lp->tgt;
    struct scsi_bus *busp = tp->bus;
    struct inquiry_2 *inq = &lp->inquiry_data.inq2;
    int i;

    switch (cmd)
    {
        case DIOC_FORMAT:
        case SIOC_FORMAT:
            if (!(flags & FWRITE) && !suser())
                return EACCES;
            if (!(llp->state & L_EXCLUSIVE || tp->state & T_EXCLUSIVE || busp->state & B_EXCLUSIVE)) {
                return EBUSY;
            }
            return mydriver_format(dev,((struct sioc_format *)
                data)->interleave);
        case SIOC_GET_IR:
            return mydriver_wce(dev, SIOC_GET_IR, data);
        case SIOC_SET_IR:
            if (!(flags & FWRITE) && !suser())
                return EACCES;
            if (*data & ~0x1)
                return EINVAL;
            return mydriver_wce(dev, SIOC_SET_IR, data);
        ...
        ...
        default:
            return EINVAL;
    }
}
```

**dd_ioctl_okay**

The `dd_ioctl_okay()` SCSI function is provided by the driver writer. It can have any unique name. Pass the name to SCSI Services by specifying it in the `dd_ioctl_okay` field of the `scsi_ddsw` structure.

The `dd_ioctl_okay()` disallows all `ioctl` commands through the pass-through driver that are not explicitly allowed by any non-pass-through driver that has the device open concurrently.

**Conditions**

`dd_ioctl_okay()` is called from `sctl_ioctl()` in a process context. It is called within a critical section; it may not block.
NOTE

Allow SIOC_INQUIRY for the pass-through driver at all times. SIOC_INQUIRY is allowed by default (if there is no dd_ioctl_okay() routine). SIOC_INQUIRY is always allowed if it will not result in I/O (lp->inquiry_sz > 0), because it does not affect the non-pass-through class driver in any way.

Declaration

```c
int dd_ioctl_okay (  
    dev_t   dev,  
    int     cmd,  
    caddr_t data,  
    int     flags
);
```

Parameters

- `cmd` The command word.
- `data` Pointer to the commands arguments.
- `dev` The device number.
- `flags` The file access flags.

Return Values

dd_ioctl_okay() is expected to return the following values:

- `PT_OKAY` Successful completion.
- `0` Error

Examples

```c
#include <sys/scsi_ctl.h>

mydriver_dd_ioctl_okay (  
    dev_t   dev,  
    int     cmd,  
    void    *data,  
    int     flags
)
{
    return PT_OKAY;
}
```

**dd_open**

The dd_open() SCSI function is provided by the driver writer. It can have any unique name. Pass the name to SCSI Services by specifying it in the dd_open field of the scsi_ddsw structure.

If this routine exists in the scsi_ddsw structure, it is called to perform the actual device “open” processing.

As an example, the scsi_disk driver’s dd_open routine, sd_open(), calls disksort_init_queue() for the LUN’s lun_disk_queue. It calls scsi_init_inquiry_data() to set the target state for SDTR and WDTR and send the Start Unit, Test Unit Ready, Prevent Media Removal, and Read Capacity commands, if appropriate, for the type of disk being opened.
This routine can be as complicated as needed to ensure the device is properly open the first time (ensured by checking `dd_open_cnt`). Calling the SCSI Service `scsi_init_inquiry_data()` is reasonable, as is performing Test Unit Ready. Changing state in the `scsi_lun` or target structures requires protection.

The tape driver `O_NDELAY` flag indicates that the device file can be opened even if there is no media loaded.

**Conditions**

The `dd_open()` is called from `scsi_lun_open()` in a process context. The open/close `lun_semaphore` is held when `dd_open()` is called. `dd_open()` is not called within a critical section; it may block.

**Declaration**

```c
dd_open(
    dev_t dev,
    int oflags
);
```

**Parameters**

- `dev` The device number.
- `oflags` The flags passed in the open call.

**Return Values**

`dd_open()` is expected to return the following values:

- `0` Successful completion.
- `<0` Error. The value is expected to be an `errno` value.

**Examples**

```c
#include <sys/scsi_ctl.h>

mydriver_dd_open(dev, oflags)
{
    struct scsi_lun *lp = m_scsi_lun(dev);
    struct mydriver_lun *llp = lp->dd_lun;
    struct scsi_tgt *tp = lp->tgt;
    struct inquiry_2 *inq = &lp->inquiry_data.inq2;
    struct capacity cap;
    u_char cdb[16];
    struct sense_hdr *hd;
    struct block_desc *bd;
    struct caching_page *c_pd;
    struct error_recovery *e_pd;
    int ret_size, bpb, error, x;

    /*
     * Only first opens are interesting.
     * `dd_open_cnt()` is defined in <sys/scsi_ctl.h>.
     * It returns the count of current opens.
     */
    if (dd_open_cnt(lp) > 1)
        return 0;
    ...
    /*
     * Inquiry.
     */
```
Call the routine provided by services to do any necessary synchronization with the pass-through driver. Success here does not imply that there is no more pending sense data. In fact, the SCSI-2 standard encourages devices not to give Check Condition status on Inquiry, but to defer it until a subsequent command. Also, if the inquiry data had already been cached as a result of a pass-through driver open or SIOC_INQUIRY, this may not even result in I/O.

```c
if (error = scsi_init_inquiry_data(dev))
    return error;
...
/*
 * Needs protection at LUN and Tgt.
 */
scsi_lun_lock(lp);
scsi_tgt_lock(tp);
tp-h>state |= T_ENABLE_SDTR;
...
scsi_tgt_unlock(tp);
scsi_lun_unlock(lp);
...
```

bzero(cdb, sizeof(cdb));
cdb[0] = CMDtest_unit_ready;
if (scsi_cmd(dev, SCB_DONT_PRINT, 6, cdb, 0, 0,
llp->mydriver_msecs, 0,&error))
{
    /*
    * Allow an incomplete open if this is a raw device.
    */
    if (major(dev) == mydriver_ddsw.raw_major)
    {
        scsi_lun_lock(lp);
        lp->state |= L_DISABLE_OPENS;
        scsi_lun_unlock(lp);
        return 0;
    }
    return error;
}
...

**dd_pass_thru_done**

The **dd_pass_thru_done()** routine is provided by the driver writer. It can have any unique name. Pass the name to SCSI Services by specifying it in the **dd_pass_thru_done** field of the **scsi_ddsw** structure.

If this routine exists in the **scsi_ddsw** structure, SCSI Services executes it on completion of a pass-through I/O. It allows the class driver to make note of any I/Os which have occurred and any resulting status and/or sense data.
The `dd_pass_thru_done()` function is called from within a critical section; it is not permitted to block.

**Declaration**

```c
int dd_pass_thru_done(
    struct buf *bp
);
```

**Parameters**

- `bp` buf structure

**Return Values**

`dd_pass_thru_done()` is declared as returning `int`; however, the return is not used by SCSI services.

### `dd_pass_thru_okay`

The `dd_pass_thru_okay()` routine is provided by the driver writer. It can have any unique name. Pass the name to SCSI Services by specifying it in the `dd_pass_thru_okay` field of the `scsi_ddsw` structure.

If a device is opened by a non-pass-through class driver and the driver specifies a `dd_pass_thru_okay()` entry point in its `scsi_ddsw` structure, the driver has complete control over what pass-through I/Os are allowed. If the driver does not specify a `dd_pass_thru_okay()` entry point, then pass-through I/Os are not allowed.

The `dd_pass_thru_okay()` function is called from within a critical section and may not block.

**Declaration**

```c
dd_pass_thru_okay(
    dev_t dev,
    struct sctl_io *sctl_io
);
```

**Parameters**

- `dev` The device number.
- `sctl_io` Struct containing ioctl information.

**Return Values**

`dd_pass_thru_okay()` is expected to return the following values:

- `PT_OKAY` Successful completion.
- `0` Error

### `dd_read`

The `dd_read()` routine is provided by the driver writer. It can have any unique name. Pass the name to SCSI Services by specifying it in the `dd_read` field of the `scsi_ddsw` structure.

If this routine exists in the `scsi_ddsw` structure, it is called instead of `physio()` by `scsi_read()`.

The `dd_read()` is called in a process context. It is not called from within a critical section; it may block.
Declaration

```c
int dd_read (
    dev_t dev,
    struct uio *uio
);
```

Parameters

- `dev` The device number.
- `uio` Structure containing transfer information.

Return Values

`dd_read()` is expected to return the following values:

- `0` Successful completion.
- `<>0` Error. The value is expected to be an `errno` value.

Example

```c
mydriver_dd_read(
    dev_t dev,
    struct uio *uio
)
{
    struct scsi_lun *lp = m_scsi_lun(dev);
    struct sf_lun *llp = lp->dd_lun;
    int error;

    scsi_lun_lock(lp);
    while (llp->state & ST_GEOM_SEMAPHORE)
        scsi_sleep(lp, &llp->state, PRIBIO);
    lp->state |= ST_GEOM_SEMAPHORE;
    scsi_lun_unlock(lp);

    sf_update_geometry(dev);
    error = physio(scsi_strategy, NULL, dev, B_READ, minphys,
                    uio);

    scsi_lun_lock(lp);
    llp->state &= ~ST_GEOM_SEMAPHORE;
    scsi_lun_unlock(lp);
    wakeup(&llp->state);
    return error;
}
```

dd_start

The `dd_start()` routine is provided by the driver writer. It can have any unique name. Pass the name to SCSI Services by specifying it in the `dd_start` field of the `scsi_ddsw` structure.

If this routine exists in the `scsi_ddsw` structure, it is called by `scsi_start()` to allow the driver to perform any necessary processing prior to sending the I/O down the SCSI stack.

The `dd_start()` function is called in the process and interrupt context from within a critical section in `scsi_start()`; `dd_start()` is not permitted to block.
The critical section in `scsi_start()` from where the `dd_start()` function is called, is protecting the `scsi_lun` structure. It is also guaranteeing that `lp->n_scbs` is consistent with the `dd_start()` function. The critical section also protects the incrementing of `n_scbs` in the `scsi_lun` structure and the incrementing of the SCSI subsystem unique I/O ID `scsi_io_cnt`.

If this routine does not exist, only “special” I/Os (`B_SIOC_IO` or `B_SCSI_CMD`) can be performed.

The driver’s `dd_start()` routine must dequeue the I/O from the appropriate list and perform whatever is necessary for the device to operate upon the I/O.

The parameters passed for this purpose are the `lp` and the `scb` parameters. The `scb` has the necessary `cdb[]` array for the SCSI command bytes.

**Declaration**

```c
struct buf *(*d_start) dd_start (  
    struct scsi_lun *lp,  
    struct scb *scb  
);  
```

**Parameters**

- `lp` The open LUN structure.
- `scb` Extra state information for I/O.

**Return Values**

`dd_start()` is expected to return the following values:

- `struct buf *bp` Successful completion.
- `NULL` Error

**Example**

```c
#include <sys/scsi_ctl.h>

struct buf *mydriver_dd_start(  
    struct scsi_lun *lp,  
    struct scb  
)  
{
    struct mydriver_lun *llp = lp->dd_lun;
    struct buf *bp;
    struct scb *head_scb, *scb_forw, *scb_back;
    int nblks, blkno, x;
    int lshift = llp->devb_lshift;

    /*
     * We could be more granular with locks, but
     * that would most likely cause too much
     * overhead getting/releasing locks.
     */
    scsi_lun_lock(lp);

    if ((bp = mydriver_dequeue(lp)) == NULL)  
    {  
        goto start_done;  
    }

    nblks = bp->b_bcount >> llp->log2_blk_sz;
```
if (bp->b_offset & DEV_BMASK)
    blkn = (unsigned) bp->b_offset >> llp->log2_blk_sz;
else
    blkn = (unsigned) (lshift > 0
        ? bp->b_blkno << lshift
        : bp->b_blkno >> -lshift);

scb->cdb[0] = (bp->b_flags & B_READ)
    ? CMDread10
    : llp->state & LL_WWV
    ? CMDwriteNverify
    : CMDwrite10;
scb->cdb[1] = 0;
scb->cdb[2] = blkn >> 24;
scb->cdb[3] = blkn >> 16;
scb->cdb[4] = blkn >> 8;
scb->cdb[5] = blkn;
scb->cdb[6] = 0;
scb->cdb[7] = nblks >> 8;
scb->cdb[8] = nblks;
scb->cdb[9] = 0;

/* Immediate Reporting (WCE) ON? */
if (llp->state & LL_IR)
if ((scb->cdb[0] == CMDwrite10) && (bp->b_flags & B_C))
    scb->cdb[1] |= WRITE_FUA_BIT;

if (lp->state & L_WOE && !(bp->b_flags & B_READ))
{
    if (lp->inquiry_data.inq2.dev_type == SCSI_MO)
        scb->cdb[1] |= 0x04;
    else /* SONY */
        scb->cdb[9] |= 0x40;
}

scb->cdb_len = 10;
scb->max_msecs = llp->mydriver_msecs;
scb->max_retries = 5;
scb->flags = SCB_4BYTE | SCB_ORDERED_TAG;
if (llp->state & LL_STINGRAY)
    scb->flags &= ~SCB_ORDERED_TAG;

/* Assume that scb->io_id will be set by caller within */
/* this CRIT */

/* Queue this bp into llp->active_bp_list HEAD for */
/* tracking */
if (llp->active_bp_list != NULL)
{
    scb->io_forw = llp->active_bp_list;
    head_scb = (void *) llp->active_bp_list->b_scb;
    scb->io_back = head_scb->io_back;

    scb->io_forw = (void *) scb->io_forw->b_scb;
    scb->io_back = (void *) scb->io_back->b_scb;
    scb->io_forw->io_back = bp;
    scb->io_back->io_forw = bp;
    llp->active_bp_list = bp;
}
else
{
    llp->active_bp_list = bp;
    scb->io_forw = scb->io_back = bp;
}
/* Although redundant with caller, set this in case
 * completion int */
bp->b_scb = (long) scb;

start_done:
    scsi_lun_unlock(lp);
    return bp;
}

dd_strategy

The dd_strategy() routine is provided by the driver writer. It can have any unique name. Pass the name to
SCSI Services by specifying it in the dd_strategy field of the scsi_ddsw structure.

The dd_strategy() routine is called by scsi_strategy() to perform whatever sorting or queueing the class
driver requires for normal I/O. For most drivers, enqueuing to lp->scb_q is necessary; the scsi_disk driver
calls disksort_enqueue().

The dd_strategy() is called both in a process and interrupt context; it is not allowed to block.

If the driver invokes scsi_strategy(), dd_strategy() is required. If the dd_read() or dd_write()
routines are not specified, SCSI Services will assume physio() is to be used.

NOTE  The scsi_strategy() calls dd_strategy() holding lun_lock.

Declaration

int (*dd_strategy) dd_strategy (  
    struct buf *bp,  
    struct scsi_lun *lp
);

Parameters

bp          transfer buf header.
lp          scsi LUN information.

Return Values

dd_strategy() is expected to return the following values:

0          Successful completion.
-1          Error

Example

The MP protection is provided for modification of the queues. Here is an example for a tape:

mydriver_dd_strategy(struct buf *bp, struct scsi_lun *lp)
{
    struct st_lun  *llp = lp->dd_lun;
    struct st_static_lun *sllp = llp->static_data;
DB_ASSERT(!(bp->b_flags & B_ERROR));

sllp->head_pos &= ~HEAD_FORWARD;

/* Check for valid request size in fixed block mode */
if (llp->block_size > 0 && bp->b_bcount %
    llp->block_size != 0)
{
    bp->b_flag = B_ERROR;
    bp->b_error = ENXIO;
    biodone(bp);
    return -1;
}
bp->b_resid = bp->b_bcount;
scsi_enqueue(lp->scb_q, bp, TAIL);
return 0;

A SCSI disk (scsi_disk) driver does not use the lp->scb_q. Instead, disksort() a service from the File
System, is used. The following is an example:

mydriver_dd_strategy{
    struct buf *bp,
    struct scsi_lun *lp
}
{
    dev_t dev = bp->b_dev;
    struct mydriver_lun *llp = lp->dd_lun;
    ASSERT(!(bp->b_flag & B_ERROR));
    return mydriver_enqueue(lp, bp);
}

mydriver_enqueue{
    struct scsi_lun *lp
    struct buf *bp
    {
        int x;
        struct mydriver_lun *llp = lp->dd_lun;
        struct buf *dp;
        /* Class driver's private queue */
        dp = lllp->lun_disk_queue;
        /* set B_FIRST to get queue preference */
        if (bp->b_flag & B_SPECIAL)
            bp->b2_flag = B2_FIRST;
        /* fake b_cyl 512K per cylinder */
        bp->b_cyl = (bp->b_offset >> 19);
        disksort_enqueue(dp, bp);
        return 0;
    }
}

NOTE The dd_strategy() must exist (be defined as non-NULL in the scsi_ddsw structure) if the
driver calls scsi_strategy().
**dd_write**

The `dd_write()` routine is provided by the driver writer. It can have any unique name. Pass the name to SCSI Services by specifying it in the `dd_write` field of the `scsi_ddsw` structure.

If this routine exists in the `scsi_ddsw` structure, it is called instead of `physio()` by `scsi_write()`.

This routine is called from `scsi_write()` in a process context. Since it is not called from within a critical section, it may block.

**Declaration**

```c
int dd_write (
    dev_t dev,
    struct uio *uio
);
```

**Parameters**

- `dev` - The device number.
- `uio` - Structure containing transfer information.

**Return Values**

`dd_write()` is expected to return the following values:

- `0` - Successful completion.
- `errno` - Error

**Example**

```c
#include <sys/scsi_ctl.h>
#define ST_GEOM_SEMAPHORE 2

mydriver_dd_write(
    dev_t dev,
    struct uio *uio
)
{
    struct scsi_lun *lp = m_scsi_lun(dev);
    struct sf_lun *llp = lp->dd_lun;
    int error;

    scsi_lun_lock(lp);
    while (llp->state & ST_GEOM_SEMAPHORE)
        scsi_sleep(lp, &llp->state, PRIBIO);
    llp->state |= ST_GEOM_SEMAPHORE;
    scsi_lun_unlock(lp);

    sf_update_geometry(dev);
    error = physio(scsi_strategy, NULL, dev, B_WRITE,
                   minphys, uio);

    scsi_lun_lock(lp);
    llp->state &= ~ST_GEOM_SEMAPHORE;
    scsi_lun_unlock(lp);
    wakeup(&llp->state);

    return error;
}
```
Synchronization in SCSI Drivers

When the SCSI Services calls the driver, it takes the appropriate locks to provide MP protection. The driver must provide protection for accessing its own private data and any data under the domain of the SCSI Services such as scsi_lun, scsi_tgt, scsi_bus, or the SCSI subsystem’s data. Locks are defined in <sys/scsi_ctl.h>.

Rules for Ordering Locks

The rules for ordering locks and semaphores help the kernel detect deadlocks in their use. When a thread of execution must hold more than one lock or semaphore, it must acquire them in increasing order. The order of locks and semaphores in the ascending order are as follows:

1. LUN lock
2. Target lock
3. Bus lock
4. Subsystem lock

If a thread of execution must hold both the LUN lock and target lock at the same time, the ordering rules assert that the code must acquire the LUN lock before it acquires the target lock. The class driver should normally not need any additional locks of its own. It should be able to use the LUN lock, for example, for protection of its own per-instance structures. However, if a driver uses a private lock, it must have the highest precedence. It must be acquired first, and must have a lower order number than the LUN lock, SCSI_LUN_LOCK_ORDER.

Subsystem Lock

The subsystem lock protects the SCSI subsystem’s global data. Only SCSI Services access this data, so the driver should have no need for this lock.

Bus Lock

Each scsi_bus structure has a lock associated with it that protects many of the fields in the structure. Most drivers do not need to use the bus lock because they ordinarily do not access the information maintained in the scsi_bus structure.

Some HP class drivers access the B_EXCLUSIVE flag in the state field of the scsi_bus structure.

Target Lock

Each scsi_tgt structure has a lock associated with it that protects some of the fields in the structure. Class drivers can access the open_cnt, sctl_open_cnt, state, and bus fields in this structure. Class drivers may only modify the state field, and must do so under the protection of the target lock. The target lock can also be used to prevent the open_cnt, sctl_open_cnt, or state field from being modified while other conditions are checked or actions are performed.

LUN Lock

Each scsi_lun structure has a lock associated with it that protects the fields in the structure and in the dd_lun private data area. See the following section on the LUN structure to see which fields class drivers can access and modify, and which locks protect those fields.
For the `driver_open()` routine, the class driver does not have any of the locks available until the kernel calls `scsi_lun_open()`, because `scsi_lun_open()` creates the `scsi_bus`, `scsi_tgt`, and `scsi_lun` structures.

For the `driver_close()` routine, the situation is similar. The locks are also available when the `dd_close()` routine is called. When `scsi_lun_close()` returns control to its caller, the locks are no longer available to the driver.

### Handling Concurrency for Tape Devices

#### I/O Access Concurrency

Make sure that no two outstanding I/O's are on the same device. The driver needs to protect the device from multiple outstanding I/O's from the same initiator. This situation can arise from concurrent I/O's from the same driver or from the pass-thru driver while tape driver I/O is active. Use the `dd_pass_thru_okay` and `dd_pass_thru_done` interfaces to handle concurrent pass_thru and tape driver I/O.

#### LUN Open Concurrency

The driver sets the `L_DISABLE_OPENS` flag on the state field of the `scsi_lun` struct defined in `scsi_ctl.h` to prevent multiple opens on the same tape device.
SCSI Services Summary

SCSI Services are commonly used SCSI functions that allow device and interface drivers to be much smaller and more supportable. In addition to providing most commonly used SCSI functions, WSIO SCSI Services also provide a supported pass-through mechanism. (See `scsi_ctl (7)` in the *HP-UX 11i v1 Driver Development Reference Guide* for information on pass-through.)

SCSI Services are summarized in Table 13-3, “SCSI Services.” For more detailed information on these services see the *HP-UX 11i v1 Driver Development Reference Guide*.

Table 13-3        SCSI Services

<table>
<thead>
<tr>
<th>Service</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>m_scsi_lun()</td>
<td>Returns <code>scsi_lun</code> pointer corresponding to the <code>dev_t</code> parameter passed in.</td>
</tr>
<tr>
<td>scsi_action()</td>
<td>Must ultimately be called after each I/O attempt completion, whether retrying the I/O or not, to log errors to the <code>dmesg</code> buffer, determine if the I/O is to be retried, or perform dynamic <code>qdepth</code> handling.</td>
</tr>
<tr>
<td>scsi_cmd(), scsi_cmdx()</td>
<td>For driver generated I/O requests, this routine creates and builds a <code>sctl_io</code> structure and a <code>bp</code>, attaches the <code>sctl_io</code> to the <code>bp</code>, forwards the <code>bp</code> to the <code>scsi_strategy()</code> routine, and cleans up when the I/O is completed.</td>
</tr>
<tr>
<td>scsi_ddsw_init()</td>
<td>Called from class driver's <code>driver_dev_init()</code> routine. Causes initialization of <code>blk_major</code> and <code>raw_major</code> fields in the driver's switch table (<code>ddsw</code>).</td>
</tr>
<tr>
<td>scsi_dequeue()</td>
<td>Removes I/O requests from queues maintained by SCSI Services.</td>
</tr>
<tr>
<td>scsi_dequeue_bp()</td>
<td>Externally available to dequeue particular <code>bp</code> from circular list. Intended for use with volume manager's <code>B_PFTIMEOUT</code>.</td>
</tr>
<tr>
<td>scsi_enqueue()</td>
<td>Places I/O requests on queues maintained by SCSI Services.</td>
</tr>
<tr>
<td>scsi_init_inquiry_data()</td>
<td>Called from class driver's <code>ddsw-&gt;dd_open()</code> routine. Performs the SCSI Inquiry request to the device to obtain the Inquiry data if not already done.</td>
</tr>
<tr>
<td>scsi_ioctl()</td>
<td>The <code>ioctl</code> commands that are supported by all drivers are implemented here to ensure consistency among drivers. If the specified <code>ioctl</code> command is not one of the common ones implemented in SCSI services, <code>dd_ioctl</code> is called if it is not NULL.</td>
</tr>
<tr>
<td>scsi_log_io()</td>
<td>Records the I/O attempt and its results in the <code>dmesg</code> buffer.</td>
</tr>
<tr>
<td>scsi_lun_open()</td>
<td>Called from class driver's <code>driver_open()</code> routine. Performs necessary open operations, including the invocation of the calling driver's <code>ddsw-&gt;dd_open()</code> routine.</td>
</tr>
<tr>
<td>scsi_read()</td>
<td>Synchronous read routine, which calls <code>dd_read()</code> if not NULL, or <code>physio()</code> if <code>dd_read()</code> is NULL.</td>
</tr>
</tbody>
</table>
### Table 13-3  
**SCSI Services (Continued)**

<table>
<thead>
<tr>
<th>Service</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>scsi_sense_action()</code></td>
<td>Interprets sense data for SCSI, CCS, or SCSI-2 or SCSI-3 compliance. It requires that the inquiry data for the device has been initialized by <code>scsi_init_inquiry_data()</code> before it can interpret it.</td>
</tr>
<tr>
<td><code>scsi_snooze()</code></td>
<td>Performs a sleep without tying up the processor. Must not be called by a thread of execution that holds any lock and must only be called in a process context (not ICS). Currently, this routine is used only by <code>scsi_disk</code> driver to delay subsequent device access following inquiry to a particular model of Quantum disk drive.</td>
</tr>
<tr>
<td><code>scsi_strategy()</code></td>
<td>The first place in the I/O path that all I/O requests have in common. Its primary purpose is to enqueue the <code>bp</code> to await the necessary resources to allow the request to be sent to the interface driver, and thus, the hardware.</td>
</tr>
<tr>
<td><code>scsi_write()</code></td>
<td>Synchronous write routine, which calls <code>dd_writer()</code> if not NULL, or <code>physio()</code> if <code>dd_write</code> is NULL.</td>
</tr>
</tbody>
</table>
14 Writing PCI Device Drivers

This chapter presents routines and conceptual material specifically for drivers of Peripheral Component Interconnect (PCI) devices. PCI is an industry-standard bus supported as a bus-nexus CDIO on HP-UX systems as of Release 10.20, as a means of providing expansion I/O. PCI Services are a supplement to the WSIO HP-UX driver environment, providing PCI-specific functionality to drivers that use a PCI bus either as a means to providing expansion slots or for core I/O functionality.

In conjunction with the WSIO Services driver environment, PCI Services form the complete environment necessary to write an HP-UX driver capable of handling a PCI card. The services are generic in nature and not tied to any particular PCI bus adapter.

This chapter corresponds to the PCI Local Bus Specification, Revision 2.1. It also specifies the features, possible limitations, and assumptions of the services that you may need to be aware of.

The HP-UX PCI Services routines are described in the HP-UX Driver Development Reference; they are also summarized in the “PCI Services Summary” section.

NOTE The examples in this chapter follow the routine naming conventions described in Chapter 4, “Writing a Driver.”
PCI Overview

This section gives you a brief overview of PCI. It is not intended to be sufficient PCI information in itself. The developer should be familiar with the PCI Local Bus Specification, Revision 2.1 before trying to write a driver for a PCI card.

PCI Register Spaces

There are three register spaces in PCI:

- PCI Configuration Space
- PCI Memory Space
- PCI I/O Space.

Generic configuration registers are placed in configuration space. Registers for card-specific control and status and for on-card data buffers are generally located in PCI memory space or (less often) in PCI I/O space.

PCI Configuration Space

PCI configuration space holds specific registers having to do with initialization and configuration of PCI devices. Some or all of this register space is the same for all PCI devices, giving generic initialization software the ability to recognize and configure all PCI-compliant devices.

This space is accessed primarily at startup time, when initialization occurs, but there is no prohibition on accessing it at other times after startup.

The following PCI Services access registers in PCI configuration space.

- pci_read_cfg_uintN_isc()
- pci_write_cfg_uintN_isc()

These functions take a configuration space offset (0x00–0xff) as their address inputs. See “Defined Constants”. The registers at addresses 0x00–0x3f are defined in the PCI Local Bus Specification, Revision 2.1, but the remainder of the space can be used by the card maker for any card-specific registers it sees fit to put there. In most cases, however, card-specific registers are placed in PCI memory space or PCI I/O space instead.

PCI Memory Space

Most cards place their registers for control, data buffering, and status in PCI memory space. In HP-UX systems, accesses to PCI memory space have higher performance than access to PCI I/O space. Registers mapped in PCI memory space respond to memory cycles on the PCI bus.

The following PCI Services access registers in PCI memory space.

- READ_REG_UINTn_ISC()
- WRITE_REG_UINTn_ISC()

These macros take Virtual Addresses (which are mapped to PCI memory addresses) as their address inputs. They have different effects depending on whether or not PCI_LITTLE_ENDIAN_ONLY is defined by the driver prior to including pci.h. See “The PCI_LITTLE_ENDIAN_ONLY Flag” for more details.

Mapped PCI memory space, on workstations only, can also be accessed directly. In this case the developer will have to handle “endian” issues oneself.
PCI I/O Space

Some cards place their registers for control, data buffering, and status in PCI I/O space. Registers mapped in PCI I/O space respond ONLY to I/O cycles on the PCI bus.

The following PCI Services access registers in PCI I/O space.

- `pci_read_port_uintN_isc()`
- `pci_write_port_uintN_isc()`

These functions take port handles and offsets as their address inputs.

PCI Transaction Ordering

This section covers the ordering of transactions to and from PCI space. These transactions include:

- Processor mastered reads and writes to PCI space
- PCI card mastered reads and writes to host memory
- Interleaved processor and PCI card mastered reads and writes of host memory space

Host bus to PCI bridges used in HP-UX systems need to comply with the transaction ordering requirements of both buses. As a result, in certain cases the order of completion guaranteed under the Producer-Consumer model as defined in the *PCI Local Bus Specification, Revision 2.1* is not met.

Ordering of Processor Mastered Reads and Writes To PCI Space

This section details transaction ordering for processor mastered PCI transactions. Typical examples of this type of transaction are reading and writing of registers on a PCI interface card.

**Blocking versus Nonblocking Transactions**  Processor mastered reads of PCI space are blocking transactions. This means that ordering is not a problem with reads, since only one read can occur at a time. A read holds the caller (processor) until it completes. The hardware implementation prevents a second processor reading from the same PCI space until the first processor’s read completes.

Writes to PCI registers, on the other hand, are nonblocking (“posted”) transactions. This means that, to get better performance, the writing process does not wait for a write to complete after calling for it (writes do not block). The write will complete on its own, and the writer can do other things, including other writes, in the meantime. Because multiple outstanding uncompleted writes are possible (and common) under this model, ordering must be established on the completion of the writes.

Processor mastered PCI write ordering is relatively simple. If a processor writes to registers A, B, and C in that order, the writes will complete such that they are only observable in the same order (for example, you could never observe that B had been written but A had not been yet). If two or more processors are writing to registers, their ordering with respect to each other is considered irrelevant, but the ordering of their individual writes is preserved as above. This is the order of completion guaranteed under the Producer-Consumer model as defined in the *PCI Local Bus Specification, Revision 2.1*.

**Write Side-Effects**  The side-effects of any write are not guaranteed to happen immediately. Writes are posted; they will complete eventually.

All posted writes must be flushed and completed before any read is allowed to complete. So, to assume a write’s effects have actually occurred, a read must be performed to flush the writes posted in the queue. You must keep this in mind when coding register writes; most of the time, it is acceptable to not know when a register write completes, but in some cases, you have to be careful.
A good example of such a case is when a driver’s interrupt service routine (ISR) is dealing with the interrupt request register (IRR) on a card. Clearing a bit in the register indicates that the interrupt has been serviced. This is done by posting a write to the register. If the driver posts this write and exits its ISR, it could conceivably get interrupted again immediately because the write hadn’t yet reached the bit in the IRR to tell it to stop trying to interrupt. One solution to this potential problem is to make sure to read back the value in the IRR before exiting from the ISR. Most drivers do this anyway so they can handle multiple interrupts in the same ISR visit.

**Ordering of PCI Card Mastered Reads and Writes to Host Memory**

We use the terms DMA read for a PCI mastered read from host memory, and DMA write for a PCI mastered write to host memory. In current hardware implementations, transaction ordering of DMA reads and DMA writes are ONLY preserved when the target memory location is contained in the same processor cacheline. In other cases, DMA reads are allowed to pass DMA writes and driver writers need to take this behavior into account.

If your driver needs the exact PCI producer-consumer behavior, as seen from the PCI card, you must ensure that the element(s) residing in host memory, requiring strict ordering, are physically on the same cacheline. Current hardware implementations have cachelines that are multiples of 32 bytes in length. For safety you should make sure that you limit your flag or status elements to 32 bytes aligned on `MAX_CACHLINE_SIZE` boundaries (defined in `/usr/include/sys/dma.h`).

**Ordering of Interleaved Processor and PCI Card Mastered Reads and Writes to Host Memory**

If your driver expects PCI or PA ordering rules to apply in this situation, you need to ensure that your producer-consumer elements reside on the same cacheline. The following scenario does not meet the producer-consumer transaction ordering requirements.

- Cache line X holds the card’s status — initially “working”
- Cache line Y holds the card’s next command — initially “go to sleep”
- Card finishes work and sets status in cacheline X to “done”
- Card reads its next command from cacheline Y
- Processor writes command to cacheline Y “do more work”
- Processor checks status in cacheline X

If the processor’s read of cacheline X (status) returns “working”, the processor assumes that the card has not checked its command yet. Therefore it has not gone to sleep and does not need to be awakened. If the status read returns “done”, the processor wakes up the card.

The crux of the problem here is that ordering is not enforced between the two cachelines and DMA reads can pass DMA writes. Thus, both the processor and the card’s reads can return the original value. This would result in the card going to sleep and the processor not waking it up.

If you cannot place the status and commands on the same cacheline, you must use some other means to ensure correct behavior. One possible workaround would be to set a timeout to ensure that the above deadlock did not occur. In most cases, commands are written to the card’s register, i.e., the command is not in host memory and the above scenario would not apply.
Ordering of Interleaved Processor Mastered Writes, PCI Card Mastered Reads of Host Memory, and Processor Mastered Writes of Host Memory

The following scenario does not meet the producer-consumer transaction ordering requirements.

- Processor writes a command to the PCI card to stop processing a task list in host memory, because the processor is about to update or change the list.
- Processor begins updating the task list in main memory.
- Card does a DMA read of the next (possibly being updated) element of the task list in main memory as a part of normal processing.
- Posted processor write to the card arrives at the card telling it to stop processing the list, which unfortunately it has just done.

The problem in this case is similar to the previous problem. DMA reads by a PCI master are allowed to pass processor writes to PCI space. Since processor writes are posted, ordering is not guaranteed on the combination of the internal system bus and the PCI bus. This situation can be avoided by doing a processor read of PCI space immediately following the processor write, as shown next.

- Processor writes a command to the PCI card to stop processing the task list.
- Processor does a “dummy” read of the PCI card (perhaps a status register on the card) to make sure that the “posted” write to PCI space has completed. Perhaps a read of card status is required here in any case to ensure that the DMA engine has stopped fetching tasks.
- Processor updates the task list in main memory.
- Processor writes a command to the PCI card to resume task processing.

The above behavior can occur on all shipping PCI based systems as of the date of this document. Drivers written for workstations should probably always ensure that, where necessary, posted writes are followed by “dummy” reads to ensure ordering. This behavior will probably not occur in servers due to chipset implementation, and may not occur in future workstation products.

PCI Endian Issues

HP PA-RISC is a big-endian system; for a multibyte quantity, the most significant byte (MSB) has the lowest address, and the least significant byte (LSB) has the highest. Intel’s i386 processors, on the other hand, are little-endian. Because PCI was derived from the PC world, it, too, is little-endian.

When multibyte words are transferred between the PCI bus and the system bus (HP PA-RISC), the bytes of the word are reversed or swapped by the hardware.

This insures that the receiving system can properly interpret and store the data, from most significant byte to least significant byte. This will not happen when the data is transferred byte-by-byte, but this method is inefficient.

Byte Swapping

In order for each system to get data in the format it expects, the PCI hardware uses a hardwired swapping mechanism at the interface between the two systems. The hardware swaps each byte of a 32-bit word so that all the bytes end up in the correct order on both sides of the interface. This means that large arrays of bytes, such as LAN packets and disk blocks, are in the correct order, even if they are transferred a multibyte word at a time.

This byte-ordering ensures that devices like disks, that are connected to the built-in SCSI on the internal system bus, can instead be connected to a SCSI card on the PCI bus.
When Pre-swapping is Required

Because of the byte swapping, the interpretation of multibyte integers is problematical. To see why this is so, assume that the transfer is occurring from the big-endian system to the little-endian system, and that swapping is being performed. If the byte array in question is a four-byte integer, it will be stored in big-endian format, MSB at the lowest address, on the little-endian side. If a device on the little-endian side of the interface decides to interpret these bytes as a four-byte integer; however, the “value” it will see will have all the bytes reversed. The same thing happens when transfers go in the opposite direction.

To correct the misinterpretation of multibyte integers on the opposite side of the bus, any multibyte quantity that is to be interpreted as an integer will have to be preswapped. This preswapping is then reversed by the hardwired swapping, making the value correct for integer interpretation on the other side of the interface. If the integer is stored in memory, however, it will end up reversed.

Several macros are provided in the file pci.h to assist in swapping data.
PCI Device Setup

This section is a collection of several pieces of information that you need to understand before attempting to set up a PCI device.

Mapping Base Address Registers into PCI Memory and I/O Space

When an HP-UX system boots, processor dependent code (PDC), I/O dependent code (IODC), and HP-UX system code maps a PCI card's memory space base address registers into PCI memory space and I/O space base address registers into PCI I/O space.

The system attempts to map in all memory and I/O regions described by every PCI device or function's memory and I/O base registers located in the PCI configuration space. If a suitable mapping is found, the system will write the base of the range back into the corresponding base address register. This address will be a PCI memory address if the base register identifies itself as a memory base, and a PCI I/O address if the base address identifies itself as an I/O base.

A driver's `driver_attach()` routine can then access the values loaded into the base registers in configuration space. It is important that a driver does not overwrite these addresses with different values, except as follows: As long as response to memory or I/O accesses via the command register has not yet been enabled, it is acceptable to store the register contents, write all ones to the register to determine the region size (as explained in the `PCI Local Bus Specification, Revision 2.1`) and, then restore the original contents.

Using the Base Address Registers

Before a driver can actually use these base addresses, another kind of “mapping” must take place. The problem is that the addresses placed in the base address registers by the system do not contain Virtual Addresses usable by the computer. Instead, they contain PCI addresses, used to talk on the bus. If a base address register is a memory base, it contains a PCI memory address. If it is an I/O base, it contains a PCI I/O address (See “PCI Register Spaces” for more information).

In either case, to use the PCI address in the base register, a mapping to a PA resource must take place, in order to allow the system to access the registers pointed to by the base.

It is very important that you do NOT arbitrarily mask bit 0 of a base address register. This bit indicates whether or not this particular set of registers responds to PCI memory cycles or PCI I/O cycles. During early PDC/IODC configuration, the defined base address registers are written in a manner prescribed by the PCI specification to determine size, alignment, and access type. If bit 0 is a “1” then PDC/IODC probing has determined that this particular register set ONLY responds to I/O cycles. If the base address register responds to I/O cycles, you MUST use the PCI services provided port I/O routines for access.

Using PCI Memory Base Registers

To use a PCI memory base address register, the range of PCI memory space must be mapped to a range of PA memory space. This is accomplished by calling `map_mem_to_host()`. The `map_mem_to_host()` call takes the PCI memory address (obtained directly from the base address register) and a size as inputs, and returns a Virtual Address that can be used to access that PCI address range. The accessor macros, `READ_REG_UINTn_ISC()` and `WRITE_REG_UINTn_ISC()`, take PA virtual memory addresses as arguments, not PCI memory addresses.
NOTE

After reading a PCI memory base address register’s value out of PCI configuration space, it is usually necessary to mask off the bottom four bits prior to making services calls such as `map_mem_to_host()`, since they have special values defined by the *PCI Local Bus Specification, Revision 2.1* (See “Sample driver_attach() Routine” for an example).

Once this virtual mapping is done, the machine uses PA memory-mapped I/O to access the range. In other words, accesses to that range of PA memory space will be transmitted through into the PCI memory space. This just means that loads and stores to these PA memory addresses will result in loads and stores to the registers you wish to access.

For WSIO drivers, the `if_reg_ptr` member of the ISC structure is a Virtual Address corresponding to a base address register that has already had this virtual mapping done to make it usable by the driver and system. If `if_reg_ptr` is NULL, the driver needs to map the range itself (see “Mapping the Memory Base Register” for information on this). This is generally done in a PCI device’s `driver_attach()` routine.

**Using PCI I/O Base Registers**

To use a PCI I/O base address register, the corresponding range of PCI I/O space must be mapped to a resource managed by PCI Services called a port handle, defined by the PCI structure `PCI_PORT_HNDL`. The PCI I/O space accessor functions, `pci_read_port_uintN_isc()` and `pci_write_port_uintN_isc()`, take port handles as arguments. To do this mapping from a PCI I/O address to a port handle, the driver must read the I/O base registers from configuration space and call `pci_get_port_handle_isc()`, which takes a PCI I/O space address and a size as input, and returns a `PCI_PORT_HNDL` as output. This port handle (with an offset) is then used to access the registers in PCI I/O space.

NOTE

When reading a PCI I/O base address register’s value out of PCI configuration space, it is necessary to mask off the bottom two bits prior to making a call to services such as `pci_get_port_handle()`, since they have special values defined by the *PCI Local Bus Specification, Revision 2.1* (See `pci_get_port_hndl_isc` (PCI3) for an example).

**Automatic IRQ Determination**

PCI drivers calling `isrlink()` and `isrunlink()` should always pass `-1` as the `irq_line` argument. This argument value causes the functions to read the needed IRQ information from the PCI device or function configuration space Interrupt Pin and/or Interrupt Line registers and use it to set up the ISR properly. If you need the IRQ information, you can read it from the Interrupt Line register.

**Mapping the Memory Base Register**

Many cards will have only a single range of registers (only a single memory base address register). For cards like these, the `if_reg_ptr` field in the ISC structure is useful.

PCI Services automatically maps one memory-space register into the `isc->if_reg_ptr` field in the following manner and with the following limitations:

- Only the first nonzero 32-bit memory base register found is mapped, starting at 0x10 and searching up to 0x24 inclusive. These are the six defined base address register locations in PCI configuration space. A Virtual Address for accessing this register is stored in `if_reg_ptr`. 

However, if that base register’s size (the size of the register range) is in excess of 8 KB, it is NOT mapped and if_reg_ptr is set to NULL. In this case, the driver itself must map the base registers it wants, using the PCI bus-dependent configuration access routines in conjunction with map_mem_to_host().

If if_reg_ptr is NULL and the result of a map_mem_to_host() call is NULL, then for whatever reason, this particular address could NOT be mapped and you MUST NOT attempt to access it.

These limitations are necessary to define which of many possible base registers will be mapped, as well as to prevent unnecessary use of translation lookaside buffers (TLB). If PCI Services do not map in any memory base register, or if there are more registers than the first one found as above, the driver can read the base registers explicitly from the PCI device or function’s configuration space and get a PA virtual mapping with the map_mem_to_host() kernel routine. (See “Sample driver_attach() Routine” for an example).

The limitations also prevent wasting of kernel resources on base registers that we may not wish to map in the normal way (for example, a graphics card frame buffer is an enormous range that should be treated differently from a regular register range). PCI Services has arbitrarily decreed that anything bigger than 8 KB should be dealt with by the driver, not mapped automatically by WSIO services.

**PCI Configuration Space Restrictions**

The registers in the PCI configuration space of each device are described in the *PCI Local Bus Specification, Revision 2.1*. Many of these registers are writable, but not every writable register is appropriate for a driver to modify. Some of the fields are set up on behalf of the driver and card by the system, which has information that a driver or card could not know about system parameters.

The basic guideline is that things that you do not understand or need not have anything to do with should not be altered. The following are some examples of configuration registers to leave alone:

- **The Command Register (most parts of it)**
  
  The command register must be written by drivers in order to enable bus-mastering, memory space access, and I/O space access, among other things. Many bits in this register are irrelevant to a driver and some have already been set by the system. Bits in the command register that may have been previously set must not be overwritten. Therefore, when a driver wants to set a bit in the register, it must first read the current state of the register, use bitwise OR or AND to make any changes, then write the value back. This procedure preserves bits previously set by the system.

- **The Latency Timer Register**
  
  This is set by the system. It should not be tampered with by individual drivers, as incorrect settings can degrade overall system performance.

- **The Cache Line Size Register**
  
  This register is set by the system to match the machine’s cache line. Drivers do not know the cache line size for the particular machine they are currently running on, so they should not change this register’s contents.

- **The Base Address Registers**
  
  The system uses the information in these registers to map their ranges into PCI memory and I/O space. It then writes a value back into the register corresponding to the base of the range it allocated. These ranges should not be overwritten by drivers, with one exception. In some cases, it may be necessary for a driver to determine the size and alignment of the range a base address register is mapped to. The procedure for getting this information involves writing all ones to the register, reading the result back, and decoding it for the needed values, as described in the *PCI Local Bus Specification, Revision 2.1*. Doing this is permitted as long as the original value is read and stored first, then restored to the register after the size has been determined. This should only be done before memory or I/O transactions to the card have been enabled through the command register.
The Interrupt Line Register

System-specific interrupt routing information is stored in this register. Writing a new value to it will probably cause the card to stop working.
PCI Device Operation

The PCI_LITTLE_ENDIAN_ONLY Flag

We recommended that drivers define the PCI_LITTLE_ENDIAN_ONLY flag before they include pci.h. This will help them get better performance from their I/O accesses.

Most PCI drivers are written for cards whose primary method of accessing registers is through PCI memory space.

PCI drivers written for workstations only, currently all third party drivers, may use direct C code constructs to access registers in PCI memory space. For example:

```c
myClearRegs(regsToInit,size)

u_int *regsToInit;
int size;
{
  int i;
  for (i=0;i<size;i++)
    *regsToInit++ = 0;
}
```

These drivers may also use the READ_REG_UINTn_ISC() and WRITE_REG_UINTn_ISC() macros with the PCI_LITTLE_ENDIAN_ONLY flag defined in the pci.h header. The choice of whether to directly access a register or to use one of the macros essentially becomes whether or not you want to swap.

The READ_REG_UINTn_ISC() and WRITE_REG_UINTn_ISC() macros are the safest accessors of PCI memory space, but what they are actually defined to do depends on whether or not the PCI_LITTLE_ENDIAN_ONLY flag was defined by the driver before the driver source code included the pci.h header.

If the driver does NOT explicitly define PCI_LITTLE_ENDIAN_ONLY before including pci.h, then the macros expand into function calls that are guaranteed to byte-swap correctly and perform the memory access. This can be considered “extra safe” mode; it will always work on all bus adapters in the future. The function calls guarantee PCI-adapter-independence. However, extra function call overhead is added to the register access, reducing its performance.

If the driver DOES explicitly define PCI_LITTLE_ENDIAN_ONLY, the performance loss due to the function call is taken away. In this case, the macros are expanded by the preprocessor into a series of in-line instructions that byte-swap and perform the access without a function call, under the assumption that the PCI adapter under which the card is running has directly mapped the PCI memory space into driver-accessible PA I/O space. This assumption is valid for all current and planned PCI adapters, with the single exception of a few special PA internal system bus based server PCI card projects. All regular drivers (i.e., those that are not explicitly written to drive a specially-equipped PA internal system bus based card) will benefit from defining the PCI_LITTLE_ENDIAN_ONLY flag and should do so before including pci.h.
Writing PCI Device Drivers
PCI Device Operation

The following pseudocode (resembling and summarizing the actual code in pci.h) may help explain the flag’s relation to the macros, and how and why to use it:

```
#ifdef PCI_LITTLE_ENDIAN_ONLY

#define READ_REG_UINTn_ISC(isc, addr, value) 
  (*value = ENDIAN_SWAP_MACRO(*addr))
#define WRITE_REG_UINTn_ISC(isc, addr, value) 
  (*addr = ENDIAN_SWAP_MACRO(value))
#else /* *NOT* PCI_LITTLE_ENDIAN_ONLY */

#define READ_REG_UINTn_ISC(isc, addr, value) 
  isc->adapter_dependent_readN_function_call(addr, value)
#define WRITE_REG_UINTn_ISC(isc, addr, value) 
  sc->adapter_dependent_writeN_function_call(addr, value)
#endif /* PCI_LITTLE_ENDIAN_ONLY */

Direct Memory Access (DMA)

A PCI device acting as a PCI bus master uses direct memory access (DMA) to generate read or write cycles that access locations in PA memory and card memory. DMA is a primary method of getting information to or from a card in large chunks, as opposed to doing many reads or writes to buffers of card registers.

PCI has no special routines to perform DMA. It uses the standard WSIO Services calls for bus-independent DMA, including:

- init_map_context()
- wsio_map(), wsio_fastmap(), and wsio_unmap()
- dma_setup() and dma_cleanup()
- The iovec and dma_parms structures

In the HP-UX Driver Development Reference, see dma_cleanup (WSIO3), dma_parms (WSIO4), dma_setup (WSI03), init_map_context (CDIO3), iovec (KER4), wsio_fastmap (WSI03), wsio_map (WSI03), and wsio_unmap (WSI03).

Be aware that certain combinations of WSIO mapping service calls can interact with PCI masters to create an inconsistent view of memory. See “PCI Masters and Coherence”.

Many EISA drivers make calls to functions like eisa_dma_setup() and eisa_dma_cleanup(). There are no corresponding PCI functions.

The only thing PCI-specific about performing DMA with a PCI device is that the device’s command register (PCI_CS_COMMAND) in PCI configuration space contains a bit (PCI_CMD_BUS_MASTER) that must be set (with pci_write_cfg_uintN_isc()) in order to allow the device to master the bus. The use of this bit is illustrated in “Sample driver_attach() Routine”.

```
PCI Masters and Coherence

It is possible for prefetching of host memory by the hardware chipset to result in a PCI master reading stale data, even though the proper \texttt{dma\_sync} calls have been made. This does not occur if the mapping is done with \texttt{wsio\_map()} with flags \texttt{IO\_NO\_SEQ} and \texttt{IO\_SAFE} set. See \texttt{pci\_errata} (PCI5) in the \textit{HP-UX Driver Development Reference} for details.
Leveraging Existing Drivers

Multibus Drivers

Some cards for different buses have similar chip sets, making the programming models very similar for the base functionality. Consequently, a single driver can handle the functionality for the different bus cards.

Writing a Multibus Driver

A multibus driver is one in which a similar chip set appears on cards that plug into multiple buses. An example of this is the current SCSI driver. Similar SCSI chips exist in devices on the GSC bus, the EISA bus, and the PCI bus. A single driver, `scsi_c720`, is capable of controlling these SCSI chips no matter where they live.

Because the programming model of the base functionality is so similar, it makes sense to have a single driver to handle this functionality. Conversely, however, the bus-specific initialization of the nonbase functionality can often be radically different. The WSIO environment supports multibus drivers in the following ways:

- Many of the initialization functions are embedded in bus-independent functions that have bus-dependent implementations. This means that WSIO is responsible for making sure that the right thing is done when a driver calls a generic function like `map_mem_to_host()`. This moves the handling of bus-specific differences out of the driver and into the WSIO environment. Keeping the driver clean of calls specific to the current PCI adapter. See “Bus-Independent Functionality, Bus-Dependent Implementation”.

- Since each bus has a different attach chain, drivers can provide a separate `driver_attach()` routine for each bus. With careful handling, this can localize bus-specific functionality in the `driver_attach()` routines, allowing the `driver_if_init()` routine to handle bus-independent initialization and keeping the rest of the driver routines clean.

Whether or not you are planning to write a multibus driver, it is a good idea to keep as much PCI specifics in the `driver_attach()` routine as possible, just in case a card comes along someday for a new bus that uses the same or similar chips as the PCI card you are writing a driver for now. This is only a suggestion, as it does not make sense to compromise your current driver or make a huge and ungainly `driver_attach()` routine if there is clearly no need to.

Bus-Independent Functionality, Bus-Dependent Implementation

This class of functions allows multibus drivers to make a single call, allowing the driver environment to hide any bus-dependent implementation.

In PCI, the following features are supported. (There are a host of completely bus-independent functions that, by having no dependency on PCI, are supported by definition.)

- `isrlink()` and `isrunlink()`:
  - Set `irq_line` to -1 to have the card supply the IRQ number to the system.
  - See “Automatic IRQ Determination”.

- `isc->if_reg_ptr` value: One memory space base register is mapped automatically, subject to the conditions described in “Mapping the Memory Base Register”.

- `wsio_map()`, `wsio_unmap()`, `wsio_dma_alloc()`, `wsio_dma_free()`, and others in the WSIO family of coherent I/O DMA services.

- `dma_setup()` and `dma_cleanup()`

The WSIO functions `wsio_get_interrupts()` and `wsio_get_registers()` are not supported for PCI. See “Unsupported WSIO Functions” for details.
PCI Services Summary

PCI Services are accessed through special PCI functions that allow device and interface drivers to be much smaller and more supportable.

These functions are summarized here and described in detail in the *HP-UX Driver Development Reference*.

- `pci_desc_bus_transactions_isc()` — Allow a driver to describe the typical bus-performance-transaction size.
- `pci_get_fru_info_isc()` — Get the field replaceable unit (FRU) information for the device associated with an ISC.
- `pci_get_port_hndl_isc()` — Get a system-defined handle for manipulating the range of PCI I/O-space ports.
- `pci_read_cfg_uintN_isc()` — Read an 8-, 16-, or 32-bit unsigned integer from a PCI configuration register.
- `pci_read_port_uintN_isc()` — Read little-endian data from a PCI I/O-space port previously identified by a call to `pci_get_port_hndl_isc()`.
- `pci_unget_port_hndl_isc()` — Delete a handle returned by `pci_get_port_hndl_isc()`.
- `pci_write_cfg_uintN_isc()` — Write an 8-, 16-, or 32-bit unsigned integer into a PCI configuration register.
- `pci_write_port_uintN_isc()` — Write little-endian data to a PCI I/O port previously identified by a call to `pci_get_port_hndl_isc()`.
- `CONNECT_INIT_ROUTINE()` — Associate a `driver_if_init()` routine with the driver.
- `PCI_ATTACH_DEV_INIT_ERROR()` — Notify WSIO Services that an error occurred during a device’s initialization.
- `READ_REG_UINTn_ISC()` — Read and byte-swap 8-, 16-, or 32-bit data from a little-endian bus.
- `WRITE_REG_UINTn_ISC()` — Byte-swap and write 8-, 16-, or 32-bit data to a little-endian bus or a host memory area shared by the driver and a little-endian bus master.
Unsupported WSIO Functions

PCI Services do NOT support the following WSIO functions.

- **wsio_get_interrupts()**
  
  This function is provided to tell the driver what interrupt line a card is using. This information is primarily used to link or unlink an ISR. In PCI, `isrlink()` and `isrunlink()` should be called with `irq_line` set to -1, indicating that the system should determine the appropriate IRQ from the card, and rendering the `wsio_get_interrupts()` call unnecessary. If the IRQ is needed for some other reason, it can be read from the Interrupt Line register in PCI configuration space.

- **wsio_get_registers()**
  
  This function is designed to return the base address register for a card’s memory-mapped I/O. For PCI this information is available in the ISC at `driver_attach` time as the value `if_reg_ptr`. Also see “Mapping the Memory Base Register”.


Multiprocessor (MP) Safety

All PCI drivers should be coded to be MP safe.

Specifically, this means that they should not rely upon SPL levels to guarantee exclusive access to critical sections, but should instead protect their own critical sections using spinlocks, semaphores, and other methods of MP protection. See Chapter 3, “Multiprocessing,” for details.
Constants and Data Structures

The constant definitions and data structures are defined in the PCI header file, pci.h.

User Visible PCI-Specific Data Structures

typedef struct _pci_id
{
    uint16_t vendor_id;
    uint16_t device_id;
} PCI_ID;

Defined Constants

/* Configuration space offsets. */
#define PCI_CS_VENDOR_ID 0x00
#define PCI_CS_DEVICE_ID 0x02
#define PCI_CS_COMMAND 0x04
#define PCI_CS_STATUS 0x06
#define PCI_CS_REV_ID 0x08
#define PCI_CS_CLASS_PROG_IF 0x09
#define PCI_CS_CLASS_SUB_CLASS 0x0a
#define PCI_CS_CLASS_BASE 0x0b
#define PCI_CS_CACHE_LINE_SIZE 0x0c
#define PCI_CS_LATENCY_TIMER 0x0d
#define PCI_CS_HEADER_TYPE 0x0e
#define PCI_CS_BIST 0x0f
#define PCI_CS_INTERRUPT_PIN 0x3d

/* masks for configuration data */
#define PCI_CS_MULT_FUNC_MASK 0x80

/* bit definitions for configuration space command
 * register */
#define PCI_CMD_IO_SPACE 0x001
#define PCI_CMD_MEM_SPACE 0x002
#define PCI_CMD_BUS_MASTER 0x004
#define PCI_CMD_SPEC_CYCLES 0x008
#define PCI_CMD_MEM_WR_INVAL_EN 0x010
#define PCI_CMD_VGA_PAL_SNOOP 0x020
#define PCI_CMD_PARITY_ERR_RESP 0x040
#define PCI_CMD_WAIT_CYCLE_CNTL 0x080
#define PCI_CMD_SERR_ENABLE 0x100
#define PCI_CMD_FAST_BACK_EN 0x200
A Sample PCI Driver

The following example is a skeleton that demonstrates how to write a PCI device driver in HP-UX using PCI and WSIO Services. The only part of this example that is PCI-specific is the `driver_attach()` routine. The other parts are typical of all WSIO drivers. They are included here for context and completeness. Chapter 4, “Writing a Driver,” contains more complete information on the structures and functions needed to write a WSIO driver.

We have a hypothetical PCI device, the ZZZ8109C PCI Blender card, for which we want to write a driver.

The blender is a character device, so our driver will be a character device driver. A character device is the counterpart of a block device, and has to do with how a device accesses its data and does DMA. The only type of PCI card that would be a block device would be a SCSI adapter or disk or tape drive controller.

Our example driver is written as a monolithic driver. This means it is both an interface driver (one that touches real hardware and registers) and a device driver (one that has a device special file). Even though we are writing both an interface and a device driver, we specify `T_INTERFACE` in the `wsio_drv_info_t` structure, since we cannot specify both.

Following the routine-naming conventions described in “Step 1: Choosing a Driver Name” in Chapter 4, “Writing a Driver,” we name the driver `ZZZ` and place it in the (arbitrary) class `blender`.

**Sample WSIO Setup and Structures**

We include the necessary header files. See the reference pages for each kernel call and data structure the driver uses to find out which headers the driver requires. WSIO drivers generally require the `<wsio/wsio.h>` header file. PCI drivers also require the `<sys/pci.h>` header file.

```
#include <wsio/wsio.h>
#include <sys/pci.h>
```

Next, we declare the driver’s routines that can be called by the kernel. These are used in the `drv_ops_t` structure.

```
int ZZZ_open();
int ZZZ_close();
int ZZZ_read();
int ZZZ_write();
int ZZZ_ioctl();
```

We need a `ZZZ_saved_attach` function pointer to store the old head of the attach chain when we add our `ZZZ_attach()` routine to it in the `ZZZ_install()` routine. We also need values for vendor ID (`ZZZ_VEN_ID`) and device ID (`ZZZ_DEV_ID`) for the comparison in `ZZZ_attach()`.

```
static int (*ZZZ_saved_attach)();
int ZZZ_VEN_ID = value
    /* these should be initialized */
int ZZZ_DEV_ID = value
    /* these should be initialized */
```

The `drv_ops_t` structure specifies the “external” driver routines to the kernel. The flags specify that the driver should be called on all device closes and that it is MP safe. See the “The `drv_info_t` Structure Type” section in Chapter 4, “Writing a Driver,” for further details.

```
static drv_ops_t ZZZ_ops =
{
    ZZZ_open, /* open */
    ZZZ_close, /* close */
    NULL, /* strategy */
```
The `drv_info_t` structure specifies the driver's name and class. The flags specify that the driver is character type and MP safe and that the configuration, including major number, should be saved and retained across reboots. See the “The `wsio_drv_data_t` Structure Type” section in Chapter 4, “Writing a Driver,” for further details.

```c
static drv_info_t ZZZ_info =
{
    "ZZZ",         /* name */
    "blender",     /* class */
    DRV_CHAR | DRV_SAVE_CONF | DRV_MP_SAFE, /* flags */
    -1,            /* block major number (-1 for dynamic) */
    -1,            /* character major number (-1 for dynamic) */
    NULL,          /* reserved */
    NULL,          /* reserved */
    NULL /* reserved */
    C_ALLCLOSES | C_MGR_IS_MP /* flags */
};
```

The `wsio_drv_info_t` structure gives WSIO Services additional information about the driver. The entries specify the driver's interface type, that it is an interface (or monolithic) driver, and that it conforms to the Release 10.0 I/O specifications. See Chapter 4, “Writing a Driver,” for further details.

```c
static wsio_drv_data_t ZZZ_data =
{
    "blender",        /* matches class name for T_INTERFACE drivers */
    T_INTERFACE,      /* drv_type - either T_DEVICE or T_INTERFACE */
    DRV_CONVERGED,    /* drv_flags */
    NULL,             /* optional function */
    NULL /* optional function */
};
```

The `wsio_drv_info_t` structure ties the preceding three structures together into a single structure used in the `ZZZ_install()` routine's call to `wsio_install_driver()`. See Chapter 4, “Writing a Driver,” for further details.

```c
static wsio_drv_info_t ZZZ_wsio_info =
{
    &ZZZ_info,
    &ZZZ_ops,
    &ZZZ_data
};
```
Sample WSIO Routines

Sample driver_install() Routine

A driver’s `driver_install()` routine registers the driver and its structures with WSIO Services and the I/O subsystem. It also links the driver’s `driver_attach()` function into the appropriate attach chain.

(If the device had a `driver_dev_init()` function, we would link it similarly into the dev_init chain here.)

NOTE

The name given to this routine is restricted. It must begin with the name of the driver, for example, `ZZZ`, and end with `install`, as in `ZZZ_install`. See “Step 1: Choosing a Driver Name” in Chapter 4, “Writing a Driver.”

```c
int ZZZ_install(void)
{
    int ret;

    /*
     * Register our driver information with WSIO services
     */
    ret = wsio_install_driver(&ZZZ_wsio_info);

    if (ret) {
        /*
         * If the install worked,
         * link ourselves into the pci_attach chain
         */
        ZZZ_saved_attach = pci_attach;
        pci_attach = ZZZ_pci_attach;
    }

    /*
     * Exit, returning the value we got
     * from the wsio_install_driver() call
     */
    return ret;
}
```

Sample driver_attach() Routine

For interface and monolithic drivers, the `driver_attach()` routine is linked into the global attach list for PCI drivers in the `driver_install()` routine.

A driver’s `driver_attach()` routine is called whenever the system finds a piece of hardware it thinks the driver might want to claim (this driver put its `driver_attach()` function on the `pci_attach` chain, so the system calls it every time a new PCI device is found). The `driver_attach()` routine first checks to see if this is the type of hardware it can claim, then claims it if it wants it and performs whatever initialization the card requires.

- PCI Services will NOT enable a PCI device or function response to memory accesses, I/O accesses, or PCI device or function mastering of the bus. This ensures that a PCI device or function remains completely disconnected from the bus until after driver initialization. It is the responsibility of the driver to do this, as shown in the following sample attach routine.
The driver is responsible for ensuring that the contents of a memory or I/O base register are not zero. All zeros indicates that either the specified configuration space register is not implemented by the PCI device or function or that the system could not find the resources to map the corresponding space into the system. If alternate register mappings exist, and those base registers are not zero, it is acceptable for the driver to use those mappings instead.

The driver must enable memory access, I/O access, and DMA.

```c
int ZZZ_pci_attach(uint32_t parm, struct isc_table_type *isc)
{
    uint8_t rev_id;
    uint16_t command_reg;
    uint32_t base_addr;
    PCI_ID *id = (PCI_ID *)&parm /* for LP64 */

    /*
    * see if this is our card
    */
    if (!(id->vendor_id == ZZZ_VEN_ID &&
         id->device_id == ZZZ_DEV_ID)) {
        goto exit0;
    }

    /*
    * If we use a standard bus interface chip we need to
    * check subsystem vendor ID and subsystem ID here
    * to make sure that our driver should be the
    * driver claiming this device
    */

    /*
    * Get the card revision
    */
    pci_read_cfg_uint8_isc(isc, PCI_CS_REV_ID, &rev_id);

    /*
    * We must check the isc->if_reg_ptr
    * before we use it. If it's NULL,
    * we read our base register and map it ourselves.
    * But if isc->if_reg_ptr isn't NULL, PCI
    * services already did the mapping work for us
    */
    if (isc->if_reg_ptr == NULL) {
        /*
        * We need to map our own base address.
        * Save the value in if_reg_ptr.
        * Get our physical base memory address.
        * For ZZZ, memory is at reg 0x10
        */
        pci_read_cfg_uint32_isc(isc, 0x10, &base_addr);

        /*
        * make sure we have a memory BAR
        * instead of an IO BAR
        */
        if (base_addr & 0x01) {
            printf("ZZZ - no memory BAR\n");
            goto exit0;
        }
    }

    /*
    * Mask off the bottom four bits of the PCI
    * memory base register (see PCI spec for
```
base_addr &= ~0xf;

/*
 * Ensure this base register was mapped in by the
 * system. If base_addr is 0, then the system
 * was unable to allocate us PCI memory space at all.
 */
if (base_addr == 0) {
    goto err0;
}

/* Get a virtual translation for card registers.
 * Assume there are 512 bytes of registers.
 * Save the value in if_reg_ptr.
 */
if ((isc->if_reg_ptr = map_mem_to_host(isc, base_addr, 512)) == NULL){
    goto err0;
}

/* Use if_reg_ptr to access the registers.
 * Enable memory access and bus mastering
 * (note: other bits in the register must be preserved)
 */
pci_read_cfg_uint16_isc(isc,PCI_CS_COMMAND,&command_reg);
pci_write_cfg_uint16_isc(isc, PCI_CS_COMMAND,
    command_reg | PCI_CMD_MEM_SPACE | PCI_CMD_BUS_MASTER);

/* Set up our interrupt handler.
 * Note that -1 is the third argument to isrlink().
 */
if (isrlink(isc, ZZZ_isr, -1, isc, 0) < 0) {
    goto err1;
}

/* set up our init routine to be run later
*/
CONNECT_INIT_ROUTINE(isc, ZZZ_if_init);

/* If everything okay, claim this card
*/
isc_claim(isc, &ZZZ_wsio_info);

/* Exit without error
*/
goto exit0;

err1:
/*
 * clean up the mapping
 */
unmap_mem_from_host(isc, isc->if_reg_ptr, 512);

err0:
/*
 * indicate that we had an error
 */
PCI_ATTACH_DEV_INIT_ERROR(isc);
exit0:
    /*
     * Always exit by calling rest of chain
     * Use link established in ZZZ_install()
     */
    return ZZZ_saved_attach(parm, isc);
}

Other Driver Entry Point Routines

The other routines defined by the code above must also be declared and written. These functions include the following list:

- **ZZZ_if_init()**: Initialization of the card after the `driver_attach()` routine
- **ZZZ_isr()**: The driver's interrupt service routine
- **ZZZ_open()**: The `drv_ops_t`-defined entry point for `open()`.
- **ZZZ_close()**: The `drv_ops_t`-defined entry point for `close()`.
- **ZZZ_read()**: The `drv_ops_t`-defined entry point for `read()`.
- **ZZZ_write()**: The `drv_ops_t`-defined entry point for `write()`.
- **ZZZ_ioctl()**: The `drv_ops_t`-defined entry point for `ioctl()`.

The code for these functions is driver dependent. See “Step 6: Writing Entry Point Routines” and “Step 7: Writing Other Driver Routines” in Chapter 4, “Writing a Driver.” See also `close (2)`, `ioctl (2)`, `open (2)`, `read (2)`, `write (2)`.
15 On-Line Addition/Replacement

The On-Line Addition/Replacement (OLA/R) is a required feature for high-availability servers. This chapter describes writing I/O device drivers on HP-UX 11i for OLA/R support. This chapter mainly focuses on additions to WSIO interface, modifications required to drivers for supporting OLA/R, and handling OLA/R events at the driver. Pseudo code is given wherever required for a better understanding of the concepts under discussion.

In the examples and pseudo code present in this chapter “driver_name” is used. This can be replaced with your driver name when you actually develop a driver for OLA/R support.

The reader of this chapter is assumed to have a good understanding of:

- HP-UX I/O Subsystem
- WSIO driver development environment
- Writing a WSIO driver
- High-availability issues
- Application level impact on OLA/R of a driver of interest
Introduction

The ability to insert device controller cards and replace such cards while a system is being used, without the interruption of services to users not directly affected by the device resource, is a necessary capability for high-availability machines. On PCI based HP 9000 servers, this capability is provided as On-Line Addition/Replacement (OLA/R) of I/O cards. Support for OLA/R is implemented at — hardware, firmware and software. Specified here are the implementation details to provide support for OLA/R in the driver software of an I/O card.

The major functional areas included in this chapter are:

- Overview of the required modifications to the driver for OLA/R support.
- The additions to the WSIO interface for OLA/R support.
- Details of the enhancement/modification steps for OLA/R.
- Details of handling OLA at the driver, MP-safe issues, resource allocation failure issues, and suspend and resume event handling for OLR.
- Miscellaneous code changes at the driver for OLA/R.
- Performing OLA/R of PCI I/O cards.

Supported Hardware and Software

Servers on which OLA/R is supported:

- L-Class, N-Class

Firmware Updates:

- Any required f/w updates

OS Version:

- HP-UX 11i and above
Overview of Driver Modifications Required for OLA/R

A driver can be enhanced or developed with OLA/R support by following these three steps:

1. Register a generic event handler. To support OLA/R functionality, each driver has to register an event handling function with the WSIO. WSIO calls this event handler to service OLA/R events like suspend and resume on the I/O adapter card the driver controls. The generic event handling function is registered in the driver install routine.

2. Register a capability mask. The driver has to register an event mask with the WSIO. The event mask specifies the capabilities of the driver in OLA/R event handling. This allows the same driver to support various capabilities on different instances of I/O adapter. The event capability mask is registered in the driver attach routine.

3. Driver Event Handler. To support OLA/R functionality in the driver, the driver has to handle suspend and resume events that are generated by the WSIO. On-Line Addition (OLA) does not generate any event. However since OLA can be done at any time once the machine is up and running, the attach and init routines of the driver must be made MP-safe and should back out of any errors during allocation of resources.

Detailed discussion on how to implement these three steps are present in later sections.

Since the OLA/R enhancements are not in the driver's main performance path, there should not be any performance penalty on the driver's regular data path.
WSIO Interfaces

WSIO provides a new set of data structures and services that allow driver enhancements for OLA/R support.

**wsio_drv_info_t**

A new field is added to the `wsio_drv_info_t` structure indicating the version of WSIO. The new `wsio_drv_info_t` structure is given:

```c
typedef struct wsio_drv_info {
    drv_info_t * drv_info;
    drv_ops_t * drv_ops;
    wsio_drv_data_t * drv_data;
    unsigned int driver_version;
} wsio_drv_info_t;
```

A driver would then use the following definition when it calls `wsio_install_driver`, `isc_claim` and `wsio_install_drv_event_handler`.

```c
static wsio_drv_info_t driver_name_wsio_drv_info = {
    &driver_name_info,
    &driver_name_ops,
    &driver_name_wsio_drv_data,
    WSIO_DRV_CURRENT_VERSION
};
```

The macro `WSIO_DRV_CURRENT_VERSION` is defined in `wsio.h`.

**WSIO Event Handling Structures**

WSIO and the driver that supports OLA/R interact through events and event completion callbacks. A new event data structure is defined in WSIO:

```c
typedef enum {
    WSIO_EVENT_SUSPEND   = 1<<0,
    WSIO_EVENT_RESUME    = 1<<1,
    WSIO_EVENT_REMOVE    = 1<<2,
    WSIO_EVENT_DEV_ERROR = 1<<3,
    WSIO_EVENT_BUS_ERROR = 1<<4,
    WSIO_EVENT_SELF_TEST = 1<<5,
} wsio_event_t;
```

For OLA/R functionality, only the first two events `WSIO_EVENT_SUSPEND` and `WSIO_EVENT_RESUME` are of interest.

```c
typedef unsigned int wsio_event_id_t;
```

The `event_id` is a tag to identify a call to a driver’s event handler and its completion. A driver will return it when it calls the completion call back. It is a number that WSIO uses to match up requests with replies. The callback function is defined as:

```c
typedef int (*generic_complete_callback_t)(
    struct isc_table_type *,
    wsio_event_id_t,
    void *
);
```

The event status is returned to WSIO using the third argument (void *) in the above `generic_complete_callback` definition. Although it is defined as a pointer, WSIO expects only one of the WSIO status values described:
typedef enum {
    WSIO_OK = 0,
    WSIO_ERROR = -1,
    WSIO_INFO_NULL = -2,
    WSIO_HANDLER_NULL = -3,
    WSIO_DRV_NOT_FOUND = -4,
    WSIO_INVALID_ISC = -5,
    WSIO_INVALID_EVENT = -6,
    WSIO_NO_DRV_HANDLER = -7,
    WSIO_INVALID_COMBIN_EVENTS = -8,
    WSIO_UNSUPPORTED_EVENT = -9,
    WSIO_HA_NA = -10, /* Not an OLA/R or HA capable system */
    WSIO_DRV_FUNC_NULL = -11,
    WSIO_UNKNOWN_FUNC_TYPE = -12,
} wsio_ret_code_t;

The generic event structure `wsio_generic_event_t` is defined as follows:

```c
typedef struct wsio_generic_event {
    wsio_event_t event; /* suspend, resume, and so on */
    wsio_event_id_t event_id;
    struct isc_table_type *isc;
    generic_complete_callback_t wsio_completion_cb;
    void *arg;
} wsio_generic_event_t;
```

The `wsio_generic_event_t` is passed to the driver event handler function. A driver event handler is a function that WSIO will call when an event, such as suspend, occurs.

The driver’s event handler is defined as:

```c
typedef void (*wsio_drv_event_handler_t) (wsio_generic_event_t *);
```
Driver Modifications for OLA/R Support

The driver must perform the following four steps to support OLA/R functionality:

1. Register its event handler.
2. Register its event capability mask.
3. Set the suspend and resume event timeout values.
4. Driver's event handler routine.

Driver Registering Its Event Handler

The driver registers a generic event handler function by calling `wsio_install_drv_event_handler`. This should be called after installing the driver by calling `wsio_install_driver` in the driver's install routine. This service is passed with two arguments:

```c
int wsio_install_drv_event_handler(
    wsio_drv_info_t * drv_info,
    wsio_drv_event_handler_t event_handler);
```

This is called in the `driver_install` routine. Pseudo code for a typical driver install routine is as follows:

```c
int driver_name_install(void)
{
    if (return_value = wsio_install_driver(
            &driver_name_drv_info)) {
        saved_attach = pci_attach;
        pci_attach = driver_name_attach;
    } else { /* Install Failure */
        return 0;
    }

    /* driver_name_event_handler() is the generic event */
    * handler implemented in the driver. */
    if (wsio_install_drv_event_handler(
            kdriver_name_drv_info,
            driver_name_event_handler) != WSIO_OK) {
        /* Driver not registered its event handler.
            * Driver’s normal operation may not be affected.
            */
    }
    return return_value;
}
```

Driver Registering Its Event Mask

This entry point is for a driver to call inside its attach routine after calling `isc_claim()`. There will be two parameters for this call: `isc` and event mask. Event mask is a `uint64_t` representing a possibility of 64 operations associated with the `isc`.

```c
typedef uint64_t wsio_event_mask_t; /* This is an OR of
    wsio_event_t described in last section */

int wsio_reg_drv_capability_mask(
```
struct isc_table_type *isc,
wsio_event_mask_t event_mask);

The following is pseudo code for a typical driver attach routine:

driver_name_attach()
{
    ....
    wsio_event_mask_t driver_name_event_mask = 0;
    /* There is no specific event for OLA. OLA does not use
     * the driver event handler. It uses the normal attach
     * and init routines. To support OLR, the driver
     * requires to handle two events - suspend and resume.
     */
    driver_name_event_mask = WSIO_EVENT_SUSPEND |
        WSIO_EVENT_RESUME;
    .... normal attach processing ....
    isc_claim(isc, &driver_name_wsio_drv_info);
    /* Register driver event capability mask.
     * NOTE: This should be called only after
     * isc_claim() is called.
     */
    return_value = wsio_reg_drv_capability_mask(isc,
        driver_name_event_mask);
    if (return_value != WSIO_OK && return_value !=
        WSIO_HA_NA) {
        /* Registering driver event mask failed.
         * Continue with normal processing.
         */
    }
    .... normal processing ...
}

Driver Event Timeout Values

When WSIO issues an event request to a driver by calling the driver's event handling function, it expects the
driver to complete the event, and to call the callback function within a specified period of time. If the driver
fails to complete the event within that period of time, the event goes into a timeout state. It is undesirable for
any event to enter a timeout state, so a driver must set the timeout value for each event such that the timeout
period will never be exceeded. However, if a driver times out for some reason during a suspend or resume
event it still can reply anytime later to change the status. The default timeout limit is ten seconds. If a driver
needs more than this to complete an OLA/R event, it should set the timeout limit to the required value using
the WSIO service wsio_set_parm. The current value of a specific parameter can be obtained from
wsio_get_parm.

The data structure wsio_parm_t defines the valid parameters for wsio_set_parm() and wsio_get_parm().

typedef enum {
    WSIO_HW_SUSPEND_TIMEOUT, /* Hardware Suspend Timeout param */
    WSIO_HW_RESUME_TIMEOUT, /* Hardware Resume Timeout param */
    WSIO_HW_REMOVE_TIMEOUT, /* Hardware Remove Timeout param */
    WSIO_HW_ERROR_TIMEOUT, /* Hardware Error Timeout param */
    WSIO_IDENTIFY_CHILD,    /* Function that identifies if a
                             * child is an interface */
}...
WSIO service `wsio_set_parm()` takes three arguments.

```c
int wsio_set_parm (struct isc_table_type * isc,
                   wsio_parm_t parm,
                   void * value);
```

Even though the third parameter “value” is declared as a pointer to a void, WSIO will read it as a value and NOT as a pointer. So do not use a pointer to a value; instead, use a defined value. The value specified is in microseconds.

This can be called at any place in the driver.

In the following example, the timeout values are set in the driver attach routine after registering the driver event capability mask.

Pseudo code for a typical attach routine is shown:

```c
{ 
    driver_name_attach()
    
    ubit32_t olar_timeout = 15000000; /* timeout set to 15 sec */
    ...
    ... normal processing ...
    
    isc_claim();
    
    if(wsio_reg_drv_capability_mask(isc,
                                    event_mask) == WSIO_OK) {
        /* Set the suspend and resume events timeout period */
        if(wsio_set_parm(isc, WSIO_HW_SUSPEND_TIMEOUT,
                        (void *) olar_timeout) != WSIO_OK) {
            wsio_reg_drv_capability_mask(isc, 0);
        }
        else if(wsio_set_parm(isc, WSIO_HW_RESUME_TIMEOUT,
                               (void *) olar_timeout) != WSIO_OK) {
            wsio_reg_drv_capability_mask(isc, 0);
        }
        ...
        ... normal processing ...
    }
}
```
Event Handling Function

WSIO generates an event to a driver when it receives an OLA/R request (from SAM or rad) on an I/O card the driver controls. WSIO calls the driver's event handling function which the driver has registered with the WSIO during the driver's installation time. It is the driver's responsibility to make sure that an OLA/R event is never timed out. OLA does not generate any event and follows the normal attach and init routines path. So the events supported by the driver would be for suspend and resume.

The following is pseudo code for a typical driver event handling routine:

```c
void
driver_name_event_handler(wsio_generic_event_t *handler_arg) {

    /* Switch based on the event The timeout calls
     * in the switch cases below are to allow the
     * event handler to return to WSIO immediately.
     * The timeout value of 0 is used in those cases
     * where there is no need to delay execution of the
     * individual event handler.
     * /
    switch (handler_arg->event) {

        case WSIO_EVENT_SUSPEND:
            /* Save callback function and other info like
             * event_id
             */
            call timeout(driver_name_suspend,
                        driver_name_suspend_info, 0);
            break;

        case WSIO_EVENT_RESUME:
            /* Check if it is a like-for-like card
             * Save callback function and other info like
             * event_id
             */
            call timeout(driver_name_resume,
                        driver_name_resume_info, 0);
            break;

        default:
            handler_arg->wsio_completion_cb(handler_arg->isc,
                                            handler_arg->event_id, WSIO_UNSUPPORTED_EVENT);
            break;
    }

    return;
}
```
Handling OLA/R

This section describes how **On-Line Addition (OLA)** and **On-Line Replacement (OLR)** have to be handled when designing the driver.

For detailed information on performing OLA/R operations, refer to the user manuals. The following information is only a description of the sequence of required operations necessary to write a driver with OLA/R support.

**OLA**

On-Line Addition of a card instance can be performed with the following steps:

1. Power-off the slot from **System Administration Manager** (SAM) or `rad`.
2. Insert the card into the slot.
3. Power-on the slot from SAM or `rad`.
4. Run `ioscan`, optionally passing the H/W path of the slot.

As mentioned earlier, On-Line addition of an I/O card does not generate any WSIO event that the driver event handler will handle. Instead, the driver's `attach` and `init` routines are called, just as they would be during the boot. There are two issues that a driver should handle to support OLA. They are:

- Driver's `attach` and `init` routines should be MP-safe
- Driver should handle resource allocation failures during the `attach` and `init`.

**MP Safe**

The `attach` and `init` routines must be MP-Safe; an OLA can occur at any time once the machine is up and running. If the driver has any global resources which are common to all instances, it is advisable to allocate them during the driver install time. If required global locks can be acquired and released during `attach` and `init` routines to serialize access and/or to protect data.

**Resource Allocation Failures**

Since an OLA can occur at any time once the machine is up and running, resource allocation may fail because of resource shortage problem. So the driver should be able to retrace all the steps and release all the resources that are allocated up until the failure occurred. This will take the driver back to a clean state.

To facilitate the back out of a driver because of a failure during `attach` and `init`, a list of resource allocation services and their corresponding releasing services are given. A word of caution is, release or free the resources in LIFO order; i.e., the most recently allocated resources are released first.

<table>
<thead>
<tr>
<th>Resource Allocation/Acquire</th>
<th>Resource Free/Release</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>kmalloc</code></td>
<td><code>kfree</code></td>
</tr>
<tr>
<td><code>wsio_map</code></td>
<td><code>wsio_unmap</code></td>
</tr>
<tr>
<td><code>wsio_map_port</code></td>
<td><code>wsio_unmap_port</code></td>
</tr>
<tr>
<td><code>wsio_allocate_shared_memory</code></td>
<td><code>wsio_free_shared_memory</code></td>
</tr>
</tbody>
</table>
All drivers must provide a return value from the `init` functions. The values must be either:

- **WSIO_OK** (0) on success.
- **WSIO_ERROR** (-1) on failure.

Refer to the ENET sample driver for an implementation example.

**OLR**

To replace a bad I/O card when the system is on-line, OLR can be used. Usually the following steps are followed while OLR of an I/O card:

1. Suspend the driver instance of the I/O card from SAM or rad.
2. Power-off the slot from SAM or rad.
3. Remove the card.
4. Insert a new I/O card.
5. Power-on the slot from SAM or rad.
6. Resume the driver instance of the I/O card from SAM or rad.

When writing a driver to support OLR, the driver should handle the events that happen during steps 1 and 6. WSIO guarantees that suspend and resume requests will be single-threaded. Also, the underlying OLA/R infrastructure guarantees that no suspend or resume events are triggered to the same card instance until the previous event processing is completed. For OLR, the underlying infrastructure does not trigger a resume event if the replacement card is outright incompatible. The driver is expected to do a more detailed check, like comparing the PCI sub-system ID etc., before claiming the replacement card.

The following WSIO OLA/R device driver state diagram in Figure 15-1, “OLA/R State Diagram,” shows the driver states from the perspective of WSIO. The internal driver states will differ from driver to driver. The timeout states in the diagram should not occur if the driver sets an appropriate value in the corresponding timeout values using `wsio_set_parm()`. Since recovery from the timeout states cannot be fully defined, it is very important that these states are never reached.
Figure 15-1  OLA/R State Diagram

The numbered state transition labels in the figure do not represent a sequence. The transitions are as follows:

- Normal suspend sequence is 1, 2.
- If a suspense fails, the sequence is 1, 5.

**NOTE**  A SUSPEND MUST NEVER FAIL.

- If a suspend timeout occurs, the sequence is 1, 6. From the SUSPENDING TIMEOUT state, 7 can occur if the driver calls the callback function with a status of `WSIO_OK` after a timeout has occurred. Transition 8 can occur if the driver calls the callback function with a status of `WSIO_ERROR` after the timeout has occurred.
- The normal resume sequence is 3, 4.
- If a resume fails, the sequence is 3, 9. From the RESUMING TIMEOUT state, 11 can occur if the driver calls the callback function with status `WSIO_OK` after a timeout has occurred. Transition 12 can occur if the driver calls the callback function with a status of `WSIO_ERROR` after the timeout has occurred.
**Suspend**

The sequence of events during a suspend is as follows:

1. Wait for the correct state. Definition of the “correct state” depends on a driver. But speaking in broad terms, no critical operation should be in progress like device reset, blocked on I/O request, etc. Pseudo code to perform this follows:

   ```c
   driver_name_event_handler(handler_arg)
   {
       ....

       switch(handler_arg->event) {
           ....

           case WSIO_EVENT_SUSPEND:
               if(driver_defined_state == DRIVER_DEFINED_RESET ||
                   driver_defined_state == DRIVER_DEFINED_IOBLOCK
                   ...) {
                   /* Set a timeout for suspend routine */
                   timeout(driver_name_suspend_handler,
                           suspend_info, PREDEFINED_WAITING_TIME);
               } else {
                   /* Call suspend handler. No need to wait here */
                   timeout(driver_name_suspend_handler,
                           suspend_info, 0);
               }
               break;
           ....
       }
       ....
   }
   ...
   }
   
   If the driver can not set any predetermined time as shown previously, the driver’s suspend handler can timeout itself for some number of clock ticks until the driver comes out of its critical operation state. Pseudo code to perform this follows:

   ```c
   driver_name_suspend_handler(suspend_info)
   {
       ...

       if(driver_defined_state == DRIVER_DEFINED_RESET ||
           driver_defined_state == DRIVER_DEFINED_IOBLOCK
           ...) {
           /* Set a timeout for suspend routine */
           timeout(driver_name_suspend_handler,
                   suspend_info, HZ);
           return;
       }
       ...
   }
   ```
A suspend request should never occur while a driver is suspending or suspended. If one occurs, the driver should immediately call the callback function with a status of `WSIO_INVALID_EVENT`. While a driver is suspending or suspended, some control requests, such as a reset or abort (or any other request which would normally cause the driver to enter a different state, or interact directly with the adapter) should be rejected. In these cases, the control request path must check for the SUSPENDED state (or equivalent) as well as any state the driver may be in during suspension for certain requests. These are just guidelines and what to do while handling a control request during and after a suspend operation is driver dependent.

2. If the driver has any timers, (other than timers specifically used to time the suspension sequence), the timers must be cancelled using `untimeout()`.

3. Quiesce the device. This is driver dependent. After this, the device is not expected to perform any I/O operations or generate interrupts to the driver.

4. Save the required device information for a following resume operation. For example, when only like-for-like replacement is allowed during resume, comparing the vendor ID, subsystem ID, etc., can be used in identifying a suitable replacement I/O card.

5. Save any other required information. This is driver dependent. Generally speaking, this information would include the required state or configuration of the device the replacement I/O card should be brought to.

6. Call the callback function with a success.

Refer to the ENET sample driver for an implementation example.

**Resume**

A resume request should only occur when the driver is suspended, and not currently in the process of resuming. If the resume event occurs at any other time, the driver should call the callback function with a status of `WSIO_INVALID_EVENT`.

If resume fails, the driver must return to the SUSPENDED state, and the callback function must be called with a status of `WSIO_ERROR`.

While a driver is resuming, some control request, such as a reset or abort (or any other request which would normally cause the driver to enter a different state, or interact directly with the adapter) should be rejected. Therefore, the control request path must check for the RESUMING state (or equivalent) as well as any state the driver may be in during resuming for certain requests. These are just guidelines and what to do while handling a control request during and after a resume operation is driver dependent. The sequence of events during a resume is described:

1. Check for a like-for-like replacement. Currently, the PCI CDIO checks for Vendor and Device IDs. If like-for-like replacement conditions do not meet the requirements of the driver, the driver must return to the SUSPENDED state, and the callback function must be called with a status of `WSIO_ERROR`.

2. Initialize the device. Again, this is driver dependent and what is part of initialization depends on the device and the driver.

3. Restore the state of the device if required. This is driver dependent.

4. Configure the device with the information saved during the suspend. This is driver dependent.

5. Call the callback function with `WSIO_OK` status.
Pseudo code for a driver resume routine is as follows:

```c
driver_name_event_handler(handler_arg)
{
    ...

    switch(handler_arg->event) {

        ....

        case WSIO_EVENT_RESUME:
            /* Check if the driver is in suspended state*/
            /* Test of like-for-like replacement. */
            /* Call the driver resume handler */
            timeout(driver_name_resume_handler,
                    resume_info, 0);

            break;

        ...}
        ....

    }

    driver_name_resume_handler(resume_info)
    {
        ....

        /* Turn on PCI memory access and bus master
         * capability on host, if applicable */

        /* Initialize the device */

        /* Configure the device */

        /* Call callback function with WSIO_OK status */

        return;
    }

Refer to the ENET sample driver for an implementation example.
```
Miscellaneous Changes Required at the Driver

As mentioned earlier, while processing control path requests like device reset, configuring the device etc., checks are required to see whether the driver is in SUSPENDED or RESUME states, or in the process of suspending or resuming. A driver state flag can be included in the driver control structure to keep track of the driver's OLA/R states.

If the driver is in SUSPENDED or RESUME states, the request should be reject with ENXIO. A portion of pseudo code for a driver reset is given:

```c
driver_name_reset(reset_info)
{
    ....
    /* Addition state check */
    if(driver_olar_state == SUSPENDED) {
        /* Return ENXIO saying that the device is busy */
        return ENXIO;
    }
    ....
}
```
Performing OLA/R of PCI I/O Cards

HP-UX provides two interfaces to perform OLA/R operations on PCI I/O cards:

- SAM (GUI)
- /sbin/rad (commandline)

Currently, only commandline tool /sbin/rad can be used to issue OLA/R requests on I/O cards that are controlled by third party drivers. SAM will be extended in later releases with support for performing OLA/R operation on I/O cards that are controlled by third party drivers.

Refer to the user documentation on *Managing PCI cards with OLA/R* for a detailed description of the use of these tools.
On-Line Addition/Replacement
Performing OLA/R of PCI I/O Cards
16 Writing a DLKM Driver

This chapter describes how to set up a single-user or multiuser system. The following topics are discussed:

- “Managing Dynamically Loadable Kernel Modules”
- “Developing Dynamically Loadable Kernel Modules”
Managing Dynamically Loadable Kernel Modules

This section presents the concepts and procedures which are necessary to understand, configure, and manage Dynamically Loadable Kernel Modules (DLKMs).

This section is divided into the following three topical sections as shown in Table 16-1, “DLKM Topical Sections.”

Table 16-1 DLKM Topical Sections

<table>
<thead>
<tr>
<th>Topic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DLKM Concepts</td>
<td>This section provides an introduction to DLKM, important DLKM terms, and detailed technical DLKM concepts.</td>
</tr>
<tr>
<td>DLKM Tools</td>
<td>This section provides a summary of tools collectively known as the Kernel Configuration Tool Set which are used when installing, configuring, and managing DLKM modules.</td>
</tr>
<tr>
<td>DLKM Procedures</td>
<td>This sections presents the key DLKM procedures used in the three phases of managing DLKM modules: Preparation, Loading, and Maintenance.</td>
</tr>
</tbody>
</table>

This section focuses on configuring and managing loadable device drivers, as they constitute the majority of supported module types for HP-UX release 11.0 and later.

NOTE The HP-UX kernel infrastructure provides the ability to dynamically load and unload DLKM drivers. While the base set of drivers shipped with HP-UX release 11i are not DLKM enabled, many Independent Software Vendors (ISVs) are coding DLKM enabled drivers for the hardware they provide. Check the documentation that shipped with any 3rd-party drivers you have to determine if they are DLKM enabled.

DLKM Concepts

This section provides a conceptual overview of DLKM features and functionality by:

- Defining DLKM at a high level.
- Explaining terms and concepts essential to understanding DLKM.
- Describing how DLKM modules are packaged in HP-UX.
- Identifying the types of kernel modules currently supported by DLKM.
- Describing the advantages of writing kernel modules in DLKM format.
- Examining DLKM module functions and configuration parameters.
What is DLKM?

The *Dynamically Loadable Kernel Modules Infrastructure* is an HP-UX operating system feature that allows “DLKM Enabled” kernel modules to be dynamically loaded into, or unloaded from, the HP-UX kernel without having to re-link the entire kernel or reboot the system.

Previously, to install a new driver you had to edit the `system` file, run the `config` or `mk_kernel` commands to create a new kernel, shut down the system, and then bring the system back up before you could use the new driver.

The DLKM feature not only provides the infrastructure to load kernel modules into a running system, but it also allows a kernel module to be statically linked when rebuilding the kernel. Setting a flag in one of the DLKM module’s configuration files determines whether the module is to be configured as dynamically loadable or statically linked.

Important Terms and Concepts

The DLKM infrastructure allows kernel modules to be configured in a number of different ways. The following Table 16-2, “Important Terms and Concepts,” considers the different ways a kernel module can be configured and loaded, and clearly defines each as a term. It also clarifies the relationship between each term as seen by the HP-UX kernel.

<table>
<thead>
<tr>
<th>Term/Concept</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel Module</td>
<td>A Kernel Module is a section of kernel code responsible for supporting a specific capability or feature. For example, file system types and device drivers are kernel modules.</td>
</tr>
<tr>
<td></td>
<td>In the kernel configuration context, a kernel module may be viewed as an object that can be installed, removed, configured or built on a system, either statically or dynamically.</td>
</tr>
<tr>
<td></td>
<td>There are two categories of kernel modules:</td>
</tr>
<tr>
<td></td>
<td>• <em>Traditional Module</em></td>
</tr>
<tr>
<td></td>
<td>• <em>Modularly-packaged Module</em></td>
</tr>
<tr>
<td>Traditional Module</td>
<td>A Traditional Module is a Kernel Module whose configuration data has not been modularized and can only be statically linked to the kernel.</td>
</tr>
<tr>
<td></td>
<td>In the kernel configuration context, configuration information about Traditional Modules is maintained in the shared <code>master</code> and <code>system</code> files, and can only be accessed upon booting a kernel in which they have been statically-configured.</td>
</tr>
</tbody>
</table>
Modularly-packed Module

A Modularly-packed Module is a Kernel Module whose configuration data has been modularized (not shared with other kernel modules), which is a pre-requisite for DLKM-enabling the Kernel Module.

In the kernel configuration context, this means that the module uses its own master and system files (as opposed to the shared master and system files in which Traditional Modules are configured).

In order to be classified as a Modularly-packed Module, the module must contain its own master and system files, as well as an individual object file, mod.o, that implements the module.

A Modularly-packed Module can be dynamically loaded into the HP-UX kernel only if that module includes the module wrapper code and additional data structures.

For this reason, we place Modularly-packed Modules in two categories:

- **Static Modularly-packed Modules**
- **Loadable Modules (or DLKM Modules)**

The terms Loadable Module and DLKM Module are interchangeable.

Static Modularly-packed Module

A Static Modularly-packed Module is a Modularly-packed Module that can only be linked statically to the kernel.

In the kernel configuration context, this means that the module uses its own master and system files but does not contain the module wrapper code and additional data structures that provide the dynamic loading and unloading ability.

Loadable Module (DLKM Module)

A Loadable Module (or DLKM Module) is a Modularly-packed Module with the capability to be dynamically loaded into a running kernel.

In the kernel configuration context, this means that the DLKM module uses its own master and system files and contains the module wrapper code and additional data structures that provide the dynamic loading and unloading ability.

However, when a DLKM module is written with self-contained module wrapper code and packaged with module-specific master and system files, it can still be statically-configured into the kernel.

For this reason, we place Loadable Modules in two categories:

- **Synchronously-configured Loadable Module**
- **Dynamically-configured Loadable Module**
DLKM Module Packaging

The DLKM infrastructure specifies that:

- A kernel module must be packaged modularly with at least:
  - Its own master and system files.
  - Its own mod.o object file that implements only that module.

- The mod.o object file must contain the Module Wrapper code (although full optimization is optional).

**NOTE**

See the master (4) manpage for descriptions of the two kinds of master files, and the config (1M) manpage for a description of the traditional and modular system files.

Kernel modules written as traditional modules are still fully supported in HP-UX. Driver developers are encouraged to re-package their static modules according to the module packaging architecture introduced with DLKM modules.
DLKM Module Types

The DLKM feature currently supports the following types of kernel modules:

- WSIO class drivers
- WSIO interface drivers
- STREAMS drivers
- STREAMS modules
- Miscellaneous modules — for example, modules containing support functions not required in the statically-configured kernel but shared among multiple loadable modules

DLKM Advantages

DLKM modules provide many advantages relative to static modules, including:

- Reducing time spent on device driver development by streamlining the driver installation process.
- Making it easier for administrators to install device drivers from other vendors.
- Improving system availability by allowing device drivers and other modules to be configured into the kernel while the system is running.
- Conserving system resources by unloading infrequently used modules when not in use.
- Providing administrators with the ability to demand load and unload modules.
- Providing the kernel with the ability to automatically load modules.

Auto loading occurs when the kernel detects a particular loadable module is required to accomplish some task, but the module is not currently loaded. The kernel automatically loads the module.

DLKM Driver Loading Concepts

When a module is dynamically loaded, its object file is read from disk and loaded into newly allocated kernel memory. Once in memory, the module's symbols are relocated and any external references are resolved. Special code in the module is then executed to perform any required module-specific setup. Then the code specific to the module's type, if any, is executed, making the newly loaded module accessible to the rest of the kernel.

A module can be loaded in the following ways:

- **Demand Load**
  
  A demand load is a user level request for a specific module to be loaded. The load is accomplished through the `kmadmin` command.

- **Autoload Event**
  
  An autoload occurs when the kernel detects that a specific module is required to provide the functionality necessary to perform a task. The load is triggered by the initiation of the task. Once the required module is loaded, the task continues.

A loadable module's `_load()` function performs any initialization tasks required by the module before the module is logically connected to the kernel. Typical initialization tasks include acquiring private memory for the module and initializing devices and data structures.

- If the module is unable to initialize itself, the `_load()` function must free any memory that it allocated and undo any other action that it took prior to the failure including canceling all outstanding calls to `timeout`. 
DLKM Driver Unloading Concepts

When the functionality provided by a module is no longer needed the module can be unloaded, thus freeing its resources for later use.

- When a module is unloaded, the code specific to the module’s type, if any, is executed to disconnect the module from the kernel. Then, special code in the module is executed to perform any module-specific cleanup. Finally, the memory allocated to the module is freed.

- A module may be unloaded only by a user level request specifying the module to be unloaded. The unload is accomplished through the `kmadmin` command. This request may fail for a number of reasons, the most common being that the module is busy at the time. An example of this would be attempting to unload a device while there are outstanding opens on the device.

A loadable module’s `_unload()` function is called by the DLKM mechanism whenever the module is about to be removed from active memory. The function may be given any name (typically `module_name_unload`); a pointer to the `_unload()` function is obtained from the module’s wrapper.

- The module’s `_unload()` function cleans up any resources that were allocated to the module, and it must remove all references to the module. Typical cleanup tasks include releasing private memory acquired by the module, removing device interrupts, disabling interrupts from the device, and canceling any outstanding timeout requests made by the module.

- The module’s `_unload()` function returns 0 on success and an `errno` value on failure. In the event of failure, the function leaves the module in a sane state, since the module will remain loaded after the return.

- The system will never attempt to unload a module that it thinks is busy. However, the system cannot determine under all cases when the module is in use. Currently, a module is considered to be busy when another module that depends on it is also loaded. In addition, WSIO class drivers and STREAMS drivers track the `open()` and `close()` calls; these types of modules are busy whenever there is at least one open on the device using the driver. Under most other circumstances, the module determines for itself whether it is appropriate for it to be unloaded. When a module is still in use, its `_unload()` function returns a non-zero value to cancel the unload.

- The argument passed to the `_unload()` function is the same type-specific value that was passed to the module’s `_load()` function. The use of this argument is described in section “STREAMS Drivers”.

DLKM Driver Configuration Concepts

Since kernel modules written in the DLKM format can be configured as either dynamically loadable or statically-configured, DLKM-compatible device drivers must accommodate either configuration.

Through the use of configurable module attributes, System Administrators can control the various functions of a DLKM driver, including whether it is dynamically loaded or statically-configured.

This section provides attributes and keywords for:

- Required components of a DLKM driver.
- Optional components of a DLKM driver.

It also presents a brief description of STREAMS and Miscellaneous drivers. See the Table, “DLKM Tools,” section for detailed instructions on how to modify the configurable module attributes presented here.

NOTE

The system must be in a run-time state before dynamic module loading is available. Thus, kernel modules required during system boot must be configured as statically-configured.
Master File Definition

Each DLKM module has its own master file. The format of the master file includes the following section keywords:

- `$VERSION` — Indicates the version number for the file format. Version is defined as an integer and starts from one. A single line containing the only supported version (version 1) is entered.

- `$LOADABLE` — Indicates that the module supports dynamic loading. If this section keyword does not exist, the module can only be statically-configured into the kernel.

- `$INTERFACE` — Identifies the interface names and versions on which the module is built. For HP-UX, versions 11.0 and higher, a single line is entered containing the word base.

- `$TYPE` — Indicates the module type and the type specific information. Valid types are wsio_class, wsio_intfc, streams_mod, streams_drv, and misc.

- Other sections (if required) — `$DRIVER_DEPENDENCY`, `$TUNABLE`, and `$DRIVER_INSTALL`.
  
  The `$DRIVER_DEPENDENCY` section, defines the names of all other modules that this module depends upon.

  The `$TUNABLE` section defines the names and default values of the tunable parameters (variables) for the module. Default (and optionally minimum) values for tunable parameters are entered here.

  The `$DRIVER_INSTALL` section defines the module’s name and associated block and/or character major device number(s).

Modstub.o File Definition

An optional component, the Modstub.o file is statically-configured into the kernel as a “place holder” for functions implemented in a loadable module that will be loaded at a later time. Its purpose is to enable the kernel to resolve references to the absent module’s functions. Configuring a module that uses stubs requires a full kernel build so that the stubs can be statically linked to the kernel.

Modstub.o contains stubs for entry points defined in the associated loadable module that can be referenced by other statically-configured kernel modules currently configured in the system. Access to a stub causes the kernel to auto load the associated loadable module.

System File Definition

Every DLKM module requires a system file. The system file includes the following three mandatory and one optional section keywords:

- `$VERSION` — Indicates the version number for the file format. Version 1 is the only supported file-format.

- `$CONFIGURE` — Indicates if the module is to be configured into the system. If `$CONFIGURE` is Y or y, the module will be configured on the next build; if `$CONFIGURE` is N or n, the module will not be configured on the next build. `kmsystem` (1M) provides the interface to modify the flag.

- `$LOADABLE` — Indicates how the module will be configured. If `$LOADABLE` is Y or y, the module will be configured as a Dynamically-configured Loadable Module; if `$LOADABLE` is N or n, the module will be statically configured into the kernel, requiring a reboot. `kmsystem` provides the interface to modify the flag.

- If `$CONFIGURE` is N or n, `$LOADABLE` is ignored.

- `$TUNABLE` (empty) — Place holder for any tunable parameter specified in the associated master file for which to specify a value other than the default value. Nothing is entered here.

**NOTE** The version number for the master file and system file must be the same.

- `$CONFIGURE` — Indicates if the module is to be configured into the system. If `$CONFIGURE` is Y or y, the module will be configured on the next build; if `$CONFIGURE` is N or n, the module will not be configured on the next build. `kmsystem` (1M) provides the interface to modify the flag.

- `$LOADABLE` — Indicates how the module will be configured. If `$LOADABLE` is Y or y, the module will be configured as a Dynamically-configured Loadable Module; if `$LOADABLE` is N or n, the module will be statically configured into the kernel, requiring a reboot. `kmsystem` provides the interface to modify the flag.

- If `$CONFIGURE` is N or n, `$LOADABLE` is ignored.

- `$TUNABLE` (empty) — Place holder for any tunable parameter specified in the associated master file for which to specify a value other than the default value. Nothing is entered here.
Writing a DLKM Driver
Managing Dynamically Loadable Kernel Modules

`kmtune (1M)` is the interface to modify tunable parameters in the module's system description file and the HP-UX system file (`/stand/system` by default).

**space.h File Definition**

An optional component, the `space.h` file contains storage allocations and initialization of data structures associated with a DLKM module *when the size or initial value of the data structures depend on configurable values such as tunable parameters*. In order to communicate these values to the rest of the DLKM module, the values are stored in global variables and accessed by the module via external declarations in the module's `mod.o` file.

---

**NOTE**

All tunable parameters specified in the `master` file are defined as global variables in the `space.h` file.

---

**STREAMS Drivers**

Initialization of STREAMS drivers is very similar for both the loadable and statically-configured module cases. The only difference is that loadable drivers must use the `drv_info_t` structure that is passed as an argument to the `_load()` function.

STREAMS drivers, like WSIO class drivers, automatically track `open()` and `close()` system calls for the STREAMS device. The system will prevent a STREAMS driver from unloading whenever the device has one or more open file handles. Of course, the driver can still disallow an unload if this check is insufficient for its needs.

**Miscellaneous Modules**

Miscellaneous modules can implement any feature within the kernel. As such, a miscellaneous module's `_load()` function must address all of the module's specific needs. Similarly, the module's `_unload()` function must determine for itself if it is safe to unload. The system will not allow a module to be unloaded if other loaded modules are dependent upon the module. Other than this check, the system performs no other checks when the administrator attempts to remove a miscellaneous module from the kernel.

The argument to the `_load()` function is not meaningful and should be ignored.

**DLKM Tools**

There are a number of HP-UX commands known collectively as the *kernel configuration tool set* for installing, configuring, and managing DLKM modules. These commands are presented with descriptions and applicable command line options in this section.

**Why you should use the kernel configuration tools instead of manually editing the system files.**

Although the HP-UX static kernel environment has not changed, it is affected by the configuration of kernel modules within the DLKM infrastructure. Specifically, DLKM requires that a kernel module have its own `master` and `system` files, and contain a Module Wrapper.

To the overall HP-UX kernel configuration environment this means:

1. The configurable module information is distributed among several files:
   - traditional modules use the `/stand/system` file
   - modularly-packaged modules use their own module-specific system file
2. The kernel structure is extended:
   - Static kernel executable file `/stand/vmunix`.
   - Associated DLKM kernel components under `/stand/dlkm`:
     - kernel symbol table
     - dynamic loadable modules

Because of the effects that the DLKM infrastructure has on the overall kernel configuration environment, it is best to configure any type of kernel module using the tools described in this section.

**NOTE**  Avoid editing the `system` file, or replacing the kernel file manually, as doing so increases the chance of introducing configuration errors.

For more detailed information regarding the `master` and `system` files, refer to the `master (4)` manpage and the `config (1M)` manpages.

**Kernel Configuration Tools Description**

The system administrator uses the kernel configuration tools to install, configure, load, unload, update, or remove kernel modules from the system; and to build new kernels. Use the commands described in this tool set to configure kernel modules of any type (static or loadable).

The action carried out by a kernel configuration tool depends upon the options you specify during the tool's invocation. This information is presented in the *Commands and Options in the Kernel Configuration Tool Set* section.

The following list describes the basic function of each of the commands that make up the kernel configuration tool set.

**Tools for Building Static or Dynamic Kernels**

- **kmsystem (1M)**
  Provides interface to set a module's configurable attributes, to indicate whether a module should be configured, and whether it should be built as loadable or static.

- **kmtune (1M)**
  Provides interface to set the tunable parameters

- **kmupdate (1M)**
  Updates the system with the newly built kernel and/or associated DLKM files.

**Tools that Provide an Interface to DLKM**

- **kminstall (1M)**
  Install, remove or update a module's component files on a system.

- **kmadmin (1M)**
  Provides general administrative interface for DLKM. Allows administrators to load, unload and query loadable modules.
Commands and Options in the Kernel Configuration Tool Set

This section the command line options with descriptions for each of the kernel configuration tools.

**NOTE** If you need further information regarding the functionality, usage or command line options for any of the kernel configuration tools, refer to their respective manpages.

For additional information, refer to Table 16-3, “Kernel Configuration Tool Set.”

<table>
<thead>
<tr>
<th>Tool/Command</th>
<th>Action</th>
</tr>
</thead>
</table>
| config       | • First form — Generates both the static kernel and associated Dynamically-configured Loadable Modules; a system reboot is necessary.  
• Second form, -M option — Generates the specified loadable module for use with the currently running kernel. The newly configured service is available immediately, without requiring a system reboot. |
| kmadm        | • -k option — Prints a list of all statically-configured modules in the running kernel.  
• -L option — Loads the specified loadable module into the running kernel  
• -Q, -q option — Prints the status of the specified loadable module.  
• -S, -s option — Prints the status of all currently loaded or registered loadable modules.  
• -U, -u option — Unloads the specified loadable module from the running kernel. |
| kminstall    | • -a option — Adds a module’s component files to certain subdirectories of /usr/conf and /stand.  
• -d option — Deletes a module’s component files from the subdirectories of /usr/conf and /stand.  
• -u option — Copies a module’s updated component files into the subdirectories of /usr/conf and /stand. |
| kmsystem     | • -c option — Assigns a value (Y or N) to the configuration ($CONFIGURE) flag of the specified module in preparation for the next system configuration.  
• -l option — Assigns a value (Y or N) to the loadable ($LOADABLE) flag of the specified module in preparation for the next system configuration.  
• -q option — Prints the values of the configuration and loadable flags of the specified module. Prints a “-” (signifies “does not apply”) for the loadable flag of a static module.  
• No options or -S option only — Prints the values of the configuration and loadable flags of all modules. Prints a “-” for the loadable flags of static modules. |
Table 16-3 Kernel Configuration Tool Set (Continued)

<table>
<thead>
<tr>
<th>Tool/Command</th>
<th>Action</th>
</tr>
</thead>
</table>
| kmtune       | • `-l` option — Prints the values of all system parameters.  
|              | • `-q` option — Queries the value of the specified system parameter.  
|              | • `-r` option — Resets the value of the specified parameter to its default value in preparation for the next system configuration.  
|              | • `-s` option — Assigns a value to the specified system parameter in preparation for the next system configuration. |
| kmupdate     | • First form — Prepares the system to move the specified static kernel and its associated files to the `/stand/vmunix` file and `/stand/dlkm` directory, respectively, during the next system shutdown and startup.  
|              | • Second form, `-M` option — Moves the configured image of the specified loadable module to the location where the DLKM loader can find it, and registers the module with the kernel either (1) immediately or (2) later at system shutdown. |
| kmmodreg (1M)| Unregisters and re-registers loadable kernel modules with the running kernel.  
|              | (Note: The interface for the initial module registration of a newly configured loadable module is `kmupdate(1M)`, which in turn calls `kmmodreg(1M)`.)  
|              | • `-U` option — unregisters the specified loadable image.  
|              | • `-M` option — re-registers the specified loadable image. |

Why you should use the kernel configuration tools instead of manually editing or copying files.

Since the introduction of DLKM to HP-UX, the installed kernel configuration information is used to configure a whole kernel, or a specific module, and is distributed among several files. The number and location of the files used to configure a kernel depends on which modules are installed on the system. Furthermore, the format of such information is subject to change. Using the `kmsystem(1M)` and `kmtune(1M)` commands to modify the planned configuration prior to a whole kernel configuration or a module configuration will avoid corruption of the configuration files, and will ensure that the correct files are updated.

**CAUTION** Avoid editing the system files; doing so is not forward-compatible and increases the chance of introducing configuration errors.

In addition, a configured, bootable kernel is no longer composed of a single file. Each new kernel configured with the `config(1)` or `mk_kernel(1M)` command is composed of several files, including the main kernel file, the loadable images or DLKMs, a symbol table files, and a kernel-specific KRS file. All component files of the kernel are essential for the kernel to boot and run properly. The `kmupdate(1)` command understands the organization of these files and prepares a directory structure accessible to the boot loader that allows a newly generated kernel to boot and run properly.

A whole kernel configuration must be followed by a kernel update step initiated via the `kmupdate(1)` command or `(config -u)` in order to be able to successfully boot the new kernel.

Because of these effects, it is best to configure any type of kernel module using the tools described in this section.
CAUTION  Avoid copying or moving kernel files; doing so may result in corrupting the running kernel, or any other configured kernel on the system.

See ksysinfo(1M), kmtune(1M), mk_kernel(1M), config(1M), and kmupdate(1M) for more information.

DLKM Procedures for Dynamically Configured Loadable Modules

This section provides detailed procedures for configuring, loading, and unloading DLKM Enabled kernel modules. Procedural information is shown in three different ways. The first two are summary formats and the third provides detailed procedure steps.

1. DLKM Procedural Flowchart

   Use the chart shown in Figure 16-1, "DLKM Procedural Flowchart," as a reference to view all of the procedures and to determine the correct sequence in which to perform them.
Figure 16-1 DLKM Procedural Flowchart

Start

Dynamically-configured Loadable Module

Prepare module as Dynamically Configured Loadable Module using the command: `kmsystem -c Y -l Y`

Optional: Tune system parameter(s) supplied by module or static kernel using the command: `kmtrune -s`

Configure loadable module into system using command: `config -M`

Move loadable module’s image into place and register module using command: `kmupdate -M`

If necessary, create device special file(s) for loadable module using command: `mknod`

Load loadable module using command: `kmadmin -L`

Optional: Query loadable module using command `kmadmin -q`

Optional: Unload loadable module using command `kmadmin -U`

Optional: Remove module’s components from system using command: `kminstall -d`

Done

Statically-configured Loadable Module

Prepare module as Statically Configured Loadable Module using the command: `kmsystem -c Y -l N`

Configure statically linked module into system by building new kernel using command: `config /stand/system`

Prepare system to move new kernel into place during next system shutdown and startup using command: `kmupdate /stand/build/vmunix_test`

Activate statically linked module by booting new kernel using command: `shutdown -r`

Optional: Query statically linked module using command `kmadmin -k`

Optional: Remove module’s components from system using command: `kminstall -d`

Optional: Create device special file(s) for statically linked module using command: `mknod`

Done
Tables of Loadable Module Configuration and Management Procedures

These tables group the procedures into 3 phases: Preparing, Loading, and Maintaining procedures. There is one table for each Loadable Module type, refer to Table 16-4, “Dynamically-Configured Loadable Module Procedures,” and Table 16-5, “Statically-Configured Loadable Modules Procedures.”

Table 16-4  Dynamically-Configured Loadable Module Procedures

<table>
<thead>
<tr>
<th>Phase</th>
<th>Configuration Option</th>
<th>Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preparing</td>
<td>Prepare Loadable Module as a Dynamically-Configured Loadable Module.</td>
<td>Prepare a loadable module for dynamic loading into the HP-UX kernel.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Optional: Query and/or Tune the system parameters supplied by a loadable module.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Configure a loadable module for dynamic loading.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Register a Dynamically-Configured Loadable Module with the kernel.</td>
</tr>
<tr>
<td>Loading</td>
<td>Demand-Load</td>
<td>Load a Dynamically-Configured Loadable Module into the kernel.</td>
</tr>
<tr>
<td>Maintaining</td>
<td>Unload</td>
<td>Unload a Dynamically-Configured Loadable Module.</td>
</tr>
<tr>
<td></td>
<td>Tune</td>
<td>Tune a Dynamically-Configured Loadable Module.</td>
</tr>
<tr>
<td></td>
<td>Update a module</td>
<td>Update a Dynamically-Configured Loadable Module’s image.</td>
</tr>
<tr>
<td></td>
<td>Query a module</td>
<td>Determine which Dynamically-Configured Loadable Modules are currently loaded.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Obtain information about a loaded Dynamically-Configured Loadable Modules.</td>
</tr>
</tbody>
</table>
All DLKM modules that are required to boot the kernel must be configured as statically configured modules.

If the module you are configuring is required to boot the kernel, refer to the configuration procedure in the section Statically-configured Loadable Modules.

3. DLKM Procedures

This presents step-by-step instructions for preparing, configuring, loading and unloading (or activating) loadable modules.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Configuration Option</th>
<th>Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preparing</td>
<td>Prepare Loadable Module as a Statically-Configured Loadable Module.</td>
<td>Prepare a loadable module for static linking to the HP-UX kernel.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Optional: Query and/or Tune the system parameters for a Statically-Configured Loadable Module present in the Static Kernel.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Configure Kernel to include Statically-Configured Loadable Module.</td>
</tr>
<tr>
<td>Loading</td>
<td>Activate a Statically-configured Loadable Module.</td>
<td>Activate a Statically-Configured Loadable Module by rebooting.</td>
</tr>
<tr>
<td>Maintaining</td>
<td>Tune a module</td>
<td>Tune a loadable module.</td>
</tr>
<tr>
<td></td>
<td>Query a module</td>
<td>Determine which Statically-Configured Loadable Module are currently loaded.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Obtain information about a currently loaded Statically-Configured Loadable Module.</td>
</tr>
</tbody>
</table>

How to prepare a loadable module for dynamic loading into the HP-UX kernel.

Use the kmsystem command to assign values (Y or N) to the configuration ($CONFIGURATION) and loadable ($LOADABLE) flags in the module's system description file. If the loadable flag is not present in the system description file and you attempt to assign it a value, kmsystem exits with an error.

Use the kmsystem command to prepare a DLKM module for configuration as either (1) dynamically-configured or (2) statically-configured.

To prepare a loadable module to be dynamically loaded into the kernel, do the following:

**Step 1.** Execute this kmsystem command:

```
/usr/sbin/kmsystem -c Y -l module_name
```

**NOTE** A module’s system description file, as well as the HP-UX system file, must only be modified using the kmsystem or kmupdate commands. It is important to follow this practice because the format of planned attributes and their location in the system will change in future releases and if you manually modify the system files as they are currently designed, your modifications will not be forward compatible.
How to query and tune the system parameters supplied by a loadable module.

Use the `kmtune` command to query, set, or reset system (tunable) parameters used by the DLKM module or the static kernel. `kmtune` reads the master configuration files, the system description files, and the HP-UX system file.

For a Modularly packaged Module, `kmtune` writes any user-modified system parameter to the module’s system description file. For a Traditionally-packaged module using pre-11.0 module packaging, `kmtune` writes any user-modified system parameter to the HP-UX system file.

**Step 1.** To query the value of a specific system parameter, execute this `kmtune` command:

```
/usr/sbin/kmtune -q system_parameter_name
```

**Step 2.** To set the value of a specific system parameter, execute this `kmtune` command:

```
/usr/sbin/kmtune -s system_parameter_name=value
```

**Step 3.** To reset the value of a system parameter to its default value, execute this `kmtune` command:

```
/usr/sbin/kmtune -r system_parameter_name
```

At this point, the values of the module’s system parameters for the next module configuration have been set. The values of the system parameters supplied by the module will become effective with the running kernel after the loadable module is configured and registered (see the following procedures).

### How to configure a loadable module for dynamic loading.

Upon completing the configuration procedure shown here, the dynamically-configured loadable module will be ready to load immediately, meaning that you do not have to wait for a reboot to be able to load it.

**Step 1.** To configure a loadable module for dynamic loading, execute this `config` command:

```
/usr/sbin/config -M module_name -u
```

This results in the generation of a loadable image. The `-u` option forces `config` to call the `kmupdate` command, which causes the system to move the newly generated image into place and register it with the running kernel.

### How to register a dynamically-configured loadable module with the HP-UX kernel.

For a DLKM module configured as dynamically loadable, you use the `kmupdate` command to update its image and register it with the kernel. Updating a dynamically-configured loadable module’s image means moving its image into place and registering it with the kernel either (1) immediately or (2) later at system shutdown.

Call `kmupdate` after first calling `config`. If you include the `-u` option in the `config` invocation, there is no need to invoke `kmupdate`. The `config -M -u` command automatically invokes `kmupdate`.

**Step 1.** To update the image of a dynamically-configured loadable module immediately, execute this `kmupdate` command:

```
/usr/sbin/kmupdate -M module_name -i
```

After updating the specified module and assuming the module was loaded originally, `kmupdate` will reload the module before exiting.

**Step 2.** To update the image of a dynamically-configured loadable module at system shutdown, execute the following `kmupdate` command:

```
/usr/sbin/kmupdate -M module_name -a
```
Writing a DLKM Driver
Managing Dynamically Loadable Kernel Modules

If you do not specify the -i or -a option, kmupdate will attempt to update the specified loadable module immediately. If the module cannot be updated immediately (for example, the current module is in use and cannot be unloaded), the module will be updated at system shutdown.

How to load a dynamically-configured loadable module into the HP-UX kernel.

To load a dynamically-configured loadable module, you use the -L option of the kmadmin command. The load operation initiated by the kmadmin -L command performs all tasks associated with link editing the module to the running kernel and making the module accessible to the system.

Specifically, the load operation performs the following tasks:

- Checks what other modules the loadable module depends upon and automatically loads any such module that is not currently loaded.
- Allocates space in active memory for the specified loadable module.
- Loads the specified loadable module from the disk and link-edits it into the running kernel.
- Relocates the loadable module's symbols and resolves any references the module makes to external symbols.
- Calls the module's _load() entry point to do any module-specific initialization and setup.
- Logically connects the module to the rest of the kernel, which is often accomplished with the help of module type-specific installation functions accessed through the module's wrapper code.

**Step 1.** To load a dynamically-configured loadable module into the running kernel, execute the following kmadmin command:

```
/usr/sbin/kmadmin -L module_name
```

When the loading completes, an identifier (ID) number prints on the standard output to identify the module that was loaded.

If you want the system to automatically load certain dynamically-configured loadable modules immediately after every system reboot, add the names of the modules to the /etc/loadmods file. At boot time, the /sbin/init.d/kminit script will execute the kmadmin command and load the modules listed in /etc/loadmods.

How to unload a dynamically-configured loadable module.

Use the -U or -u option of the kmadmin command to unload a DLKM module configured as dynamically loadable. You have the choice of unloading the module by its name or its ID number.

The unloading operation logically disconnects the module from the running kernel and calls the module's _unload() entry point to perform any module-specific cleanup including:

1. Canceling all outstanding calls to timeout().
2. Disabling device interrupts.
3. Freeing all active memory allocated to the specified loadable module.

**Step 1.** To unload a dynamically-configured loadable module by name, execute this kmadmin command:

```
/usr/sbin/kmadmin -U module_name
```

**Step 2.** To unload a dynamically-configured loadable module by ID number, execute this kmadmin command:

```
/usr/sbin/kmadmin -u module_id
```
How to determine which modules are currently loaded

Use the -S or -s option of the kmadmin command to view detailed information about all current registered DLKM module.

Step 1. To print the full status for all dynamically-configured loadable modules currently registered, execute this kmadmin command:

```
/usr/sbin/kmadmin -S
```

Step 2. To print the brief status for all dynamically-configured loadable modules currently loaded, execute this kmadmin command:

```
/usr/sbin/kmadmin -s
```

Step 3. To print a list of all statically-configured modules, execute the following kmadmin command:

```
/usr/sbin/kmadmin -k
```

How to obtain information about a loaded module.

Use the -Q or -q option of the kmadmin command to view detailed information about the DLKM module. For a DLKM module configured as dynamically loadable, you have the choice of displaying information for the module by its name or ID number.

Step 1. To display a dynamically-configured loadable module's status by name, execute this kmadmin command:

```
/usr/sbin/kmadmin -Q module_name
```

Step 2. To display a dynamically-configured loadable module's status by ID, execute the following kmadmin command:

```
/usr/sbin/kmadmin -q module_id
```

Depending on the type of module, information on the module's block major number, character major number, and flags may also be printed.

Information returned by the -Q and -q options includes the module's:

- name
- ID
- pathname to its object file on disk
- status (LOADED or UNLOADED)
- size
- virtual load address
- The memory size of Block Started by Symbol (BSS) (the memory size of the un-initialized space of the data segment of the module's object file).
- base address of BSS
- Reference or hold count (the number of processes that are currently using the module).
- Unload delay value (currently not used — always 0 seconds).
- descriptive name
Writing a DLKM Driver
Managing Dynamically Loadable Kernel Modules

- type of module (WSIO, STREAMS, or Misc)

**DLKM Procedures for Statically Configured Loadable Modules**

**How to prepare a loadable module for static linking.**

Use the *kmsystem* command to prepare a DLKM module for configuration as either (1) *dynamically loadable* or (2) *statically-configured*.

Use the *kmsystem* command to assign values (Y or N) to the configuration ($CONFIGURE) and loadable ($LOADABLE) flags in the module’s system description file. If the loadable flag is not present in the system description file and you attempt to assign it a value, *kmsystem* exits with an error.

**Step 1.** To prepare a DLKM module for static linking to the HP-UX kernel, execute this *kmsystem* command:

```
/usr/sbin/kmsystem -c Y -l N module_name
```

**How to query and tune the system parameters for a statically-configured loadable module present in the static kernel.**

Use the *kmtune* command to query, set, or reset system (tunable) parameters used by the DLKM module or the static kernel. *kmtune* reads the master configuration files, the system description files, and the HP-UX system file.

For a Modularly-packaged module or a Traditionally-packaged module using 11.0 module packaging, *kmtune* writes any user-modified system parameter to the module’s system description file. For a Traditionally-packaged module using pre-11.0 module packaging, *kmtune* writes any user-modified system parameter to the HP-UX system file.

To query the value of a specific system parameter, do the following:

**Step 1.** Execute this *kmtune* command:

```
/usr/sbin/kmtune -q system_parameter_name
```

**Step 2.** To set the value of a specific system parameter, execute this *kmtune* command:

```
/usr/sbin/kmtune -s system_parameter_name=value
```

**Step 3.** To reset the value of a system parameter to its default value, execute this *kmtune* command:

```
/usr/sbin/kmtune -r system_parameter_name
```

At this point you have set the values of system parameters that will take effect after the next whole HP-UX kernel configuration, update and system reboot (see the following procedures).
How to configure the HP-UX kernel to include a statically-configured loadable module.

Use the config command to configure a DLKM module into the system as either dynamically loadable or statically-configured. Use this procedure to statically link the DLKM module to a new kernel.

To configure the HP-UX kernel to include a statically-configured loadable module, do the following:

Step 1. Execute this config command:

/usr/sbin/config -u /stand/system

The config builds a new kernel. The -u option forces config to call the kmupdate command, which causes the system to perform the following actions when you shutdown and restart the system:

a. Save the existing kernel file and its kernel function set directory as /stand/vmunix.prev and /stand/dlkm.vmunix.prev, respectively.

b. Move the newly generated kernel file and its kernel function set directory to their default locations, /stand/vmunix and /stand/dlkm, respectively.

After the system reboots, your DLKM module will be available as statically-configured in the new running kernel.

Whole HP-UX kernel configuration to include a statically configured loadable module.

When configuring a DLKM module into the system as statically configured, config builds an entire kernel, that is, configures the static kernel and all kernel modules. For a whole kernel configuration, config proceeds as follows:

1. Reads the traditional and modular master configuration files for the kernel modules.
2. Reads the HP-UX system file.
3. Reads the system description files for the kernel modules.
4. Checks the interface functions or symbols used by the kernel modules.
5. Generates several C output files (including two makefiles named config.mk and config.mod) describing the system configuration.
6. Executes the config.mod makefile to configure any dynamically loadable modules associated with the new kernel.
7. Executes the config.mk makefile to configure a new kernel, to link the kernel with the appropriate kernel libraries associated with traditional modules, and with the statically linked modules, and to generate the kernel symbol table.
8. Places the newly generated static kernel file (vmunix_test) in the /stand/build directory.
9. Places the generated symbol table file (symtab) and generated dynamically loadable module image files (module_names) under the kernel function set directory associated with the newly generated kernel.
10. Generates a kernel-specific KRS file that includes the module registration information associated with the newly generated kernel.

NOTE A whole kernel configuration generates several files that will make up the kernel component set for the new kernel. After a whole kernel configuration, a kernel update and a system shutdown and restart are required to be able to make the newly configured kernel the active kernel. The kernel update step, initiated with the kmupdate command or config -u, is now a required step to be able to successfully boot a newly configured kernel.
To configure the HP-UX kernel to include a statically configured loadable module and update the system with the new kernel, execute this `config` command:

```
/usr/sbin/config -u /stand/system
```

The `config` builds a new kernel. The `-u` option forces `config` to call the `kmupdate` command to schedule the update of the kernel image, a required step in the process of successfully booting a newly configured kernel.

After the system reboots, your DLKM module will be available as statically configured in the new running kernel.

### Updating the Kernel

For a DLKM module configured as statically linked, use the `kmupdate` command to schedule the update of the system with the newly configured kernel.

Each new kernel configured with the `config(1M)` command is composed of several distributed files, which are all required for a kernel to boot and run properly. The main kernel file is associated with its component files by way of a **Kernel ID String** (KIS) in the main kernel image. For any kernel, all the files required by the boot loader need to be accessible from the `/stand/boot.KIS` directory, where KIS uniquely identifies the kernel being booted.

The `kmupdate(1M)` command does the following:

- Schedules the move of the main kernel image and its associated kernel-component-set files to the default locations.
- Creates and populates the kernel-specific directory (`/stand/boot.KIS`) from which the boot loader will be able to obtain all the kernel components needed at boot time.

Call `kmupdate` after first calling `config`. If you include the `-u` option in the `config` invocation, there is no need to invoke `kmupdate`. The `config -u` command automatically invokes `kmupdate`.

```
/usr/sbin/kmupdate /stand/build/vmunix_test
```

### Loading the Kernel

For a DLKM module configured as statically linked, use the `-r` option of the `shutdown(1M)` command to load the newly generated kernel by shutting down and rebooting the system. Executing the `shutdown -r` command terminates all currently running processes and reboots the system, which causes the new kernel configuration to take effect. At that time, all statically linked, dynamically configured modules associated with the new kernel are accessible to the system.

To load the new kernel, move to the `/directory`, and execute the following `shutdown` command:

```
cd /
/usr/sbin/shutdown -r
```
Developing Dynamically Loadable Kernel Modules

This section explains the process of writing modules in the DLKM format and provides background information specific to device driver development. It focuses on the writing and installation of loadable device drivers because they constitute the majority of supported DLKM modules for HP-UX 11.0 and later releases.

This section is intended primarily for programmers who want to write DLKM modules and/or convert existing static (non-loadable) modules to the DLKM format. It is assumed that the reader has a good understanding of the HP-UX operating system, the C programming language, and the writing and testing of static device drivers.

Since kernel modules written to the DLKM specification can either be configured as dynamically loadable or statically linked, device driver developers must write their DLKM modules to accommodate either configuration.

Writing modules in the DLKM format requires writing additional module initialization code called wrapper code, which enables the DLKM infrastructure to logically connect and disconnect a loadable module to and from the running kernel. Existing traditional device drivers that are going to be configured into the system as loadable modules must be converted to the DLKM format, and must be re-packaged according to the module packaging architecture introduced in HP-UX 11.0.

This section covers the following topics:

- The “Module Component Files” section identifies and describes the file set that makes up a DLKM, which consists of the module's object file and other configuration files needed to install the module into the system.
- The “Initializing and Terminating DLKM Modules” section describes the steps that each type of DLKM module must take to initialize and terminate itself.
- The “DLKM Module Development Process” section describes the interactive processes and procedures that are of specific interest to developers of new DLKM modules, such as writing the module, and installing, testing and debugging it. Other procedures useful in the development process, such as configuring, registering, and loading a DLKM, are covered in the section “Managing Dynamically Loadable Kernel Modules” section of this chapter.
- The “Sample DLKM WSIO Class Driver” section presents a complete skeleton of a DLKM Workstation Input/Output (WSIO) class driver in HP-UX.

Module Component Files

A DLKM module must be modularly packaged (see “Managing Dynamically Loadable Kernel Modules”). A DLKM module is installed on disk as a set of files, which includes the module's object file and other configuration files needed to install the module into the system. The files are located in a single directory having an arbitrary name. The kminstall(1) command must then be used to copy the module's installation component files into the locations at which the kernel configuration tools expect to find them.

The component files for a DLKM module are:

- mod.o — the object file for the module
- master — the master configuration file for the module
- system — the system configuration file for the module
- space.h (optional) — a configuration file that allocations and initializes some module variables (tunable parameters)
Writing a DLKM Driver
Developing Dynamically Loadable Kernel Modules

- Modstub.o (optional) — an object file used by the DLKM stubs mechanism

**mod.o File Definition**

The C program source code for a DLKM module, which is completed and possible linked to become mod.o, is similar to the source code for a static module except that it contains the following additional information:

- The source code for a DLKM module includes two additional header files, `<mod_conf.h>` and `<sys/moddefs.h>`.
- The source code for a DLKM module contains two additional sections of code, one for the module’s wrapper and one for the module’s `_load()` and `_unload()` functions (routines).

**Module Wrapper**

The source code for a DLKM module must contain a wrapper, defined by the `modwrapper` structure, specifying the `_load()` and `_unload()` function entry points and other data structures used by the module. The name of the `modwrapper` structure for a DLKM module must be `module_name_wrapper`.

The wrapper data structures are partially initialized by the DLKM infrastructure using values taken from the module’s configuration files. These structures provide information needed to populate a loadable driver’s device switch table entries for the major device number it supports.

The code definition of the `modwrapper` structure is defined in header file `moddefs.h` and shown here. The data structures and `#define` statement preceding the `modwrapper` structure are also defined in `moddefs.h`. The `modwrapper` structure contains a pointer to the `modlink` structure, and the `modlink` structure contains pointers to the `mod_operations` and `mod_type` data structures. The `#define` statement indicates the revision (version) number of the `modwrapper` structure, which is 1.0 for HP-UX 11.0.

**NOTE** The `modwrapper` revision numbers will be used to maintain backward/forward compatibility of DLKM modules within the DLKM infrastructure.

An extract from `moddefs.h` file, showing `modwrapper` structure:

```c
#define MODREV 10

struct modlink {
    struct mod_operations *ml_ops;
    void *ml_type_data;
};

/*
 * Module type specific linkage structure.
 */
struct mod_type_data {
    char *mtd_info;
    void *mtd_odata;
}

struct modwrapper {
    int mw_rev;
    int (*mw_load)();
    int (*mw_unload)();
    void (*mw_halt)();
    void (*mw_conf_data);
    struct modlink *mw_modlink
};
```
The elements of the modwrapper structure are:

- **mw_rev** — Module revision number; use `MODREV` in the module wrapper.
- **mw_load** — Pointer to module’s `_load()` function.
- **ms_unload** — Pointer to module’s `_unload()` function.
- **mw_halt** — Currently unused; use `(void (*)()) NULL` in the module wrapper.
- **mw_conf_data** — Pointer to configuration data created by configuration tool `config` (1M).
- **mw_modlink** — Pointer to a modlink array, which is a NULL terminated array of modlink structures that specify the type-specific operations required by the module.

The elements of the modlink structure are:

- **ml_ops** — Pointer to a mod_operations structure that depends on the type of DLKM module being defined:

<table>
<thead>
<tr>
<th>Module Operations</th>
<th>DLKM Module Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>gio_mod_ops</td>
<td>WSIO class driver</td>
</tr>
<tr>
<td>gio_mod_ops</td>
<td>WSIO interface driver</td>
</tr>
<tr>
<td>str_drv_ops</td>
<td>STREAMS module</td>
</tr>
<tr>
<td>str_mod_ops</td>
<td>STREAMS module</td>
</tr>
<tr>
<td>mod_misc_ops</td>
<td>Miscellaneous module</td>
</tr>
</tbody>
</table>

- **ml_type_data** — Pointer to a mod_type_data structure.

The elements of the mod_type_data structure are:

- **mtd_info** — Information string returned to the `modstat()` system call.
- **mtd_pdata** — Pointer to type-specific data that depend on the type of loadable module being defined; for both STREAMS drivers and STREAMS modules, use a pointer to the `streams_info_t` structure; for all other types, use `(void *) NULL`.

A sample wrapper for a WSIO driver is as follows:

```c
/*
 * Wrapper Table
 */
extern struct mod_operations gio_mod_ops;
static drv_info_t module_name_drv_info;
extern struct mod_conf_data module_name_conf_data;

/* module type specific data */
static struct mod_type_data module_name_drive_link = {
    "module_name — Loadable/Unloadable Test Module",
    (void *) NULL
};

static struct modlink module_name_mod_link[] = {
    { &gio_mod_ops, (void *) &module_name_drv_link },
    /* WSIO */
    { NULL, (void *) NULL }
};
```
When a DLKM module is configured as statically linked, its wrapper is not used by the system.

### Load Function

**int module_name_load (void *arg);**

A module’s _load() function is called by the DLKM infrastructure whenever the module is about to be removed from active memory. The function may be given any name (typically `module_name_unload`); a pointer to the _unload() function is obtained from the module’s wrapper.

The _load() function must perform any initialization tasks required by the module before the module is logically connected to the kernel. Typical initialization tasks include acquiring private memory for the module and initializing devices and data structures.

If the module is unable to initialize itself, the _load() function must free any memory that it allocated and undo any other action that it took prior to the failure including canceling all outstanding calls to timeout. The _load() function should return 0 on success and an _errno value on failure.

The argument passed to the function is type-specific. The use of this argument is described in the “Initializing and Terminating DLKM Modules” section.

### Unload Function

**int module_name_unload (void *arg);**

The _unload() function is called by the DLKM infrastructure whenever the module is about to be removed from active memory. The function may be given any name (typically `module_name_unload`); a pointer to the _unload() function is obtained from the module’s wrapper.

The _unload() function must clean up any resources that were allocated to the module, and it must remove all references to the module. Typical cleanup tasks include releasing private memory acquired by the module, removing device interrupts, disabling interrupts from the device and canceling any outstanding timeout requests made by the module.

The module’s _unload() function should return 0 on success and an errno value on failure. In the event of failure, the function must leave the module in a sane state, since the module will remain loaded after the return.

The system will never attempt to unload a module that it thinks is busy. However, the system cannot determine under all cases when the module is in use. Currently, a module is considered to be busy when another module that depends on it is also loaded. In addition, WSIO class driver and STREAMS drivers track the open() and close() calls; these types of modules are busy whenever there is at lease one open on the device using the driver. Under most other circumstances, the module must determine for itself whether it is appropriate for it to be unloaded. When a module is still in use, its _unload() function can return a nonzero value to cancel the unload.

The argument passed to the _unload() function by the DLKM infrastructure is the same type-specific value that was passed to the module’s _load() function. The use of this argument is described in the “Initializing and Terminating DLKM Modules” section.
Master File Definition

Each DLKM module has its own master file. The format of the master file includes the following section keywords:

- **$VERSION** — Indicates the version number of the file format. Version is defined as an integer. A single line containing the version is entered. For PA systems, the only supported version is 1.

- **$LOADABLE** — Indicates that the module is dynamically loadable. If this section keyword does not exist, the module can only be statically configured into the kernel. Optionally, this section may contain the keyword:
  
  stub
  
  Which by its presence, indicates that the module has stub.

- **$INTERFACE** — Identifies the interface names and versions on which the module is built. For HP-UX 11.0 and later releases, enter a single line containing the word base.

- **$TYPE** — Indicates the module type and the type specific information. Valid types are wsio_class, wsio_intfc, streams_mod, streams_drv and misc.

- **Other sections (if required)** — $DRIVER_DEPENDENCY, $TUNABLE and $DRIVER_INSTALL:

  In the $DRIVER_DEPENDENCY section, enter the names of all other modules that depend upon this module.

  In the $TUNABLE section, enter the names and default values of the tunable parameters (variables) for the module. You may also enter minimum values for tunable parameters.

  In the $DRIVER_INSTALL section, enter the module's name and associated block and/or character major device number(s). To have the system dynamically assign a major number to your module, specify -1 in both the block major and char major fields of $DRIVER_INSTALL.

  For complete details on the use of these keywords, see the “Modular Master File” section of the master (4) manpage.

System File Definition

Every DLKM module requires a system file. The system file includes the following mandatory section keyword and three option section keywords:

- **$VERSION** — (mandatory) Indicates the version number for the file format. Version is defined as an integer and starts from one. Version 1 is the only supported file format for PA systems.

  **NOTE** The version number for the master file and system file must be the same.

- **$CONFIGURE** — (optional) Indicates if the module need to be configured into the system. If $CONFIGURE is Y or y, the module will be configured on the next build; if $CONFIGURE is N or n, the module will not be configured on the next build. The kmsystem (1M) provides the interface to modify the flag. The absence of this section implies a value of n.

  **NOTE** If $CONFIGURE is N or n, $LOADABLE is ignored.

- **$LOADABLE** — (optional) Indicates how the module will be configured. If $LOADABLE is Y or y, the module will be configured as a dynamically configured loadable module; if $LOADABLE is N or n, the module will be statically linked into the kernel, requiring a reboot. The kmsystem provides the interface to modify the flag. The absence of this section implies a value of n.
NOTE If the master file for the module does not have a $LOADABLE section, the system file should not have one either.

- $TUNABLE (empty) — Place holder for any tunable parameter specified in the associated master file for which you want to specify a value other than the default value. Leave this section blank.

The kmtune (1M) is the interface to modify tunable parameters in the module’s system description file and the HP-UX system file (/stand/system by default).

For complete details on the use of these keywords, see the “Modular System File” section of the system (4) manpage.

The space.h File Definition

An optional component, the space.h file contains storage allocations and initializations of data structures associated with a DLKM module when the size or initial value of the data structures depend on configurable values such as tunable parameters. In order to communicate these values to the rest of the DLKM module, the values are stored in global variables and accessed by the module via extern declarations in the modules’s mod.o file.

NOTE All tunable parameters specified in the master file are defined as global variables in the space.h file. See the “Sample DLKM WSIO Class Driver” section for clarification.

Modstub.o File Definition

An optional component, the Modstub.o file is statically configured into the kernel as a “place holder” for functions implemented in a loadable module that will be loaded at a later time. Its purpose is to enable the kernel to resolve references to the absent module’s functions.

Modstub.o contains “stubs” for entry points defined in the associated loadable module that can be referenced by other statically linked kernel modules currently configured in the system. Access to a stub causes the kernel to auto load the associated loadable module.

Configuring a module that uses stubs requires a full kernel build so that the stubs can be statically linked to the kernel.

Initializing and Terminating DLKM Modules

In a traditional system with statically linked modules, the modules are initialized during system boot. Because dynamically loadable modules can be loaded either during boot or after the system is booted, they must be initialized differently than modules that are statically linked into the kernel during system boot. On PA systems, dynamically loadable modules must be loaded at run-time.

The rest of this section describes the steps that each type of dynamically loadable module must take to initialize itself and to terminate itself. In addition to WSIO class drivers, WSIO interface drivers, STREAMS drivers, STREAMS modules and miscellaneous modules, this section includes the initialization and termination tasks for WSIO monolithic drivers. A monolithic driver is both a class driver (one that has a device special file) and an interface driver (one that touches real hardware and registers).
WSIO Class Drivers

To make a WSIO class driver loadable, the driver must provide a _load() and _unload() function and make minor changes to the way that the driver initializes itself. When a driver is statically linked, it typically links its _init() function into the dev_init chain, and the _init() functions calls the next driver’s _init() function in the chain.

This mechanism is inappropriate for loadable drivers since the dev_init chain is only used during system boot, and there is no mechanism to remove the entry from the chain when the module is unloaded.

Therefore, the module’s _load() function should perform both the installation tasks (those normally done during install for static modules) and the driver initialization tasks, and the dev_init chain should be ignored.

Additional, the _load() function is passed a pointer to an updated version of the drv_info_t structure; the driver must use this version of the drv_info structure when it registers itself with WSIO. A sample _load() function for a WSIO class driver is as follows:

```c
static wsio_drv_info_t module_name_wsio_info = { ... };

/*
 * LOAD
 */

static int
module_name_load (void *arg)
{
    if (module_name_debug)
        printf ("module_name> Loading\n");

    /* Use drv_info passed to us instead of static version */
    module_name_wsio_info.drv_info = (drv_into_t *) arg;

    /* Register with WSIO */
    if (!wsio_install_driver(&module_name_wsio_info))
    {
        printf(module_name> wsio_install_driver failed!!\n");
        return (ENXIO);
    }

    /* Perform driver-specific initialization, but do not call
     * next function in the dev_init list.
     */
    (void) module_name_init();

    return (0);
}
```

**NOTE**

The wsio_install_driver() function call returns 1 on success. However (and inconsistently), the wsio_uninstall_driver() function call returns 0 on success.

Initialization for a statically linked WSIO class driver is unchanged from historical practice. That is, the initial driver entry point is module_name_install. This function typically installs an _init() function in a list of functions that will be invoked later in the system boot process, and then the driver registers itself with WSIO. For example:
Writing a DLKM Driver
Developing Dynamically Loadable Kernel Modules

static void module_name_linked_init (void);
static int (*module_name_saved_init) ()
{

    /*
     * INSTALL
     * This function is called if module is statically linked.
     */
    int
    module_name_install (void)
    {
        extern int (*dev_init) (void);
        /* Link my init function into chain */
        module_name_saved_init = dev_init;
        dev_init = (int (*)()) &module_name_linked_init;

        /* Register driver with WSIO */
        return (wsio_install_driver(&module_name_wsio_info));
    }

The _init() function in the static case must call the next driver's _init() function:

    /*
     * Device initialization Link
     * Called only for tatically linked drivers to link init
     * routine into list.
     */
    static void
    module_name_linked_init (void)
    {
        /* Perform driver-specific initialization */
        (void) module_name_init();

        /* Call next init function in chain */
        (void) (*module_name_saved_init)();
    }

The _unload() function frees resources allocated by the driver and unregisters the driver from WSIO. If it is safe to unload the driver when the number of open devices for the driver goes to zero, then the module need not perform any special checks to determine if it is busy. If any step in the _unload() process fails, the driver must undo any action prior to the failure. A sample _unload() function for a WSIO class driver is as follows:

    /*
     * UNLOAD
     */
    static int
    module_name_unregister (void *drv_infop)
    {
        /* This function is only called when the administrator
         * attempts to unload the module and there are no open
         * devices using the module. If here is some reason that
         * the module should not be unloaded, check it now and
         * return non-zero.
         */
        if (module_name_no_unload) {
            printf("module_name> I’m BUSY\n");
            return (EBUSY);
        }

        /* Unregister with WSIO */
if ( wsio_uninstall_driver(&module_name_wsio_info) ) {
    /* Uninstall failed! Return to a loaded, functional state. */

    printf("module_name> wsio_uninstall_driver failed!!\n"
            "return (ENXIO); 
        } 

    /* Cancel pending timeouts, free allocated memory and resources, etc. */

    if (module_name_debug)
        printf("module_name> Unloaded\n");

    return (0);
}

NOTE The wsio_uninstall_driver() function call returns 0 on success. However (and inconsistently), the wsio_install_driver() function call returns 1 on success.

WSIO Interface Drivers

A WSIO interface driver requires an _attach() function for any interface card it intends to claim. It will also require a _probe() function if it intends to install device probes. When installing a WSIO interface driver, the driver must register itself with WSIO and set up the _attach() function and any device probes.

Because the timing of initialization differs between loadable and statically linked modules, some of the steps that are handled automatically during boot for static drivers must be explicitly performed by the _load() function.

The primary difference in making a WSIO interface driver loadable concerns the handling of the attach list. Historically, drivers have added their _attach() functions to a global list, and the _attach() function was responsible for calling the next driver's _attach() function in the list. However, this method does not allow the driver to remove its _attach() function from the list; thus, this approach cannot support unloading.

The primary difference in making a WSIO interface driver loadable concerns the handling of the attach list. Historically, drivers have added their _attach() functions to a global list, and the _attach() function was responsible for calling the next driver's _attach() function in the list. However, this method does not allow the driver to remove its _attach() function from the list; thus, this approach cannot support unloading.

New support functions have been added to WSIO to allow interface drivers to add and remove their _attach() functions from the attach list. These functions are:

int mod_wsio_attach_list_add(list_type, &attach_func);
int mod_wsio_attach_list_remove(list_type, &attach_func);

The list_type specifies the attach list to use; valid entries are MOD_WSIO_CORE, MOD_WSIO_PCI, and MOD_WSIO_EISA. Both functions return 0 on success and 1 on failure.
These functions should only be called when the module is dynamically loaded. Statically linked modules should continue to use the existing attach chain.

Device probes are normally associated with interface drivers during the initialization of the WSIO Context Dependent Input/Output (CDIO) at boot time. Since this is only done once, loadable interface drivers must explicitly connect the device probe to the driver's `drv_info` structure:

```c
void wsio_activate_probe (char *probe_name, struct drv_info *drv_infop);
```

The `probe_name` is the name of the probe as registered by `wsio_register_dev_probe()` or `wsio_register_probe_func()`. The `drv_infop` is a pointer to the `drv_info` structure for this driver. In general, use a probe provided by the system. For example, `parallel_scsi_probe`. See the HP-UX 11i v1 Driver Development Guide for complete information.

Finally, similar to class drivers, the `_load()` function for interface drivers is passed a pointer to an updated version of the `drv_info_t` structure; the interface driver must use this version of the `drv_info` structure when it registers itself with WSIO.

The sample code shown here, puts all these concepts together to demonstrate the loading and initialization steps for an interface driver. This sample can be configured as either a loadable module or a statically linked module.

WSIO interface driver loading and initialization coding:

```c
static wsio_drv_info_t module_name_wsio_info = { ... };
static int (*module_name_saved_attach) () ;

int module_name_load(void *arg)
{
    /* Use the drv_info passed to us instead of the static
     * version */
    wsio_info.drv_info = (drv_info_t *) arg;

    /* Add the attach function to the DLKM attach list. */
    mod_wsio_attach_list_add(MOD_WSIO_CORE, 
        &module_name_core_attach);

    /* Register the device probe. */
    wsio_register_dev_probe (IF_CLASS, probe_func,
     "probe_name");

    /* The following step is only required for dynamically
    * loaded modules: attach the probe function to the
    * drv_info structure */
    wsio_activate_probe("probe_name", wsio_info.drv_info);

    return 0;
}

int module_name_install(void)
{
    /* Add the attach function to the list. */
    extern init (*core_attach)();
    module_name_saved_attach=
        core_attach=
        core_attach=
        module_name_attach_linked;
```
The `module_name_unload()` function determines if the driver is still busy, and if not, it cleans up all resources that were obtained by the driver. It should free memory that it allocated and remove the `attach()` function from the attach list. It should also unregister any `probe()` function pointers that the driver has registered with other kernel services; for example, the `module_name_unload()` function should unlink the interrupt service function. If an operation fails while unloading, the `module_name_unload()` function must be able to restore the driver to a working state and return a nonzero value.

Unlike class drivers, there is no automatic method for the system to determine if an interface driver is still busy. This problem can sometimes be avoided by making class drivers that use the interface driver dependent upon the interface driver. With this dependency relationship in place, the system will not allow the interface driver to be unloaded as long as any class driver that depends upon it is still loaded. If it is not appropriate for the interface driver to rely on dependencies, then it must determine via other means if it is possible to unload the driver. The `module_name_unload()` function is always free to return a nonzero value, and the module will remain loaded. A sample `module_name_unload()` function for a WSIO interface driver is as follows:

```c
int module_name_unload(void *arg)
{
    int ret;
    struct isc_table_type *isc;
    void *token, *priv_ptr;
    return (EINVAL);

    /* Remove the attach function from the DLKM attach list. */
    if (mod_wsio_attach_list_remove(MOD_WSIO_CORE, &module_name_core_attach)) {
        return (ENXIO);
    }

    /* Unregister the device probe. */
    wsio_unregister_dev_probe(IF_CLASS, probe_func,
        "probe_name");
```
/* For each ISC belonging to the driver:
   * Conceal pending timeouts, free allocated memory.
   * unregister ISR routine, etc.
   */

If (wsio_uninstall_driver(&module_name_wsio_info)) {
   /* uninstall failed — go back to loaded state
    * undo what has been done in _unload routine */
   return ENXIO;
} return(0);

WSIO Monolithic Drivers

The _load() function for a monolithic driver must effectively be the union of the _load() functions for the class and interface drivers described previously. Similarly, the _unload() function must effectively be the union of the _unload() functions for the class and interface drivers.

STEAMS Drivers

Initialization of STREAMS drivers is very similar for both the loadable and statically linked module cases. The only difference is that loadable drivers must use the drv_info_t structure that is passed as an argument to the _load() function. The major numbers from this structure must also be used in the streams_info_t structure passed to str_install. Sample _load() and _install() functions for a STREAMS driver are as follows:

```c
static drv_info_t str_drv_info = { ... };
static drv_info_t *drv_into_p = &str_drv_info;

static streams_info_t str_info = { ... };
static drv_ops_t module_name_str_drv_ops = { ... };

int module_name_load(void *arg)
{
    int    retv;

    /* Use the drv_info passed to us instead of the static
    * version */
    drv_info_p = (drv_info_t *) arg;
    str_info.inst_major = drv_info_p->c_major;

    if (module_name_install())
        return ENXIO;
    return 0;
}

int module_name_install(void)
{
    int    retv;

    /* Install in cdevsw */
    if ((retv=install_driver(drv_info_p,
                      &module_name_str_drv_ops)) != 0)
        return (retv); /* install failed */
```
STREAMS drivers, like WSIO class drivers, automatically track open() and close() system calls for the STREAMS device. The system will prevent a STREAMS driver from unloading whenever the device has one or more open file handles. Of course, the driver can still disallow an unload if this check is insufficient for its needs. A typical _unload() function for a STREAMS driver is as follows:

```c
module_name_unload(void *arg)
{
    int retval;

    /* Uninstall from Streams */
    if ((retval = str_uninstall(&str_info)) != 0)
        return retval;

    /* Free module specific resources, etc. */
    ...
    return 0;
}
```

**STREAMS Modules**

Loadable STREAMS modules have no special requirements during initialization. The argument passed to the _load() function should be ignored. Sample _load() and _install() functions are as follows:

```c
static streams_info_t str_info = { ... };

int module_name_load(void *arg)
{
    int retval;

    if (module_name_install())
        return ENXIO;
    return 0;
}

int module_name_install(void)
{
    /* Install in Streams */
    if ((retval = str_install(&str_info)) != 0)
        return retval;
    return 0;
}
```

The system automatically tracks pushes and pops of a STREAMS module on active streams; a module cannot be unloaded while it is pushed onto one or more streams. A typical _unload() function for a STREAMS module is as follows:

```c
int module_name_unload(void *arg)
{
    int retval;
```
/* Uninstall from Streams */
if ((retval = str_uninstall(&str_info)) != 0)

    /* Free module specific resources, etc. */
    ...
    return 0;
}

Miscellaneous Modules

Miscellaneous modules can implement any feature within the kernel. As such, a miscellaneous module’s
_load() function must address all of the module’s specific needs. Similarly, the module’s _unload() function
must determine for itself if it is safe to unload. The system will not allow a module to be unloaded if other
loaded modules are dependent upon the module. Other than this check, the system performs no other checks
when the administrator attempts to remove a miscellaneous module from the kernel.

The argument to the _load() function is not meaningful and should be ignored.

DLKM Module Development Process

Developing a DLKM module such as a device driver requires more up front planning than most application
programming projects. At the very least, testing and debugging are more involved, and more knowledge about
hardware is required.

This section describes the procedures for developing a DLKM module. Since you can include a DLKM module
in the system in one of two ways, either as a Dynamically Configured Loadable Module (DCLM) or as a
Statically Configured Loadable Module (SCLM), this section considers both methods of inclusion.
Complete command line invocations are included in the detailed descriptions where appropriate.

Developing a DLKM module is an iterative process. The developer writes, tests and debugs module in a
piecemeal fashion, building up to the implementation of the complete module.

During the first phases of development, possible errors in the module code may panic or damage the system,
even parts of the system that may seem unrelated to your module. Testing should be done when no other
users are on the system and all production data files ar backed up. Ideally, testing would be performed on a
restricted use system set up specifically for the purpose of developing kernel modules.

Notes

- Before starting the development process, it is always a good idea to make a backup of /stand/vmunix for
emergency recovery purposes.

- Although the kmupdate(1M) command always saves the previous /stand/vmunix to the kernel
  name/stand/vmunix.prev, be aware that a subsequent kmupdate(1M) will again overwrite the
  vmunix.prev kernel.

- The recommended way to permanently save /stand/vmunix is to move it to a save name, such as
  /stand/vmunix.bkup.

- **Do Not Copy!** Copying the /stand/vmunix file may result in the eventual loss of the kernel component
  set necessary to boot the copy.

- Furthermore, if /stand/vmunix had been the running kernel, we recommend an immediate reboot from
  the renamed location:

  # mv /stand/vmunix /stand/vmunix.bkup
  # shutdown -r

456 Chapter 16
Once /stand/vmunix has been moved, there is no longer a default kernel from which to boot until a new kernel is configured and updated via the kmpupdate(1M) command. To boot from a non-default kernel, refer to “Boot from an Alternate Kernel” in the Managing Systems and Workgroups manual.

Creating the Module's Component Files

Create the mod.o, master, system, space.h (optional), and Modstub.o (optional) files for the DLKM module in a single directory. It is suggested the subdirectory of /usr/conf be used such as /usr/conf/module_name.

When choosing a name for your module, choose a unique name to avoid conflict with kernel functions and global variables. Consider using your company's name and something that indicates the module's purpose.

To compile the module source code, use the ANSI C compiler /opt/ansic/bin/cc and the compiler options shown. The developer can also use the compiler options found in the makefile /stand/build/config.mk or /stand/build/config.mod.

To compile the module source code, follow these steps:

1. Change directories to the directory containing the module's component files.
2. For a 64-bit PA target machine, execute the following cc(1) command:

```
/opt/ansic/bin/cc -I. -I/usr/conf -I/usr/conf/sys \ 
-VM, -H300000 -chx0 +R500 +ESesfc +ESesf +XixD U \ 
+ES1.Xindirect_calls +ESlit +02 +Oentriesched +Ofastaccess \ 
+DA2.0W +DS2.0 -Ae -DLMSYSCALL -DPGPROF -DACLS -DAUDIT \ 
-DIDDS -D__ROSE__ -DHIPONCPLUS -D__ROSEVILLE__ \ 
-DSPP_OBP_BOOT -DSPP_RUNWAY_ERR_ENABLED -DPARISC \ 
-DRDB -DNEW_RDB -DKGBON -DIVT_INTERCEPT -DCOMB_FLIPPER \ 
-DNEW_MFCTLW -DSTCP -DUNSUPPORTED \ 
-D_HPUX_SOURCE -D_XPG4_EXTENDED -D_KERNEL -D__STDC_EX T__ \ 
-D_CLEAN_BE -D__TEMP64 -D_hp9000s800 -D__NO_EM_HDRS \ 
-U_hp9000s700 -o mod.o -c module_name.c
```

Installing the Module's Component Files

Once the module's mod.o, master, system, and optional files have been created, call the kminstall command to copy those files to certain subdirectories of /usr/conf and /stand. The kminstall command creates the required subdirectories if they do not exist.

The kminstall expects to find the module's component files in the current working directory. If module_name already exists on the system, kminstall prints a message and fails.

To install a DLKM module's components, follow these steps:

1. Change directories to the directory containing the module's component files.
2. Execute the following kmininstall command:

```
/usr/sbin/kmininstall -a module_name
```

The kmininstall copies the module's component files to the appropriate locations. For example, kmininstall copies the master file to /usr/conf/master.d/module_name/0.1.0, and the system file to /stand/system.d/module_name/0.1.0. File /usr/conf/master.d/module_name/0.1.0 is known as the module's master configuration file. File /stand/system.d/module_name/0.1.0 is known as the module's system description file.

**NOTE** The file locations receiving copies of the module's component files may change in future HP-UX releases.
Activating the DLKM

After a module's component files have been installed using the kminstall command, the module is ready to be configured and registered with the running kernel. Follow the procedures "DLKM Procedures for Dynamically Configured Loadable Modules" for instructions on how to prepare, configure, register, load and manage your DLKM as dynamically loadable, on the target system.

To configure and test the DLKM as statically linked refer to the "DLKM Procedures for Statically Configured Loadable Modules" section.

Creating Device Special Files for the Module

Before devices supported by a driver type DLKM module can be accessed by the system. The developer does not have to create these files every time they build and boot a new kernel-only upon adding the new module. There must be a special file for each device on the system.

When you set up the special files for a newly added device driver, the developer must specify the device type (character or block) and the major and minor device numbers. The kernel recognizes any single device by the major-minor number combination encoded in the device special files.

For a dynamically loadable driver, the system assigns a major number to the driver when it first registers with the system. For a statically linked driver, the system assigns a major number to the driver during system boot. The system remembers assigned major numbers from system boot to system boot.

To have the system dynamically assign a major number to the driver follow these steps:

1. Specify -1 in both the block major and char major fields in the $DRIVER_INSTALL section of the driver's master file.
2. Specify -1 in both the b_major and c_major fields of the drv_info_t structure of the driver source code.

Also, set the following bit values in the flags field in the drv_info_t structure; if developer has a block driver, set the DRV_BLOCK value; if the developer has a character driver set the DRV_CHAR value; if your driver is both a block and a character driver, set both values.

NOTE
This step applies only to statically configured modules. The drv_info_t structure in driver source is ignored for dynamic configuration.

To create device special files for the driver, follow these steps:

1. Use the lsdev(1M) command to identify the major number assigned to the device driver. lsdev lists all device drivers configured into the kernel and their block and character major numbers.

To extract the block major number from the display for a driver named mydriver, execute the following command:

```
/usr/sbin/lsdev -h -d mydriver | awk '{print $2}'
```

To extract the character major number from the display for a driver named mydriver, execute the following command:

```
/usr/sbin/lsdev -h -d mydriver | awk '{print $1}'
```

As an alternative and assuming a dynamically loadable driver, you can execute the following two commands to identify the major number assigned to a driver named mydriver:

```
/usr/sbin/kmadmin -L mydriver
/usr/sbin/kmadmin -Q mydriver
```

2. Construct a minor number for each device special file that you create for the driver.
3. Use the `mknod` command to create the device special files for the driver.

   In the following example, `mydriver` was dynamically assigned the block and character major numbers 65 and 234, respectively. Its minor number, 0x026000, is constructed like that of `instr0` (bits 8 through 15 encode 2 as the instance of the interface card, and bits 16 through 19 encode 6 of the device's address).

   ```
   /usr/sbin/mknod /dev/mydriver b 65 0x026000
   /usr/sbin/mknod /dev/mydriver c 234 0x026000
   ```

   **NOTE** You may be able to use the `ioscan` and `insf` commands to install special files in the `/dev` directory. If no options are specified, `insf` creates special files for all new devices in the system. New devices are those devices for which no special files have been previously created. See the `ioscan(1M)` and `insf(1M)` manual pages for more details.

   **Testing the Module**

   Testing a DLKM module consists of testing all of its functions under a variety of operating conditions and for both configurations; dynamically loadable and statically linked. Debugging a module is largely a process of analyzing the code to determine what could have caused a given problem.

   The most common debugging technique is monitoring the kernel code using the `printf` function to print statements included in the module source code so that the developer knows what the module is doing during run-time operation. There are also debuggers such as Q4 to examine the kernel code. See the Q4 description file in the `/usr/contrib/doc` directory for more detail.

   The developer can use a combination of C preprocessor macros, conditional compilation, and control variables to turn `printf()` messages on or off. The sample template character driver, “dlclass.c” section, includes a control variable named `dlclass_debug` to determine whether or not to generate the messages. All of the messages can be disabled at once by using `kmtune -s` to change the value of `dlclass_debug` to 0 before configuring `dlclass`.

   **Correcting Errors in the Module's Component Files**

   Correcting errors in the DLKM module is largely a process of analyzing the code to determine what could have caused a given problem. Some common errors are as follows:

   - Coding problems
   - Installation problems
   - Data structure problems
   - Timing errors
   - Corrupted interrupt stack
   - Accessing critical data
   - Overuse of local driver storage
   - Incorrect DMA address mapping

   **Updating the Module's Components**

   Use the `-u` option of the `kminstall` command to update a DLKM module's components in the subdirectories of `/usr/conf` and `/stand`. Use this command when you modify the module source code, you must first recompile the source code in accordance to the “Creating the Module's Component Files” section.
The `kminstall` expects to find the module's component files in the current directory. If `module_name` already exists on the system, `kminstall` updates the module. If `module_name` does not exist on the system, `kminstall` prints a warning and adds the module's components to the system.

When updating an existing module, `kminstall` takes the values of the turnable parameters and the `$CONFIGURATION` and `$LOADABLE` flags from the module's current system description file and save them to the module's new system description file.

To update a DLKM module's components, follow these steps:

1. Change directories to the directory containing the module's component file.
2. Execute the following `kminstall` command:

   ```
   /usr/sbin/kminstall -u module_name
   ```

   The `kminstall` copies the module's component files to the appropriate locations, thereby overwriting the same named files at those locations.

Removing the Module's Components from the System

Use the `-d` option of the `kminstall` command to remove a DLKM module's components from the system. If the specified module is listed in the `/etc/loadmods` file, `kminstall` prints a warning message and removes the module entry from `/etc/loadmods`. If the specified module is currently loaded, `kminstall` tries to unload the module. If the unload fails, it prints a message and exits with an error; otherwise, `kminstall` tries to unregister the module. If the unregistration fails, `kminstall` prints a message and exits with an error.

To remove a DLKM module's components, execute the following `kminstall` command:

```
/usr/sbin/kminstall -d module_name
```

The `kminstall` deletes the module's components from the `/usr/conf` and `/stand` subdirectories.

**NOTE**  
`kminstall` does not delete the original component files for the module.

If the removed DLKM module is statically linked into the running kernel, you will have to execute `config -u` to reconfigure the kernel once the module has been removed. Then you will have to shutdown and restart the system for the new configuration to take effect.

Sample DLKM WSIO Class Driver

This section presents a complete skeleton of a DLKM WSIO class driver in HP-UX. The files shown are:

- `dlclass.c` — The template character driver.
- `master` — The master file for the sample character driver.
- `system` — The system file for the sample character driver; `$CONFIGURE` and `$LOADABLE` flags are both set to `Y`.
- `space.h` — A configuration file that sets two flags and allocates space for the sample character driver.
- `Makefile` — A makefile for the template character driver; initiates the compiling of `dlclass.c` and places a copy of `mod.o` in the current working directory.
- `mod.mk` — A file called by `Makefile` that contains `cc` command and appropriate options to compile `dlclass.c` for a 64-bit PA target machine.

This driver skeleton is a complete working character driver that does not use any hardware. It can be added to any HP-UX 11.0 kernel and executed as an example.
The character driver supports a pseudo device. Characters written to the device are passed to the kernel message buffer, so you have to execute the `dmesg(1M)` command to actually see the output. A read sequence results in the printing of the following hardcoded string compiled into the driver: “Reading from loadable WSIO driver ‘dlclass’.”

The character driver supports three tunable parameters:

- **dlclass_no_unload** (integer) — Prevents unloading when nonzero; initially set to 0 in the `$TUNABLE` section of the master file.
- **dlclass_debug** (integer) — Writes debugging output into message buffer when nonzero; initially tuned on (set to 1) in the `$TUNABLE` section of the master file.
- **dlclass_bufsz** (integer) — Size of internal write buffer; initially set to 40 characters in the `$TUNABLE` section of the master file.

To install, configure and load the character driver, log in as user `root` and perform the following steps:

1. Execute the following command to create a working directory for `dlclass`:
   ```bash
   mkdir /usr/conf/dlclass
   ```
2. Change directories to the `/usr/conf/dlclass` directory.
3. Create all component files for `dlclass` in addition to `Makefile` and `mod.mk` in the `/usr/conf/dlclass` directory.
4. Execute the following command to generate the `dlclass` driver:
   ```bash
   make
   ```

   The `make(1)` command examines the local `Makefile` and generates the `dlclass` driver according to the rules in the `Makefile`. It compiles `dlclass.c` into `mod.o`. It then installs and configures the driver. The `make` command also loads the driver and extracts the dynamically assigned major number used to create device special file `/dev/dlclass`.

The developer can read a message from `dlclass` by executing the following command: `cat /dev/dlclass`. The following message prints on your computer screen: “Reading from loadable WSIO driver ‘dlclass’.”

The developer can write a message to `dlclass` by executing a command such as `echo Hello Hello > /dev/dlclass`. To see the output, execute the following command: `dmesg | tail`. The system responds as follows:

```
dlclass> OPEN -- WRITE BUFFER SIZE = 40
dlclass> 'Hello Hello'
dlclass> CLOSE
```

**NOTE** When `dlclass` is unloaded (`kmadmin -U dlclass`), you can auto load `dlclass` by executing a command such as `cat /dev/dlclass`.  

Chapter 16  461
Writing a DLKM Driver
Developing Dynamically Loadable Kernel Modules

Sample Component File

dlclass.cl

/*
 /* Loadable/Unloadable Test Driver - dlclass
 */
#include <sys/types.h>
#include <sys/param.h>
#include <sys/errno.h>
#include <sys/malloc.h>
#include <sys/mod_conf.h>
#include <sys/moddefs.h>
#include <sys/io.h>
#include <sys/wsio.h>

/* Entry Points */
int dlclass_install(void);
static int dlclass_load(void *drv_infop);
static int dlclass_unload(void *vrv_infop);
static int dlclass_open(dev_t dev, int flags, intptr_t dummy
int mode);
static int dlclass_close(dev_t dev, int flags, int mode);
static int dlclass_read(dev_t dev, struct uio *uio);
static int dlclass_write(dev_t dev, struct uio *uio);

/* Local funcitns */
static int dlclass_init (void);
static void dlclass_linked_init (void);

/* Tunable Parameters */
extern int dlclass_no_unload;
extern int dlclass_debug;
extern int dlclass_bufsz;

/* message for dlclass_read() */
static char dlclass_msg[] = "Reading from loadable WSIO driver
'dlclass'.\n";

/*
 * Wrapper Table
 */
extern struct mod_operations gio_mod_ops;
static drv_info_t dlclass_drv_info;
extern struct mod_conf_data dlclass_conf_data;

/* module type specific data */
static struct mod_type_data dlclass_drv_link = {
   "dlclass - Loadable/Unloadable Test Module",
   (void *)NULL
};

static struct modlink dlclass_mod_link[] = {
   { &gio-mod_ops, (void *)&dlclass_drv_link }, /* WSIO */
   { NULL, (void *)NULL }
};
struct modwrapper dlclass_wrapper = {
    MODREV,
    dlclass_load,
    dlclass_unload,
    (void (*)(void))NULL,
    (void *)&dlclass_conf_data,
    dlclass_mod_link
};

/*
 * Driver Header
 */
static drv_info_t dlclass_drv_info = {
    "dlclass",
    "pseudo",
    DRV_CHAR|DRV_PSEUDO|DRV_SAVE_CONF|DRV_MP_SAFE,
    -1,
    -1,
    NULL,
    NULL,
    NULL,
    NULL,
    DRV_PSEUDO
},

static drv_ops_t dlclass_drv_ops = {
    dlclass_open,
    dlclass_close,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    0
},

static wsio_drv_info_t dlclass_wsio_info = {
    &dlclass_drv_info,
    &dlclass_drv_ops,
    &dlclass_wsio_data,
};

static int (*dlclass_saved_init) ();

/*
 * LOAD
 */
static int dlclass_load(void *arg)
{
    if (dlclass_debug)
        printf("dlclass> Loading\n");
/* Use drv_info passed to us instead of static version */
dlclass_wsio_info.drv_info + (drv_info_t *) arg;

/* Register with WSIO */
if (!wsio_install_driver(&dlclass_wsio_info))
{
    printf("dlclass> wsio_install_driver failed!!\n")严重
    return (ENXIO);
}

/* Perform driver-specific initialization, but do not call
* next function in the dev_int list.
*/
(void) dlclass_init();
return (0);

/*
* UNLOAD
*/
static int
dlclass_unload(void *drv_infop)
{
    /* This function is only called when the administrator
    * attempts to unload the module and there are no open
    * device using the module. If there is some reason that
    * the module should not be unloaded, check it now and
    */
    return (dlclass_no_unload) {
        printf("dlclass> wsio_uninstall_driver failed!!\n")严重
        return (EBUSY);
    }
    /* Cancel pending timeouts, free allocated memory and
    * resources, etc.
    */
    if (dlclass_debug)
        printf("dlclass> Unload\n");

    return (0);
}

/*
* INSTALL
* This function is called if module is statically linked.
*/
ing
dlclass_install(void)
{
    extern int (*dev_init) (void);
    /* Link my init function into chain */
dlclass_saved_init = dev_init;
dev_init = (int (*) ()) &dlclass_linked_init;

    /*
    * Device initialization Link
    * Called only for statically linked drivers to link init
    * routing into list.
    */
    static void
dlclass_linked_init (void)
{  /* This driver has no initialization code. But if it did,
   * it would go here!
   */

   return 0;  /* return value is ignored. */
}

/*
 * OPEN
 */
static int
dclass_open(dev_t dev, int flag, intptr_t dummy, int mode)
{

   if (dclass_debug)
      printf("dclass> OPEN -- write buffer size = %\n", dclass_bufsz);

   return (0);
}

/*
 * CLOSE
 */
static int
dclass_close(dev_t dev, int flag, int mode)
{

   if (dclass_debug)
      printf("dclass> CLOSE\n");

   return (0);
}

/*
 * READ
 */
static int
dclass_read(dev_t dev, struct uio *uio)
{

   if (dclass_debug)
      printf("dclass> READ\n");

   if (uio->_offset == sizeof(dclass_msg))
      return (0);
   if (uio->uio_offsret > sizeof(dclass_msg))
      return (ENXIO);

   /* copy out my message */
   return (uiomove((caddr_t) (dclass_msg +uio->uio_offset),
                    sizeof(dclass_msg)-uio->uio_offset, 
                    UIO_READ, UIO) );
}
/* This code could have problems if uio->uio_offset
* is negative (a page boundary is crossed)
*/

/* WRITE */
static int
dlclass_write(dev_t dev, struct uio *uio)
{
    caddr_t bufp;
    char *end;
    int len, err;

    bufp = kmalloc(dlclass_bufsz, M_IHV, M_WAITOK);
    len = MIN(uio->uio_resid, dlclass_bufsz);

    /* print out user data */
    if ( !err = uiomove(bufp, len, UIO_WRITE, UIO) ) {
        bufp[len] = 0;
        printf("dlclass> ", bufp);
        if (len == dlclass_bufsz) {
            printf("...
");
        }
    }

    kfree(bufp, M_IHV);
    return (err);
}

master
*
* master file for "dlclass" module
*

$VERSION
1
$$$

$DRIVER_INSTALL
*
* Driver Block major Char major Required for
* minimal system
*
dlclass
$$$

$LOADABLE
$$$

$INTERFACE
bass
$$$

Chapter 16
Writing a DLKM Driver
Developing Dynamically Loadable Kernel Modules

Chapter 16 467

$DRIVER_DEPENDENCY

$TYPE
diclass wsio_class pseudo cpml -1 -1

$TUNABLE
diclass_no_unload DLCLASS_NO_UNLOAD 0
diclass_debug DLCLASS_DEBUG 1
diclass_bufsz DLCLASS_BUFSZ 40 8

system

* system file for "dlclass" module
*
$VERSION 1
$CONFIGURE Y
$LOADABLE Y
$TUNABLE

space.h

/*
 * Tunable parameters for "dlclass" module
 */
int dlcals_no_unload = DLCLASS_NO_UNLOAD;
int dlclass_debug = DLCLASS_DEBUG;
int dlclass_bufsz = DLCLASS_BUFSZ;

Makefile

# This makefile needs to be invoked as follows:
#
#make <options>
#
# Here, options include:
#
# all to configure module and create device file
# config to configure the module in target system
# dev to create the necessary device file
# load to load the module
# status to show status of loaded module(s)
#
# CC must be set to the ANSI-C compiler installed on the target
# system, e.g. CC=/opt/ansic/bin/cc
#

MODULE=dlclass
DEV_C=/dev/dlclass
DEV_B=

all: config dev
config:mod.o
   kminstall -a $(module) 2> /dev/null

config -M $(MODULE) -u 2> /dev/null

dev:config $(DEV_C) $(DEV_B)

# If the module requires a block device, this target can be used as a template. Replace "DEV_C" with "DEV_B", "Character" with "Block", and the "c" in the mknod command with a "b".

# $(DEV_C):
kmadmin -L $(MODULE)
kmadmin -Q $(MODULE) | awk '/Character Major/ { \
    re-system("mknod $(DEV_C) c \"$3 \" 0"), \ 
    exit re; \ 
}';

mod.0:$(MODULE).c
@if [ -n "$(arch)" ]; then 
make -f mod.mk mod.o ARCH=$(ARCH) MODULE=$(MODULE); 
elif [ "\`uname -m\" = "ia64" "]; then 
make -f mod.mk mod.o ARCH=ia64 MODULE=$(MODULE); 
else 
make -f mod.mk mod.o ARCH=pa64 MODULE=$(MODULE); 
fi 

load:config
kmadmin -L $(module) 2> /dev/null

status:
kmadmin -s

clobber:
rm -f mod.o $(DEV_C) $(DEV_B)

mod.mk

# #Makefile to build DLKM modules # #Inputs: # #MODULE must be defined as the DLKM module to build #ARCH must be pa64 # INCDIR= -I. -I/usr/conf -I/usr/conf/sys # #PA20 Specific definitions #
IDENTS_pa64+ -DLWSYSCALL -DPGPROF -DACLS -DAUDIT -DIDDS -
D__ROS__ -DHPPONCPLUS -D__ROSEVILLE__ -DSPP_OBP_BOOT -
DSPP_RUNWAY_ERR_ENABLED -DPARISC -DRDB -DNEW_RDB -DKGDB_ON -
DIVT_INTERCEPT -DCOMB_FLIPPER -DRDB -DNEW_MFCTL_W -DSTCP -DIPSEC -
D_UNSUPPORTED -D_HPUX_SOURCE -D_XPG4_EXTENDED -D_KERNEL -
D__STDC_EXT__ -D_CLEAN_BE -D__TEMP64__ -D__hp9000s800 -
D__NO_EM_HDRS -U__hp9000s700

CC_OPTS-PA64= -wP, -h300000 +Hx0 +R500 +ESsfc +ESsfc +XixdU
+ES1.Xindirect_calls +ESlit +O2 +Oentriesched +Ofastaccess
+DA2.0W +DS2.0 -Ae

all: mod.o

mod.o: ${MODULE}.O

${LD} -r -o mod.o ${MODULE}.O

${MODULE}.o: ${MODULE}.c

echo "Compiling ${MODULE}.c ..."

${CC} ${INCDIR} ${CC_OPTS_${ARCH}} ${INDENTS_${ARCH}} -o
${MODULE}.o -c ${MODULE}.c
Chapter 17

How to Make Pre 11.0 Drivers 64-Bit Safe

This chapter will show you how to modify existing 32-bit drivers to be 64-bit clean so they can be compiled to run in either type of kernel.

For information about modification of Pre-Release 10.0 drivers, see the previous HP-UX 10.20 Driver Development Guide, HP Part No. B2355-90066.

NOTE

For information about specific driver entry, points see the information for that entry point in a previous chapter; for example, `driver_ioctl()`, in Chapter 4, “Writing a Driver.”

HP-UX 11.0 has two versions of the operating system kernel: a 32-bit kernel and a 64-bit kernel. The 64-bit kernel extends the capabilities of the 32-bit kernel in several ways. Among the extensions are the ability to address larger than 4 gigabytes of physical memory, map up to 16 terrabytes of virtual address space for 64-bit application programs and allow 32-bit and 64-bit applications to coexist on the same system.

To integrate your I/O driver with the 64-bit kernel requires an understanding of the differences between the 32-bit and 64-bit data models. It also requires that you make your I/O driver source “64-bit clean” and robust enough to deliver predictable results when executing in the 64-bit kernel environment.
The 64-bit Data Model

ILP32 is a C language data model where \textit{int}, \textit{long}, and \textit{pointer} data types are 32 bits wide. This is the data model commonly used by 32-bit operating systems, including the 32-bit HP-UX.

LP64 is another C language data model where the \textit{int} data type remains 32 bits wide, but \textit{long} and \textit{pointer} data types are scaled up to 64 bits. This is the data model that has been adopted by HP for its 64-bit HP-UX implementation. Table 17-1, “Summary of LP64 and ILP32 Data Models,” summarizes the two data models.

### Table 17-1 Summary of LP64 and ILP32 Data Models

<table>
<thead>
<tr>
<th>Data type</th>
<th>LP64 bit size</th>
<th>ILP32 bit size</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>
<tr>
<td>int</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>long</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>long long</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>pointer</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>float</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>double</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>long double</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>struct</td>
<td>depends on members</td>
<td>depends on members</td>
</tr>
<tr>
<td>enum</td>
<td>32</td>
<td>32</td>
</tr>
</tbody>
</table>

LP64 Considerations

Size differences between the ILP32 and LP64 data models occur with \textit{long} and \textit{pointer} data types. As a consequence, the default integral data type \textit{int} can differ in size from \textit{long}. This subtle difference can cause results unintended by the programmer.

Consider the following example:

```c
main() {
    long L = -1;
    unsigned int i = 1;
    if (L > i)
        printf ("L greater than i\n");
    else
        printf("L not greater than i\n");
}
```

Under ANSI C integral promotion rules, if sizeof(int) equals sizeof(long) this will print:

```
L greater than i
```
However, if sizeof(int) is less than sizeof(long) this will print:

L not greater than i

Both results are ANSI conforming, correct, and consequences of the value preserving integral promotion rules of ANSI. If the same code is compiled in K&R mode, the unsigned preserving integral promotion rules of K&R will result in the program printing “L greater than i” with ILP32 and LP64.

If we modify this example and change L to int. Under ANSI and K&R integral promotion rules, this will print “L greater than i” for both ILP32 and LP64.

This example illustrates that caution needs to be exercised when a data type is changed to or from long. The HP C compiler provides the option +M to identify code where ANSI and K&R may differ.

Another consideration is the alignment of data objects. With LP64, structures that contain long or pointer data types will be aligned to a double word offset. As an example, consider the following:

```c
struct vals {
    int intval;
    long longval;
    int endval;
};
```

The data member longval has the strictest alignment requirement of the data members in the structure. In LP64 mode, the entire structure and longval are double word aligned. A 32-bit gap will exist between intval and longval. The compiler may also pad the size of the structure to the next double word size.
General Guidelines

64-bit Clean Headers

Header files contain the data declarations, structures, constants, macros, function prototypes, and external data objects that are the interfaces to modules. The same header files are expected to be compiled with source code for 32-bit and 64-bit drivers, and possibly with 32-bit and 64-bit libraries and applications.

To make this possible, header files need to be examined for declarations and usage that may not be compatible between the ILP32 and LP64 data models. This is referred to as making header files 64-bit clean.

Here are the general guidelines to clean an I/O driver header file:

- Examine declarations where `long` (or a variant of `long`) is specified. Where appropriate, fix the size of the declaration by replacing `long` or a variant of `long` by `int` (or a variant of `int`). Cases where this may be appropriate include application visible data structures that must be sized the same for both ILP32 and LP64, driver data structures that ought to be kept from growing unnecessarily large, and hardware data structures that must be fixed in size with 32-bit data types.

- Examine declarations where `int` (or a variant of `int`) is specified. Where appropriate, scale the size of the declaration by replacing `int` (or a variant of `int`) by `long` (or a variant of `long`). Cases where this may be appropriate include offsets that must displace greater than 4 GBytes in a 64-bit kernel, storage that is overloaded to store a pointer value, and storage for machine register values.

- Examine declarations where pointer data types are specified. If the pointer is a pointer to a function, specify the full ANSI function prototype with all arguments declared. This enables ANSI code to be type checked against the function prototype and expose incompatibilities at compile-time. If the pointer is in an application visible data structure, things get complicated because the application may be a 32-bit application (compiled with ILP32 data types) and your driver may be executing in the 64-bit kernel (compiled with LP64 data types). This situation is discussed in Chapter 4, “Writing a Driver.”

Useful Data Types

`int32_t` and `uint32_t`

The data types `int32_t` and `uint32_t` represent storage that must be fixed in size to 32 bits. They are declared in the header file `_inttypes.h` as:

```c
typedef int int32_t;
typedef unsigned int uint32_t;
```

Consider the following data structure from the 10.20 header file `diskio.h`:

```c
typedef struct {
    long lba; /* capacity in DEV_BSIZE blocks */
} capacity_type;
```

Contained in the structure is the data member `lba` declared as `long`. This is an application visible data structure that should be fixed in size to avoid compatibility problems between a 32-bit application and a 64-bit driver. For example:

```c
typedef struct {
    int32_t lba; /* capacity in DEV_BSIZE blocks */
} capacity_type;
```
Since lba is intended to store values greater than or equal to zero and negative values are never stored, the correct declaration for lba is an unsigned type. The following is the 64-bit clean version of capacity_type in release 11.0:

```c
typedef struct {
  uint32_t lba;
  /* capacity in DEV_BSIZE blocks */
} capacity_type;
```

### intptr_t and uintptr_t

The data types intptr_t and uintptr_t represent storage that must scale in size with a pointer data type. They are declared in the header file _inttypes.h as:

```c
typedef long intptr_t;typedef unsigned long uintptr_t;
```

Consider the following macro from the 10.20 header file cpu.h:

```c
#define ALIGN_IOBUF(P) ((char *)(((uintptr_t)(P) & \ ~(CPU_IOLINE-1)))
```

The pointer P is cast as (int), but the cast does not scale in size with the pointer. The cast is changed to (uintptr_t) in release 11.0 as shown below:

```c
#define ALIGN_IOBUF(P) ((char *)(((int)(P) \ + CPU_IOLINE-1) & ~(uintptr_t)CPU_IOLINE-1)))
```

The cast in the second line explicitly promotes the value CPU_IOLINE-1 to uintptr_t before complementing the value.

### ptr32_t

The data type ptr32_t represents storage for a 32-bit pointer and is declared in _inttypes.h as:

```c
typedef uint32_t ptr32_t;
```

For an example see the “Writing a driver_ioctl() Routine” section in Chapter 4, “Writing a Driver.”

### int64_t and uint64_t

The data types int64_t and uint64_t represent storage that must be fixed in size to 64 bits. They are declared in the header file _inttypes.h as:

```c
typedef long long int64_t;
typedef unsigned long long uint64_t
```

Be aware that application code compiled with strict ANSI (compiler option -Aa) will not recognize the data type long long. However, long long is accepted in extended ANSI (compiler option -Ae) and K&R (compiler option -Ac) modes.

### __LP64__

The __LP64__ is defined when source code is compiled with LP64 data types. The #ifdef __LP64__ may be useful where there are special LP64 considerations. For example, consider the following data structure in the 11.0 header file dma_A.h:

```c
typedef struct {
  lock_t *lock;
  dma_A_chain_type *chain_list;
#ifdef __LP64__
  uint32_t pad[12]; /* align it to 64 bytes */
#else /* !__LP64__ */
  uint32_t pad[14]; /* align it to 64 bytes */
#endif /* !__LP64__ */
} dma_A_pool_t;
```
General Guidelines

The `#ifdef __LP64__` is used to pad the data structure so that the size is 64 bytes in both ILP32 and LP64 modes.

Hardware Considerations

I/O drivers often map data structures into I/O hardware registers and memory areas that are accessed by I/O hardware. Care must be exercised to ensure that these structures are fixed in size when compiled with LP64 data types.

Consider the data structure `compl_head` in the 10.20 header file `llio.h`:

```c
struct compl_head {
    int sema;
    struct compl_entry *link;
    int filler[2];
};
```

A `compl_head` structure is updated by hardware when an I/O request completes. The `link` data member is viewed by hardware as a 32-bit value. With LP64, however, `link` will be scaled to 64 bits.

The first step is to fix the size of the structure. In release 11.0, the structure is declared as:

```c
struct compl_head {
    uint32_t sema;
    uint32_t link;
    uint32_t filler[2];
};
```

The next step is to examine code that accesses the `compl_head` and make corrections as needed.

Software Considerations

While data structures that map onto hardware and must be fixed in size are easily identified, data structures with software considerations can be much more subtle. See the following example:

Consider messages sent and received by I/O drivers in the SIO driver environment. In release 10.20, the message header is declared as:

```c
typedef struct {
    shortint msg_descriptor;
    shortint message_id;
    int transaction_num;
    port_num_type from_port
} llio_std_header_type;
```

Notice that no data member in the structure is larger than 32 bits. As such, the size of `llio_std_header_type` is 12 bytes. In iLP32 and LP64 modes, `sizeof(llio_std_header_type)` returns the value 12.

The programming assumption that causes a problem is that the message body immediately follows the header. This assumption holds with ILP32 data types; but with LP64 data types, the message body happens to contain 64-bit data members that cause the compiler to align the body to a 64-bit boundary. The header is also aligned to a 64-bit boundary and this creates a 32-bit hole between the header and body of the message.

The message is declared as:

```c
typedef struct {
    llio_std_header_type msg_header;
    union {
        creation_info_type creation_info;
        do_bind_reg_type do_bind_req;
    }
};
```
To send a message, SIO drivers add the \texttt{sizeof(llio_std_header_type)} to the size of the specific message body type being sent. In LP64 mode, the calculated size of messages are 4 bytes less than the programmer intended.

A possible solution is to explicitly pad the message header as follows:

```c
typedef struct {
    shortint  msg_descriptor;
    shortint  message_id;
    int       transaction_num;
    port_num_type from_port;
    #ifdef __LP64__
    int       filler; /* pad to 64-bit boundary */
    #endif /* __LP64__ */
} llio_std_header_type;
```

### ANSI Function Prototypes

With LP64, programmers can no longer assume that \texttt{int}, \texttt{long}, and \texttt{pointer} data types are the same size, and that these data types can be mixed across function call arguments and function return types. Many kernel functions return \texttt{pointer} or \texttt{long} data types. Without the appropriate function prototypes in scope, the compiler incorrectly truncates the return type to \texttt{int}. Programmers need to provide appropriate ANSI function prototypes (or K&R forward declarations) to their I/O drivers.

Compile time errors will result, however, if ANSI function prototypes are visible to the K&R compiler. To prevent such errors, kernel header files use the \texttt{__()} macro to declare prototypes. The macro is defined in the header file \texttt{stdsyms.h} as follows:

```c
#ifdef (_PROTOTYPES)
#define __(arg) arg
#else
#define __(arg) ()
#endif
```

To illustrate the \texttt{__()} macro, consider the following function prototype:

```c
void *fl __((long arg1, int arg2));
```

Note the double parenthesis surrounding the arguments. This notation is equivalent to the cumbersome form it replaces:

```c
#ifdef (_PROTOTYPES)
void *fl(long arg1, int arg2);
#else
void *fl();
#endif
```

Many kernel header files have incorporated ANSI function prototypes in release 11.0. Programmers should make sure that function prototypes are in scope for all kernel interfaces called by their drivers by including the appropriate header files.

To take full advantage of function prototypes, set the compiler option to \texttt{-Ae} to compile in extended ANSI mode. If your driver is compiled K&R, now is the time to convert to ANSI.

### kern_svc.h Header File

A new header file, \texttt{kern_svc.h}, has been added to \texttt{/usr/include/sys} and \texttt{/usr/conf/h}. This header file declares ANSI function prototypes of common kernel services.
Driver writers do not need to explicitly include `kern_svcs.h`. WSIO drivers include `kern_svcs.h` when `wsio.h` is included.

When porting a driver you should always check your routine declarations and parameters against `kern_svcs.h`.

**conf.h Header File**

Among the changes made to `conf.h` are the inclusion of ANSI function prototypes for the function pointers declared in the `cdevsw` and `bdevsw` structures. Driver entry points are expected to match the function prototypes declared therein.

**dma.h Header File**

Several changes have been made to the `dma.h` header file. WSIO specific macro definitions have been moved to `wsio.h`. Additionally, function prototypes have been added to the function pointer declarations in the `io_map_cntrl` structure. These prototypes enable type checking of arguments that are passed to the WSIO and SIO mapping services. Consider the `wsio_dma_alloc` macro:

```c
#define wsio_dma_alloc(Isc, Iova)  
   (*Isc->map_funcs->dma_alloc)(Isc->map_funcs->arg, Iova)
```

The macro calls the function pointed to by `dma_alloc` in the `io_map_cntrl` structure. The second argument to the macro is `Iova` which is prototyped by `dma_alloc` as `caddr_t *`. Suppose the driver calls this macro with the following code:

```c
uint32_t iova; /* only 32 bits of storage */
caddr_t pva = wsio_dma_alloc(isc, &iova);
```

This is clearly a type mismatch (and a programming error with LP64) for which the ANSI C compiler will generate a warning message. For DMA hardware that uses 32-bit `iova` values, the driver can do the following:

```c
uint32_t iova; /* only 32 bits of storage */
caddr_t tmp_iova;
caddr_t pva = wsio_dma_alloc(isc, &tmp_iova);
iova = (uint32_t)tmp_iova;
```

**uio.h Header File**

The `uio.h` header file is the declaration of the `iovec` structure. A pointer to the `iovec` structure is used not only in the `uio` structure, but is also specified as an argument type to the DMA mapping services. The `iovec` structure is declared as:

```c
struct iovec {  
caddr_t  iov_base;  
size_t  iov_len;  
};
```

With LP64 data types, the `iov_base` and `iov_len` fields will each require 64 bits of storage and must be aligned to a double word (64bit) address. I/O drivers that declare alternate data structures that map onto the `iovec` structure may require changes. For example, consider a driver that declares a data structure that represents an address/length pair as in the following:

```c
struct ac_pair {  
   uint32_t  iov_base;  
   uint32_t  iov_len
};
```

The driver must not pass a pointer to an `ac_pair` as an argument to the DMA mapping services where a pointer to an `iovec` is expected.
18 Interrupt Migration

At boot time, the HP-UX kernel allocates external card interrupts to CPOS in a round-robin method. The kernel is not aware of card traffic patterns at that time. The boot allocation scheme could allow two heavily loaded cards to be mapping interrupts to the same CPU. This might lead to system performance degradation.

Interrupt Migration provides a flexible mechanism for managing CPU interrupt assignments by the movement of external I/O device interrupts from one CPU to another.

The /usr/contrail/bin/Intel command allows the user to display the interrupt configuration of the system and to migrate the interrupts between CPUs.
Overview of Interrupt Migration Impact on Driver

The interrupt migration operation is a transparent one to applications. The associated cards need not be acquiesced during migration between processors. The applications utilizing the cards are not affected by these operations.

Drivers must be designed to spend minimal time in their ISR routines. They may need to quiesce the card interrupts during a migration operation, but quiescing the card DMA engines is not necessary.

In the HP-UX environment, drivers can use **Line Based Interrupts** (LBIs) or **Transaction Based Interrupts** (TBIs).

**LBI Drivers**

Most drivers using LBIs do not need to know which CPU is handling the interrupt. For the exceptions, WSIO provides new interrupt migration related events and informs the drivers of their use. If a driver using LBI has not registered for these interrupt event flags with WSIO, the interrupts of the associated card will be migrated to a different processor without informing the driver.

**TBI Drivers**

Drivers using TBIs spread the interrupt load across CPUs in the system and could interrupt more than one CPU. The drivers need to program their cards with the interrupt specifics (CPU address and data vector), so any TBI interrupt migration will be processed by the driver. The TBI drivers must first register with WSIO to utilize the new events.

WSIO provides new event masks for the drivers; they will be notified when a new CPU is enabled for interrupts in the system.

**Interrupt Line Sharing**

According to PCI bus specifications, a device can share an interrupt line with another device, port or function. All devices sharing the same interrupt line interrupt the same CPU. Migration of any of the interrupts sharing a line results in all interrupts on that line being migrated.
Interrupt Migration Event Masks and Registration with WSIO

WSIO defines three new event flags as part of interrupt migration. Two are for TBI based drivers, and one is for LBI based drivers.

LBI Event Flags

An optional new event flag, WSIO_EVENT_LBI_MIGR, upon registration, notifies the driver when the specified Line Based Interrupt (LBI) is migrated to a new processor.

WSIO_EVENT_LBI_INTR_MIGR

Drivers of cards using LBIs may register for this event flag. They do this with the wsio_reg_drv_capability_mask() call. A driver registers for events in its _attach() routine after claiming the card it controls by calling isc_claim(). The driver also needs to register for an event handler in its _install() routine. Registration for this event flag is needed only if the driver has cached the interrupt-CPU. With an interrupt migration operation initiated, the interrupting CPU changes to a different CPU. If the driver has registered for this event flag, it is notified of this CPU change through the event handler mechanism.

Example of a driver registering for WSIO_EVENT_LBI_INTR_MIGR event:

driver_attach(....) {
    wsio_event_mask_t newmask;
    
    isc_claim(isc);
    newmask = oldmask | WSIO_EVENT_LBI_INTR_MIGR
    ret = wsio_reg_drv_capability_mask( isc, newmask);
    ..
    ..
}

The wsio_reg_drv_capability_mask() call is described in the *HP-UX 11i v1 Driver Development Reference Guide* (DDR). If the driver has registered for the event flag, its event handler is invoked twice with the WSIO_EVENT_LBI_INTR_MIGR flag by WSIO. The first invocation notifies the associated card that the interrupt is migrating to a different CPU. The second invocation notifies the associated card that the migration is complete. The two invocations can be differentiated by the wsio_generic_event_t arg structure passed to the driver event handler. The element wsio_intr_migr_info_t inside the wsio_generic_event_t structure differentiates the two notifications.

typedef struct wsio_generic_event {
    wsio_event_t event; // WSIO_EVENT_LBI_INTR_MIGR
    wsio_event_id_t event_id; // wsio provided event_id
    struct isc_table_type *isc; // pointer to isc
generic_complete_callback_t wsio_completion_cb;
    // wsio callback
    void *arg;          // pointer to wsio_intr_migr_t
} wsio_generic_event_t;

typedef struct wsio_intr_migr {
    void *intr_object;  // Associated interrupt object

    typedef struct wsio_generic_event {
        wsio_event_t event; // WSIO_EVENT_LBI_INTR_MIGR
        wsio_event_id_t event_id; // wsio provided event_id
        struct isc_table_type *isc; // pointer to isc
generic_complete_callback_t wsio_completion_cb;
        // wsio callback
        void *arg;          // pointer to wsio_intr_migr_t
    } wsio_generic_event_t;

typedef struct wsio_intr_migr {
    void *intr_object;  // Associated interrupt object
Once the driver handles the event, it invokes the WSIO provided completion callback routine to indicate completion status of the event. The code for the completion callback routine is as follows:

```c
typedef int (*generic_complete_callback_t)(
    struct isc_table_type *,
    wsio_event_id_t, void *);

(*wsio_completion_callback) (isc, event_id, wsio_intr_migr_p);
```

The parameters are defined as:

- **isc**: Pointer to the ISC structure
- **event_id**: event id
- **wsio_intr_migr_p**: Pointer to the `wsio_intr_migr_t` that was passed to the driver event handler through the `arg` field of the `wsio_generic_event_t` structure.

The general flow of events are shown in Figure 18-1, “LBI Interrupt Migration.”
The only permissible return value from the driver for this event flag is WSIO_OK.

### NOTE
The driver event handler should *not* register for a timeout call to invoke the WSIO provided callback routine.

The specific steps are:

1. WSIO invokes the driver’s event handler routine with the event flag as WSIO_EVENT_LBI_INTR_MIGR. The `arg` field of the `wsio_generic_event_t` structure contains the interrupt object for which this event is being performed. The event `info` flag is set to WSIO_LBI_INTR_MIGR_NOTIFY.

2. The driver returns the completion status by invoking the `wsio_completion_cb` function pointer.

3. Once the driver returns, WSIO invokes low level routines (Machine dependent layer) to migrate the line based interrupt to the new CPU.

4. The migration causes a spurious interrupt on the CPU to which the interrupt of this card has been migrated. Drivers *must* be capable of handling spurious interrupts.

5. When migration is complete, WSIO invokes the driver’s event handler with the event flag as WSIO_EVENT_LBI_INTR_MIGR. The `arg` field of the `wsio_generic_event_t` structure contains the interrupt object for which this event is being performed. The event `info` flag is set to WSIO_LBI_INTR_MIGR_COMPLETE. The `new_spup_id` field in the `arg` structure contains the value of the new CPU.

6. The driver returns the completion status by invoking the `wsio_completion_cb` function pointer.
**TBI Event Flags**

Two new flags are introduced as part of interrupt migration for drivers using TBIs. The first flag, `WSIO_EVENT_OFFLINE_CPU`, is a mandatory event, and upon registration notifies the driver when the concerned Transaction Based Interrupt (TBI) is migrated to a new processor.

The second flag, `WSIO_EVENT_ONLINE_CPU`, is an optional event, and upon registration notifies the driver of any processor being enabled for interrupts.

**WSIO_EVENT_OFFLINE_CPU**

This event flag is for drivers using TBIs for their cards. All drivers using transaction based interrupts must register for the `WSIO_EVENT_OFFLINE_CPU` event flag. Drivers register for this event flag with the `wsio_reg_drv_capability_mask()` call. A driver registers for events it its _attach routine after claiming the card it controls by calling `isc_claim()`. The driver also must register for an event handler in its _install() routine.

Example of a driver registering for `WSIO_EVENT_OFFLINE_CPU` event:

```c
driver_attach(....) {
    wsio_event_mask_t newmask;
    ...
    isc_claim(isc);
    newmask = oldmask | WSIO_EVENT_OFFLINE_CPU;
    ret = wsio_reg_drv_capability_mask(isc, newmask);
    ...
}
```

Refer to the *HP-UX 11i v1 Driver Development Reference Guide* for a description of the `wsio_reg_drv_capability_mask()` call.

The interrupt objects are allocated in the `init()` routine. If a driver attempts to allocate a TBI and it has not registered for the `WSIO_EVENT_OFFLINE_CPU` event flag, WSIO fails the TBI allocation call with `WSIO_ERROR`.

The WSIO subsystem invokes the driver handler with the `WSIO_EVENT_OFFLINE_CPU` for the following scenarios:

- A CPU is being disabled or reserved for interrupts.
- The card interrupt is being bound to a different CPU.

The `wsio_generic_event_t` argument for the event handler is stated as follows:

```c
typedef struct wsio_generic_event {
    wsio_event_t event;              // WSIO_EVENT_OFFLINE_CPU
    wsio_event_id_t event_id;        // wsio provided event_id
    struct isc_table_type *isc;      // pointer to isc
    generic_complete_callback_t wsio_completion_cb;
                              // wsio callback
    void *arg;                      // pointer to wsio_intr_migr_t
} wsio_generic_event_t;
```

```c
typedef struct wsio_intr_migr {
    void *intr_object;            // Associated interrupt object
    spu_t new_spu_id;             // '-1' or new CPU id
    wsio_intr_migr_info_t info;   // NULL for this event
    wsio_intr_ret_t ret;          // return value set by the driver
} wsio_intr_migr_t;
```
Once the driver handles the event, it invokes the WSIO provided completion callback routine to indicate completion status of the event. The code for the completion callback routine is:

```c
typedef int (*generic_complete_callback_t)(struct isc_table_type *, wsio_event_id_t, void *);

(*wsio_completion_callback)(isc, event_id, wsio_intr_migr_p);
```

- `isc` Pointer to the ISC structure
- `event_id` Event identification
- `wsio_intr_migr_p` Pointer to the `wsio_intr_migr` that was passed to the driver event handler through the `arg` field of the `wsio_generic_event_t` structure.

Upon return from the driver event handler, status is set in the `ret` field inside the `wsio_intr_migr_t` structure. The permissible return values from the driver event handler are:

- **WSIO_E_DRV_DEACTIVATE_CALL** The deactivate call failed
- **WSIO_E_DRV_SET_CPU_SPEC** The `wsio_set_cpu_spec` call failed
- **WSIO_E_DRV_ACTIVATE_CALL** The activate call failed
- **WSIO_OK** Everything is OK

**NOTE** The driver event handler should not register for a timeout call to invoke the WSIO provided callback routine.

The general flow of events is presented in *Figure 18-2, “TBI Interrupt Migration.”*
The specific steps are:

1. WSIO invokes the driver’s event handler using the \texttt{WSIO\_EVENT\_OFFLINE\_CPU} flag.
2. The driver performs the following tasks to migrate the interrupt from the CPU.
   
   a. Disable the card interrupts; \emph{only} the interrupt corresponding to the interrupt object passed in the \texttt{wsio\_generic\_event\_t} is disabled. It does \emph{not} have to disable all other interrupts associated with this card.

   b. Invoke the \texttt{wsio\_intr\_deactivate()} routine for the interrupt object passed in \texttt{wsio\_generic\_event\_t} structure. This disables the interrupt at the higher levels. For example, the driver’s ISR will be removed from the CPU’s interrupt switch table. This can cause pending interrupts to be lost. To nullify these effects, a spurious interrupt is generated when the driver calls \texttt{wsio\_intr\_enable()} (described later in this sequence).

   c. Invoke the \texttt{wsio\_intr\_set\_cpu\_spec()} routine. This migrates the interrupt object sent in the \texttt{wsio\_generic\_event\_t} structure to the new CPU. See the \texttt{wsio\_intr\_set\_cpu\_spec()} call for further details.

   d. The drivers detect the new CPU to which the interrupt has been migrated with the \texttt{wsio\_intr\_get\_assigned\_cpu()} call, and determine the transaction address and the data with the \texttt{wsio\_intr\_get\_txn\_info()} call.
e. With the interrupt migrated to a new CPU, the driver invokes the `wsio_intr_enable()` routine to enable the interrupt object passed in the `wsio_generic_event_t` structure. This interface results in a spurious interrupt being fired off to the migrated CPU. Drivers must be capable of handling these spurious interrupts.

f. Program the card to use the new transaction address and transaction data obtained in step d.

g. Return the completion status (WSIO_OK) with the `wsio_completion_cb()` routine.

**WSIO_EVENT_ONLINE_CPU**

This event flag is for drivers using TBIs. If the driver has registered for this event, WSIO invokes the driver event handler with the `WSIO_EVENT_ONLINE_CPU` flag when it is notified that new CPUs are enabled for interrupt processing.

Drivers register for this event flag with the `wsio_reg_drv_capability_mask()` call. A driver registers for events in its `_attach()` routine after claiming the card it controls by calling `isc_claim()`. The driver also must register for an event handler in its `_install()` routine.

Example of a driver registering for a `WSIO_EVENT_ONLINE_CPU` event:

```c
void driver_attach(...)
{
    wsio_event_mask_t newmask;
    isc_claim(isc); newmask = oldmask | WSIO_EVENT_OFFLINE_CPU.
    ret = wsio_reg_drv_capability_mask(isc, newmask);
    ..
    ..
}
```

See the *HP-UX 11i v1 Driver Development Reference Guide* for a description of the `wsio_reg_drv_capability_mask()` call.

The `wsio_generic_event_t` argument for the event handler is written as follows:

```c
typedef struct wsio_generic_event {
    wsio_event_t event;   // WSIO_EVENT_ONLINE_CPU
    wsio_event_id_t event_id; // wsio provided event_id
    struct isc_table_type *isc; // pointer to isc
    generic_complete_callback_t wsio_completion_cb;
    // wsio callback
    void *arg;              // Number of processors enabled
} wsio_generic_event_t;
```
Interrupt Migration Flow

The HP-UX environment interrupts could be either LBIs or TBIs. From WSIO perspective, the events that can lead to interrupt migration operation are:

- Changing the interrupt state of a CPU to RESERVED or DISABLED.
  
  When changing the interrupt state of a CPU from ENABLED to RESERVED or DISABLED, all the card interrupts associated with the CPU are redistributed in a round-robin method to the other interrupt ENABLED CPUs.

- Changing the interrupt state of a CPU to ENABLED.
  
  When a CPU state is changed to interrupt ENABLED, all the drivers registered for this event are informed.

- Migration of an interrupt from one CPU to another CPU through the user level command /usr/contrib/bin/intctl.

For CPU interrupt state change details refer to subsystems such as Processor Sets, Instant Capacity on Demand (ICOD), or Real Time Extension (RTE).

The following are the steps involved in migrating an LBI and a TBI:

- **LBI processing**
  
  For every Line Based Interrupt:

<table>
<thead>
<tr>
<th><strong>WSIO Layer</strong></th>
<th><strong>Driver Layer</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>When the driver is registered for WSIO_EVENT_LBI_INTR_MIGR, the driver event handler is invoked with the event information by WSIO_LBI_INTR_MIGR_NOTIFY.</td>
<td>The driver handles the _NOTIFY event and returns WSIO_OK to WSIO.</td>
</tr>
<tr>
<td>WSIO calls the low level machine dependant routines to migrate the interrupt to the new CPU. Later, WSIO invokes the driver event handler to notify the completion event information as WSIO_LBI_INTR_MIGR_COMPLETE event.</td>
<td>The driver handles the _COMPLETE event and returns WSIO_OK to WSIO.</td>
</tr>
</tbody>
</table>

- **TBI processing**
  
  For every Transaction Based Interrupt:

<table>
<thead>
<tr>
<th><strong>WSIO Layer</strong></th>
<th><strong>Driver Layer</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>WSIO invokes the driver event handler with an event flag as WSIO_EVENT_CPU_OFFLINE.</td>
<td>Since this is a TBI, the driver performs the necessary steps for reprogramming the card with the new CPU value. Later the driver returns WSIO_OK to WSIO.</td>
</tr>
</tbody>
</table>
Assumptions and Dependencies

- The driver must handle spurious interrupts. The interrupts can happen even if the card is suspended or powered off.

- The routines `isrlink()`, `isrunlink()`, `wsio_intr_alloc()` and `wsio_intr_free()` are only invoked by the driver as part of handling events initiated from WSIO. This includes card online addition and removal, ioscan, DLKM load or unload, and interrupt migration. Drivers must not invoke these routines while handling non-WSIO events such as card reset. The `wsio_intr_set_cpu_spec()` and `wsio_intr_set_irq_line()` functions are not invoked for non-WSIO events.

- The routines `wsio_intr_activate()` and `wsio_intr_deactivate()` behave in different ways:
  - If drivers using LBI invoke routines `wsio_intr_activate()` or `wsio_intr_deactivate()` while interrupt migration is in progress the routines return failure.
  - If interrupt migration is invoked and driver invocation with the previous calls is in progress, interrupt migration will wait for the calls to complete.
  - If the driver is using a TBI, the driver carries out the migration process. Part of the migration process is disabling and later enabling card interrupts. This can involve invocation of `wsio_intr_activate()`, `wsio_intr_deactivate()` and `wsio_intr_set_cpu_spec()` calls. Since the driver initiates all the processing, the drivers ensure there are no other threads calling these routines and the interrupt migration thread at the same time. WSIO does not be ensure this synchronization.

- When invoking the `wsio_intr_get_assigned_cpu()`, `wsio_intr_get_txn_info()` and `wsio_intr_get_irq_line()` routines, and interrupt migration is in progress, it is not guaranteed the values returned by these routines are the same as those assigned when interrupt migration is complete.

- Only one high availability event, such as card OLAR/D, error handling, processor OL*, ioscan, or interrupt migration will be in progress in the system at any given point in time.
19 Creating a Software Depot

This chapter contains the following sections that describe the **Software Depot** (SD) creation techniques. These techniques are necessary for an HP-UX Driver Developer to package a driver for distribution. The overall process is:

- Creating a Package
  - Software Depot Overview
  - Designing a Software Depot Structure
  - Selecting Product Install Directory on the Host
  - Writing a PSF
  - Writing Control Scripts (if required)
  - Packaging the Software Depot Components
  - Registering the Software Depot (Package)

- Installing the Software Depot

- Managing the Software Depot

Some Terms and Attributes used in this chapter are defined in the Glossary at the end of this Guide.
Software Depot Overview

This section focuses on the basic Software Depot (SD) packaging techniques that a HP-UX Driver Developer needs to know, to package a driver for distribution.

After a software application is developed, files are taken from the programmer’s environment and "Integrated" for distribution. To ensure ease of installation, maintenance and de-installation, software developed on HP-UX is often distributed as Software Depots. A Software Depot (SD) is a directory location on the local or remote host that is used as a “gathering place” for software products. It is a customizable source of software used for direct installations by the host on the network. The SDs are created using the Software Distributor. The C prepares these software files by organizing them into specific products, subproducts and fileset structures. It also uses special information files (see Creating a Product Specification File (PSF)) that are used to help other commands identify, distribute and manage the application. After it is organized, the software is then “mastered” or copied onto CD-ROMs or tapes for further distribution to Users or Customers. The resulting package can also be made “network accessible” to Users.

There are two types of Depots:

1. Directory Depot
   Software in a Directory Depot is stored under a normal directory on the file system (usually /var/spool/sw). This software is in a hierarchy of subdirectories and filesets organized according to a specific media format. A Directory Depot can be “writable” or read-only. When using the Software Distributor software management commands (see “jkg”), refer to a directory depot via its top-most directory. In a CD-ROM depot, this directory would be the CD-ROMs “mount point”.

2. Tape Depot
   Software in a Tape Depot is formatted as a tar archive. Tape depots such as cartridge tapes, DAT and 9-track tape ar referred to by the file system path to the tape drive’s device file. Software in a tape depot must first be transferred to a directory depot before it can be “pulled” by other hosts on the network. A tape depot can be accessed by only one command at a time.

A depot usually exists as a directory location (that is, a directory depot). Therefore, a host may contain several depots. For example, a designated SD server on the network might contain a depot of word processing software, a depot of CAD software, and a spreadsheet software depot, all on the same server.

The SD provides a powerful set of tools for centralized HP-UX software management. Software Depot has a set of commands which are included with the HP-UX operation system. These commands have the following Tasks distributed across them:

- Packaging the Software as a Depot
- Installing Depots
- Managing Depots
- Removing Depots

When working with SD the methods and objects are largely abstract, making it sometimes difficult to visualize objects, processes, and what is actually happening in the system.

Depots contain software objects such as bundles, products and filesets. A depot is similar to an operating system’s root area. Software objects exist only inside the depot or “root”. Objects can only be viewed in the depot with SD commands.

SD commands work on a hierarchy of software objects — bundles, products, subproducts and filesets, that make up the applications or operating systems that are to be managed.
Bundles

A collection of filesets, possibly from several different products, that are “encapsulated” for a specific purpose. Bundles can be stored in software depots and copied, installed, removed, listed, configured and verified as single entities. All HP-UX OS software is packaged in bundles. Bundles, since they are groups of filesets, are not necessarily supersets of products. Customer creation of bundles are not supported.

Products

A collection of subproducts (optional) and filesets. The SD commands maintain a product focus but still allow specifying of subproducts and filesets. Different versions of software can be defined for different platforms and operating systems, as well as different versions (releases) of the product itself. Several different versions could be included on one distribution media or depot.

Subproducts

Subproducts are used to group logically related filesets within a product if the product contains several filesets.

Filesets

Filesets include all of the files and control scripts that make up a product. They are the smallest manageable (selectable) SD software object. Filesets can only be part of a single product but they could be included in several different HP-UX bundles.

Bundles are designed to provide customers with a single installation unit to install when they purchased software products, such as the ANSI/C compiler. Bundles can be used to provide logical groupings for functionality, such as the “web server”.

Bundling the products makes it easy to treat several filesets as a single entity using the SD commands (discussed later). By specifying a bundle, all filesets under that bundle are automatically included in the operation. In general, performing a single operation on a bundle is the same as performing it individually on all the filesets listed in the bundle.

It is important to note that bundles do not eliminate the ability to “pick and choose” the products and filesets to install.

NOTE

A depot is often confused with software objects. Depots can not be installed or moved like a software object. A depot can not exist without at least one software object residing in it.
SD Structure Capabilities

The structure of a Software Depot is shown in Figure 19-1, “SD Structure.”

**Figure 19-1  SD Structure**

SD can be viewed as a 4-tier/3-tier software structure as shown. The bottom tier are “Filesets”, the set of files to be packaged in the depot. A fileset is the atomic element of the SD. Each fileset must appear in at least one, but only one SD-Product. SD-Filesets can only be installed or removed by the user as a whole entity. It is not possible (via SD) to install or remove only a select set of files from a given SD-Fileset. Files which are used for the same basic purpose should be grouped together in a single SD-Fileset. Likewise, files which are not used for the same basic purpose should be kept apart. The top tier are “Products” — a collection of filesets or (optional) subproducts and control scripts. SD-Product is the fundamental component in SD:

- Packaging is done at the SD-Product level.
- Many attributes are specified at the SD-Product level.
- Filesets only exist within an SD-Product.

SD-Filesets which are for the same functionality should be in the same SD-Product.

The middle tier are Subproducts; if an SD-Product contains several filesets, subproducts are used to group logically related filesets.

Bundles are another tier in the software structure in Figure 19-1, “SD Structure.” They are collections of filesets, possibly from several different SD-Products that are encapsulated for a specific purpose. Bundles provide an alternate way for users to view or select software. The SD structure can be viewed as:

Bundle ---> Products ---> Subproducts ---> Filesets

An SD structure can be any of these combinations:

Bundle1---->Product1---->SP1---->Fileset1 and Fileset2
Bundle2---->Product2---->SP4---->Fileset1 and Fileset2
Bundle1---->Product1---->SP3---->Fileset2 and Fileset1
and similarly for SP2 and SP5.

\[ \text{Product1.Fileset1} \neq \text{Product2.Fileset1} \]

**Software Objects Nomenclature**

**SD-Bundles**
SD-Bundle names must be a maximum 16 characters with no underscores or white spaces. The following characters are not allowed in any of the above software objects names:

\[ #, \, ?, \, / \, [], \, *, \, & \]

**SD-Products**
SD-Product names must be sixteen or fewer characters. They should specify the type of functionality that they contain. Underscores are not permitted in SD-Product names. SD-Product names should be written in mixed case, with significant letters capitalized, and contain no white space. Acronyms may be entirely capitalized. Like SD-Fileset names, SD-Product names should be descriptive and unique. No two SD-Products may have the same name.

**SD-Subproducts**
SD-Subproducts names must be sixteen characters or fewer in length and should accurately describe the SD-Subproduct's contents. The name should be descriptive enough that a user selecting software to install or remove can make an informed choice regarding the SD-Subproduct based solely on its name. No user should be forced to look at the underlying SD-Filesets when making this decision. The recommended format for SD-Subproduct names is the same as that for SD-Product.

**SD-Filesets**
SD-Fileset's names must be entirely capitalized. Name must be unique. The maximum length of SD-Fileset names is 14 characters.
Software Depot Package Components

The SD consists of four major components; Packager, Controller, Daemon and Agent. All except the Agent are installed in the directory /usr/sbin/.

1. **Packager (swpackage)** — The swpackage is a standalone utility that takes a **Product Specification File** (PSF), a set of Control Scripts, and the files the user wants to deliver and packages them into an SD depot. A depot can be either a directory tree or a tar archive. This depot can then be distributed to customers, who then install it on their systems using the swinstall command. While the depot is simply a directory structure, it may not install using cp or tar. This is run as a command line user interface only.

2. **Controller** — Sometimes called the **Integrated Controller**, better known to SD customers as those commands used to manage packaged software; swaci, swconfig, swcopy, swdepot, swinstall, swjob, swlist, swreg, swremove and swverify. These commands are delivered as hard links pointing to a single binary. Controller commands are invoked by users to initiate SD actions. A **Graphical User Interface** (GUI) is available for swcopy, swinstall, swlist -i, and swremove.

3. **Daemon** — The (/usr/bin/swagent) coordinates Controllers and Agents, acting as an intermediary. When a controller command, such as swinstall, is executed, swagentd is contacted via a **Remote Procedure Call** (RPC). The swagentd then forks, and the child process issues an expects call to run the Agent swagent.

4. **Agent** — The (/usr/bin/swagent) performs the majority of the work done by SD. The program installs and removes the software. Agents perform “source” and “target” activities. A source agent is a swagent process which is reading a software source, such as a depot, while a target agent is operating on a target. A target can be a root filesystem or a depot.

**Product Specification File**

The master file is where the bundle configuration (attributes) information exists. It specifies the revision of the product, architecture, dependencies, installation path, and other attributes. The details of these and other attributes are explained in detail in the following sections. All the characteristics of a given product are described in a PSF file.

The PSF contains attribute information for all the software objects and it has a structure accordingly:

```plaintext
#Vendor information
    vendor
        Vendor Attributes
    #end vendor
#Category information
    Category
        Category Attributes
    #end category
#Bundle information
    Bundle
        Bundle Attributes
    end #Bundle
#Product information
    Product
        Product Attributes
    #Subproduct information
    Subproduct
        Subproduct Attributes
    end #Subproduct
Subproduct2
    Subproduct2 Attributes
```
The Category information is “optional” and need not be used. An SD can be packaged without Category information. It is used only for selection mechanism.

The package information is shown in Figure 19-2, “SD Package.”
SD Objects Attributes Classification and Flow

Vendor Attributes:
- Object definition — tag, title, and so forth.
- Information attributes — such as Description.

SD-Category Attributes:
- Object definition — tag, title, and so forth.
- Information attributes — such as Description.

SD-Bundle Attributes:
- Object definition — attributes like tag, title, and so forth.
- Information attributes — such as Description.
- Control attributes — OS, OS version, release, architecture, contents, and so forth.

SD-Product Attributes:
- Object definition — attributes like tag, title, and so forth.
- Information attributes — like Description.
- Control attributes — such as OS, OS version, release, and architecture.
- Control scripts — preinstall, postinstall, and so forth.
- Subproducts and Filesets.

SD-Subproduct Attributes:
- Object definition — like tag, title, and so forth.
- Information attributes — like Description.
- Control attributes — like contents.

SD-Fileset Attributes:
- Object definition — Definition attributes like tag, title, and so forth.
- Information attributes — Informative ones like Description.
- Control attributes — OS, OS version, release, and architecture.
- Control scripts — preinstall, postinstall, and so forth.
- Files — sources, binaries and object files to package.

Policies of the PSF Attributes

A PSF is a master file which contains configuration information (attributes) set accordingly. The SD commands operate based on the values set against these attributes.

The attributes are classified into:

Optional attributes: Optional and need not be used, although they still can be included by the user.

Required attributes: These are not optional.
HP software suppliers who need to structure and package software for distribution across SD require some policies for SD structuring and SD attributes. These policies ensure that customers will be more effective in using the software distribution tools if the software they are dealing with is consistently named and structured.

**Vendor Attributes**

The following attributes are to be included in the PSF file and their policies:

- **Tag**
  - Status: Required
  - Description: The short name for the vendor (distributor) of the software.

- **Title**
  - Status: Required
  - Description: The longer name for the vendor of the software.

- **Description**
  - Status: Required
  - Description: A file or text string describing the vendor (displayed only when the user asks to see a description of the software and then selects the vendor description for view).

**NOTE**

The vendor attributes will be that of the Distributor of the product. For example, if company A develops the product and HP distributes it, the vendor attributes will be HP's.

**Category Attributes**

The following are the category attributes and their policies:

- **Tag**
  - Status: Required
  - Description: The identifier for the category object. In general, only the choices listed should be used. There is a one-to-one relationship between category.tag and category.title attributes as explained under the category.title attribute.
  
  **Policy:**
  - **OrderedApps** Specify a product that a customer can obtain from HP.
  - **TrialUseApps** Trial version of an available product.
  - **Patch** Software automatically set if the is_patch attribute is set to “true”; it is reserved for patches.

The following category tags are reserved for patches:

- defect_repair
- hardware_enablement
- enhancement
- general_release
- special_release
- trial_patch
- beta_release
- manual_dependencies
- critical
Title Status: Required

Description: The one line, detailed name for the category. Notice that there is a one-to-one relationship between category.tag and category.title attributes. Use the category.title values exactly as defined:

Ordered Software Applications When category.tag is OrderedApps.
Trial Use Software Applications When category.tag is TrialUseApps.
Ordered HP-UX Bundles When category.tag is OrderedHP-UX.
Additional HP-UX Functionality When category.tag is HPUXAdditions.

These category.titles are for patches and are controlled by the patch creation tools:

— Use “” when category.tag is beta_release.
— Fix corruption when category.tag is corruption.
— Fix a critical defect when category.tag is critical.
— Provide defect repair when category.tag is defect repair.
— Provide enhancement when category.tag is enhancement.
— Fix a hang or abort when category.tag is halts_system.
— Provide new hardware support when category.tag is hardware_enablement.
— General release patch when category.tag is general_release.
— Fix a memory leak when category.tag is memory_leak.
— Requires manual review of dependencies when category.tag is manual_dependencies.
— Fix a system panic when category.tag is panic.
— Use “” when category.tag is special_release.
— Use “” when category.tag is trial_patch.

Description Status: Optional

Description: A file or text string providing a more detailed description of the category. Use only the text strings listed in the Policy since the category.description is “global”, that is, a particular category.description applies to all objects in the depot/on the media with the corresponding category.tag.

Policy:

Ordered Software Applications When category.tag is OrderedApps.
Trial Use Software Applications When category.tag is TrialUseApps.
SD-Bundle Attributes

Tag  Status: Required
Description: The identifier for the SD-Bundle (appears in the Software Selection window as the basic name for the SD-Bundle) — 16 characters maximum. The tag should be unique and descriptive.
Policy: A bundle tag requires one or more characters from “A–Z”, “a–z”, “0–9”, including the first character. No white space characters are allowed. The directory path character “/” is not allowed. The following SDU and shell metacharacters are not allowed:

., : # ; & { } | < > " \ 

Title  Status: Required
Description: The one-line, more detailed name for the SD-Bundle (appears on the Software Selection window and in the swlist one-liner) — 80 characters maximum. Use this chance to expand upon the sixteen character name of the SD-Bundle.
Policy: This attribute will be seen rather often, so it should be carefully chosen to clearly and concisely tell the user what the SD-Bundle is.

Description  Status: Required
Description: A file or text string (limited to 8 Kb) describing the SD-Bundle (displayed only when the user asks to see a description of the software and then selects this description for view). The free-form ASCII style of this attribute allows for detailed description of the software.
Policy: For maximum readability, each line should be limited to seventy-two characters or less. Tabs should not be used in this file/text string.

Revision  Status: Required
Description: The revision (release) number of the SD-Bundle.
Policy: The revision number of a SD-Bundle is of the form major:minor:release(path).

Architecture  Status: Required
Description: The target systems on which this SD-Bundle is to be installed and will run.
Policy: There are three parts to the architecture attribute; os name, os release, and os bits. The attribute is structured like:

<os name>_<os release>_<os bits>

The “os name” portion of the architecture attribute is always HP-UX.
The “os_release” portion of the string must match the targeted OS for which the SD-bundle is ignited.
The “os_bits” portion specifies the os bit configuration the SD-bundle will run on.

- 32 runs only on an OS in 32-bit mode
- 64 runs only on an OS in 64-bit mode
- 32/64 runs on either

For example:

- HP-UX_B.11.11_32 earliest OS is 11.11, 32-bit only
- HP-UX B.11.11_64 earliest OS is 11.11, 64-bit only
Creating a Software Depot

Software Depot Package Components

- HP-UX_B11.11_32/64 earliest OS is 11.11, 32 or 64 bit

Whenever architecture is defined for a software object, the 3 related attributes “machine_type”, “os_name” and “os_release” must be defined as well. These compatibility attributes are the responsibility of the packager.

**os_name**

Status: Required

Description: OS’s on which the SD-Bundle will run:

$(uname -s):$(getconf CS_KERNEL_BITS)

The “getconf CS_KERNEL_BITS” indicates the bit configuration of the running kernel. Possible results are “32” (if the running kernel is 32 bit) and “64” (if the running kernel is 64 bit, and therefore supports execution of 32-bit applications also)

Policy:

- HP-UX — SD-Bundle runs on both 32-bit and 64-bit operating systems. Most 32-bit SD-bundles would use this specification.
- HP-UX:32* — SD-Bundle runs on 32-bit operating systems only.
- HP-UX:*64 — SD-Bundle runs on 64-bit operating system only.

**os_release**

Status: Required

Description: OS releases on which the SD-Bundle is to be installed and will run:

$(uname -r).

Policy:

- B.11.11 means compatible with 11.11 only.
- ?.11.11 means compatible with 11.11 only

**os_version**

Status: Optional

Description: OS versions on which the SD-Bundle will run: $(uname -v). This attribute is not used. If it is set, it must be set to “*”.

Policy: * — allow all versions.

**Category_tag**

Status: Required (If Category object was defined in the PSF).

Description: Visible only from swlist, and so forth, when users specify a software specification to select a class of SD-Bundles from the command line. In general, only the following choices should be used. There is a one-to-one relationship between bundle.category_tag and bundle.category_title attributes as explained under the bundle.category_title attribute.

Policy:

OrderedApps Specify a product that a customer can obtain from HP.
TrialUseApps Trial version of an available product.
HPUXAdditions Additional HP-UX functionality SD-Bundle.

**vendor_tag**

Status: Required

Description: The short name for the vendor (distributor) of the software. This should match the vendor.tag attribute.
Category_title Status:

Obsolete if a category object is defined. Use category object instead of this attribute.

Description: The one-line, more detailed name of the SD-bundle category tag. Note that there is one-to-one relation between bundle.category_tag and bundle.category_title.

Policy:

- “Ordered Software Applications” when bundle.category_tag is “Ordered Apps”
- “Trial Use Software Applications” when bundle.category_tag is “TrialUseApps”
- “Additional HP-UX Functionality” when bundle.category_tag is “HPUXAdditions”

Contents Status: Required

Description: Describes the software contained in the SD-Bundle.

Policy: Should be a list of SD-Filesets with their software specifications, but can be a list of SD-Products with their software specifications. Each software specification should be fully qualified. This means each software specification should explicitly list the revision, architecture and vendor. The revision listed in any software specification is always the product revision. The software specification should not include the SD-Fileset level architecture (fa=) unless the SD-Bundle contains a subset of multi-stream SD-Fileset pairs, if applicable. For example, a typical bundle.contents would be:

Prod1.FILESET1,r=B.11.11,a=HP-UX_B.11.11_32,v=HP, fa=HP-UX_B.11.11_64

SD-Product Attributes

Tag Status: Required

Description: The identifier or name for the SD-Product (appears on the Software Selection window as the basic name for the SD-Product) — sixteen characters maximum.

Policy: SD-Product names must be sixteen or fewer characters. They should succinctly specify the type of functionality that they contain. It is recommended that SD-Product names be written in mixed case, with significant letters capitalized, and contain no white space. Acronyms can be entirely capitalized. Underscores are not permitted in SD-Product names.

Title Status: Required

Description: The one-line, more detailed name for the SD-Product (appears on the Software Selection window and in the swlist one-liner) — 80 characters maximum. Use this chance to expand upon the 16 character tag of the SD-Product.

Policy: This attribute will be seen rather often, so it should be carefully chosen to clearly and concisely tell the user what the SD-Product is.

Description Status: Required

Description: A file or text string (limited to 8Kb) describing the SD-Product (displayed only when the user asks to see a description of the software and then selects this description for view). The free-form ASCII style of this attribute allows for detailed description of the software.

Policy: For maximum readability, each line should be limited to 72 characters or less. Do not use tabs in this file/text string.
Creating a Software Depot

Software Depot Package Components

Revision

Status: Required

Description: The revision (release) number of the SD-Product.

Policy: Because each ISU/ISD product has its own revision cycle, there is not a definitive policy for specifying product revision. ISU partners are free to manage their revision numbers, within the given constraints, in a way that works best for each product.

The suggested revision number of a SD-Product is of the form \texttt{major:minor:release} (path) each product.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{NOTE} & If the product is part of a bundle then, the “revision” attribute has the same value as that of the Bundle. \\
\hline
\end{tabular}
\end{table}

architecture

Status: Required

Description: The target systems on which this SD-Product is to be installed and will run. Summarizes the supported hardware and operating systems.

\texttt{<os name>_<os release>_<os bits>}

The os name portion of the architecture attribute is always HP-UX.

The os release portion should be the minimum OS revision necessary to run the software.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{NOTE} & If the SD-Product can run only on 11.11 and later, the \texttt{os\_release} portion must be B.11.11. \\
\hline
\end{tabular}
\end{table}

Policy: There are three parts to the architecture attribute: “os name”, “os release”, and “os bits”.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{NOTE} & If the product is part of a bundle then, the architecture attribute has the same value as that of the Bundle. \\
\hline
\end{tabular}
\end{table}

The attribute is structured like:

The “os\_bits” portion specifies the OS bit configuration the SD-product will run on:

- 32 — runs only on an OS in 32-bit mode
- 64 — runs only on an OS in 64-bit mode
- 32/64 — runs on either

For example:

- HP-UX.B.11.11.32 — earliest OS is 11.11, 32-bit only
- HP-UX.B.11.11.64 — earliest OS is 11.11, 32-bit or 64-bit
- HP-UX.B.11.11.32/64 — lowest common denominator for multi release compatible software on 11.x (32 or 64-bit OS)

Whenever architecture is defined for a software object, the 3 related attributes \texttt{machine\_type}, \texttt{os\_name} and \texttt{os\_release} must be defined as well.
Creating a Software Depot

Software Depot Package Components

Chapter 19

os_name

Status: Required
Description: OS releases on which the SD-Product will run:

$(uname -s):$(getconf CS_KERNEL_BITS)

The “getconf CS_KERNEL_BITS” indicates the bit configuration of the running kernel which are “32” (if running kernel is 32-bit) and “64” (if the running kernel is 64-bit, and therefore supports 32-bit also.)

Policy:

- HP-UX — SD-product runs on both 32-bit and 64-bit. Most 32-bit product would use this.
- HP-UX:32* — SD-product runs on 32-bit OS only.
- HP-UX:*64 — SD-product runs on 64-bit OS only

NOTE: If the product is part of a bundle then, the os_name attribute has the same value as that of the Bundle.

os_release

Status: Required
Description: OS releases on which the SD-Product is to be installed and will run:

$(uname -r).

Policy:

- ?.11.* — Any 11.0 release
- ?.11.[01][01] — 11.00 through 11.11
- ?.10.* | ?.11.* — Any 10.x and 11.x release
- ?.11.00 — 11.00 only

NOTE: If the product is part of a bundle then, the os_release attribute has the same value as that of the Bundle.

os_version

Status: Optional
Description: OS versions on which the SD-Product will run: $(uname -v). This attribute is not used. If it is set, it must be set to “*”.

Policy: * — Allow all versions

directory

Status: Required
Description: The default, absolute pathname of the directory which is the root of the file system tree under which the SD-Product will be installed. Can be re-mapped by the user if is_locatable attribute is true.

Policy: Use a standard type of location (like /opt/myprod); do not include a trailing slash like (/opt/mywrongslash/).

is_locatable

Status: Required
Description: If set to true, users can install the SD-Product into an alternate product directory.
Policy:

- **FALSE** — SD-Product must be installed into the path specified by `product.directory`.
- **TRUE** — SD-Product may be installed to an alternate location.

**NOTE**
This attribute must be explicitly set to either true or false. The default is true.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Status:</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>category_tag</td>
<td>Required</td>
<td>The short name for the category of the software. This attribute associates the SD-Product with the standard category object. Policy: This should match the proper <code>category.tag</code> attribute.</td>
</tr>
<tr>
<td>vendor_tag</td>
<td>Required</td>
<td>The short name for the vendor of the software. Policy: HP (or whatever company is providing ISV software). This should match the <code>vendor.tag</code> attribute.</td>
</tr>
<tr>
<td>machine_type</td>
<td>Required</td>
<td>Machine types on which SD-product will run: $(uname -m):$(getconf CS_HW_CPU_SUPP_BITS) The “getconf CS_HW_CPU_SUPP_BITS” indicates the OS bit configuration that the hardware supports. Possible results are “32” (only 32-bit HP-UX is supported), “32/64” (both 32-bit and 64-bit HP-UX is supported) and “64” (only 64-bit HP-UX is supported)</td>
</tr>
<tr>
<td>preinstall</td>
<td>Optional</td>
<td>Pathname for the install pre-load SD-Product control script. The script is run by <code>swinstall</code> during the execution phase before loading the software files.</td>
</tr>
<tr>
<td>unpreinstall</td>
<td>Optional</td>
<td>Pathname for the undoing install preload SD-Product control script. The script is run during the <code>swinstall</code> load phase if it is initiated. An unpreinstall script should undo any operation taken by the preinstall script. Policy: unpreinstall scripts need to be carefully crafted and extensively tested.</td>
</tr>
<tr>
<td>postinstall</td>
<td>Optional</td>
<td>Pathname for the install post-load SD-Product control script. The script is run by <code>swinstall</code> during the Load phase after loading the software files.</td>
</tr>
<tr>
<td>unpostinstall</td>
<td>Optional</td>
<td>Pathname for the undoing install post load SD-Product control script. The script is run during the <code>swinstall</code> load phase if recovery is initiated. An unpreinstall script should undo any operation taken by the postinstall script.</td>
</tr>
</tbody>
</table>
configure  Status: Optional
Description: Pathname for the configure for use SD-Product control script. The script is run by `swinstall` or `swconfig` to configure the host for the software, or to configure the software for host-specific information. Configure scripts are run by `swinstall` for all SD-Products after all SD-Products have completed the Load phase. The `swconfig` command can also be used to rerun configure scripts that failed during a normal install.

verify  Status: Optional
Description: Pathname for the verify integrity SD-Product control script. The purpose of the verify script is to check for the correctness of the product or fileset installation and configuration. It is run by the `swverify` command.

unconfigure  Status: Optional
Description: Pathname for the configure undo SD-Product control script. The script is run by `swconfig` or `swremove` to undo host or software configuration originally performed by a configure control script.

preremove  Status: Optional
Description: Pathname for the remove pre-remove SD-Product control script. The script is run by `swremove` during the Remove phase just before removing files.

postremove  Status: Optional
Description: Pathname for the remove post-remove SD-Product control script. The script is run by `swremove` during the Remove phase just after the files have been removed.

SD-Subproduct Attributes

Tag  Status: Optional
Description: The identifier for the SD-Subproduct (appears in the Software Selection window as the basic name for the SD-Subproduct) — sixteen characters maximum.

There is a one-to-one relationship between the “subproduct tag” and “subproduct title” attributes as explained under the “subproduct title” attribute. If a non-standard SD-Subproduct name was created for the SD-Product, the “subproduct tag” attribute should be that name.

SD-Subproducts and SD-Filesets within the same SD-Product can not have the same tag.

Policy: Standard subproduct tag names:

- ManualsByLang  Localized manual pages and product documentation.
- Help  On-line help.
- HelpByLang  Localized on-line help.
- MessagesByLang  Localized messages.
- Demonstration  Demonstration of the product.
- Development  Software development SD-Filesets
- ReleaseNotes  Separate release notes SD-Fileset(s)
Title

Status: Optional

Description: The one-line, more detailed name for the SD-Subproduct (appears in the Software Selection window) — 80 characters maximum. Use this chance to expand upon the standard name of the SD-Subproduct.

There is a one-to-one relationship between the “subproduct tag” and “subproduct title” attributes. If creating a non-standard SD-Subproduct name for the SD-Product, also develop a corresponding non-standard entry for the subproduct.title attribute.

Policy: Standard subproduct titles:

Manual Pages and Documentation

When subproduct.tag is “Manuals”.

Manual Pages and Documentation in Multiple Languages.

When subproduct.tag is “ManualsByLang”.

On-line Help

When subproduct.tag is “Help”.

On-line Help in Multiple Languages

When subproduct.tag is “HelpByLang”.

Messages in Multiple Languages

When subproduct.tag is “MessagesByLang”.

Demonstration of the Product

When subproduct.tag is “Demonstration”.

Software Development

When subproduct.tag is “Development”.

Product Release Notes

When subproduct.tag is “ReleaseNotes”.

Description

Status: Optional

Description: A file or text string (limited to 8Kb) describing the SD-Subproduct (displayed only when the user asks to see a description of the software and then selects this description for view). The free-form ASCII style of this attribute allows for detailed description of the software.

Policy: For maximum readability, each line should be limited to 72 characters or less. Do Not use tabs in this file/text string.

SD-Fileset Attributes

Tag

Status: Required

Description: The identifier for the SD-Fileset (appears in the Software Selection window as the name for the SD-Fileset) — 14 characters maximum.

Policy: SD-Subproducts and SD-Filesets within the same SD-Product can not have the same tag; the fileset.tag cannot be identical to the “SD-Product tag”. SD-Fileset tags must be unique, even across SD-Products. The Fileset tag should be in capital letters.
Title  Status: Required
Description: The one-line, more detailed name for the SD-Fileset (appears in the Software Selection window) — 80 characters maximum. Use this chance to expand upon the fourteen character tag of the SD-Fileset.
Policy: This attribute will be seen rather often, so it should be carefully chosen to clearly and concisely tell the user what the SD-Fileset is. Often, the first position includes an abbreviation of the SD-Product name as well.

Description  Status: Required
Description: A file or text string (limited to 8Kb) describing the SD-Fileset (displayed only when the user asks to see a description of the software and then selects this description for view). The free-form ASCII style of this attribute allows for detailed description of the software.
Policy: For maximum readability, each line should be limited to seventy-two characters or less; tabs should not be used in this file/text string.

Revision  Status: Required
Description: The revision/version number of the SD-Fileset.
Policy: Each SD-Fileset must have the same revision as the SD-Product containing it.
The attributes (architecture, machine_type, os_name, os_release) may be specified not only at SD-product level, but also at SD-Fileset level. SD-Filesets which do provide values for the attributes will use their own values, not those of their parent SD-product. This allows a single SD-product to contain, for example an SD-Fileset compatible with both 32-bit and 64-bit machines, another SD-Fileset specific to 32-bit systems and other with 64-bit systems. The syntax and semantics of SD-Fileset attributes are identical to those of SD-product level.

architecture  Status: Required
Description: The target systems on which this SD-Fileset is to be installed and will run. Summarizes the supported hardware and operating systems.
Policy: There are three parts to the architecture attribute: os name, os release, and os bits. The attribute is structured like:
<os name>_<os release>_<os bits>
The os name portion of the architecture attribute is always HP-UX.
The os release portion should be the minimum OS revision necessary to run the software.
If the SD-Product can run only on 11.11 and later, the os_release portion must be “B.11.11”.
The os bits portion specifies the OS bit configuration on which the SD-Fileset will run on:
• 32 — runs only on an OS in 32-bit mode.
• 64 — runs only on an OS in 64-bit mode.
For example:
• HP-UX_B.11.11_32 — earliest OS is 11.11, 32-bit only
• HP-UX_B.11.11_64 — earliest OS is 11.11, 32-bit or 64-bit
• HP-UX_B.11.11_32/64 — earliest OS is 11.11, 32-bit or 64-bit
• HP-UX_B.11.11_32/64 — lowest common denominator for multi release compatible software on 11.x (32 or 64-bit OS)

**os_name**  
Status: Required  
Description: OS releases on which the SD-Fileset will run:  
$(uname -s):$(getconf CS_KERNEL_BITS)  
The “getconf CS_KERNEL_BITS” indicated the bit configuration of the running kernel. Possible results are “32” (if running kernel is 32-bit) and “64” (if running kernel is 64-bit and, therefore supports 32-bit also)  
Policy:  
• HP-UX — SD-Fileset runs on both 32-bit and 64-bit. Most 32-bit SD-Filesets would use this.  
• HP-UX:32* — SD-Fileset runs on 32-bit OS only  
• HP-UX:*64 — SD-Fileset runs on 64-bit OS only

**os_release**  
Status: Required  
Description: OS releases on which the SD-Fileset is to be installed: $(uname -r).  
Policy:  
• ?.11.* — Any 11.0 release  
• ?.11.[01][01] — 11.00 through 11.11  
• ?.11.00 — 11.00 only

**os_version**  
Status: Optional  
Description: OS versions on which the SD-Fileset will run: $(uname -v). This attribute is not used. If it is set, it must be set to “*”.  
Policy: * — Allow all versions.

---

**NOTE**  
Since the fileset is part of a product or a bundle, the os_version attribute has the same value as that of the product or bundle.

**category_tag**  
Status: Optional  
Description: The short name for the category of the software. This attribute associates the SD-Fileset with the standard category object.  
Policy: This should match the proper category.tag attribute.

**is_kernel**  
Status: Optional  
Description: Defined if the SD-Fileset contains OS kernel files; that is, it contains files which are used to build the kernel. If it is set to TRUE additional policy should be handled in the control scripts which will be explained later in this document.  
Policy:  
• FALSE — The SD-Fileset does not require a system reboot after installation.  
• TRUE — The SD-Fileset requires a system reboot after installation.
is_reboot    Status: Optional

Description: Defined if the SD-Fileset requires a system reboot after installation.

Policy:
- FALSE — The SD-Fileset does not require a system reboot after installation.
- TRUE — The SD-Fileset requires a system reboot after installation.

Prerequisites, corequisites and exrequisites

Status: Optional

Description: Defined the SD-Fileset(s) upon which this SD-Fileset depends. Prerequisites
define SD-Fileset(s) which must be installed (configured) before this SD-Fileset can be
installed (configured). Note that all SD-Filesets are installed and then configured.
Prerequisites specify an install-time dependency. If prerequisites are installed within the
same session as the dependent software, the prerequisites will be installed before its
dependent. Corequisites define SD-Fileset(s) must be configured before the dependent
SD-Fileset can operate correctly (i.e. a run-time dependency). Exrequisites define
SD-Fileset(s) which are required not to be installed when this SD-Fileset is installed.

Policy:
When dependencies exist, state them. Prerequisites and corequisites are declared in the
same manner and must follow the same rules. A dependent is the SD-Fileset that is
declaring the dependency. A requisite is the SD-Fileset that is depended upon. The
requisite(s) can be contained in the dependent's product or in another product. Each
requisite SD-Fileset must be sufficiently specified. A requisite is normally specified as
product.fileset.

The revision specified is that of the requisite's SD-product, not that of SD-Fileset itself, you
should use >= operator when specifying revision. For example, r>=B.11.11 will allow your
dependency to be satisfied by the highest revision greater than or equal to the 11.11
pre-release software.

preinstall    Status: Optional

Description: Pathname for the install pre-load SD-Fileset control script. The script is run by
swinstall during the execution phase before loading the software files.

unpreinstall    Status: Optional

Description: Pathname for the undoing install pre-load SD-Fileset control script. The script
is run during the swinstall load phase if is initiated. An unpreinstall script should undo
any operation taken by the preinstall script.

Policy: Unpreinstall scripts need to be carefully crafted and extensively tested.

postinstall    Status: Optional

Description: Pathname for the install post-load SD-Fileset control script. The script is run by
swinstall during the Load phase after loading the software files.

unpostinstall    Status: Optional

Description: Pathname for undoing the install post-load SD-Fileset control script. The script
is run during the swinstall load phase if recovery is initiated. An unpreinstall script
should undo any operation taken by the postinstall script.

Policy: Unpostinstall scripts need to be carefully crafted and extensively tested.
configure  Status: Optional
Description: Pathname for the configure for use SD-Fileset control script. The script is run by swinstall or swconfig to configure the host for the software, or to configure the software for host-specific information. Configure scripts are run by swinstall for all SD-Filesets after all SD-Filesets have completed the Load phase. The swconfig command can also be used to rerun configure scripts that failed during a normal install.

verify  Status: Optional
Description: Pathname for the verify integrity SD-Fileset control script. The purpose of the verify script is to check for the correctness of the product or fileset installation and configuration. It is run by the swverify command.

unconfigure  Status: Optional
Description: Pathname for the configure undo SD-Fileset control script. The script is run by swconfig or swremove to undo host or software configuration originally performed by a configure control script.

preremove  Status: Optional
Description: Pathname for the remove preremove SD-Fileset control script. The script is run by swremove during the Remove phase just before removing files.

postremove  Status: Optional
Description: Pathname for the remove postremove SD-Fileset control script. The script is run by swremove during the Remove phase just after the files have been removed.

request  Status: Optional
Description: Pathname for the request interactive SD-Fileset control script. The script is run by the swask command or the swinstall or swconfig commands with the “ask” option. The script requests response from the user as part of software installation or configuration.
Policy: Since request scripts are user-interactive, they cannot be used in Cold Install or unattended installation situations. Because of this, their use is not recommended.

File Definitions
The files contained in each SD-Fileset must be specified within the PSF. A file can be in one-and-only-one SD-Fileset. A file cannot reside in multiple SD-Filesets.

NOTE  A symlink (4) command can cause confusion during installation. Suppose a link /usr/bin/x which is a symlink to /usr/bin/y. If the product wants to install /usr/bin/x as a file or directory, SD will install the file or directory at /usr/bin/y and leave /usr/bin/x as a symlink to /usr/bin/y. SD will follow the links down and will not replace them.

file permissions  Status: Optional
Description: Explicitly specifies default permissions for the files being packaged into the SD-Fileset. This keyword applies only to the SD-Fileset in which it is defined. Later definitions within a SD-Fileset replace previous definitions. In the default condition, destination files receive permissions from their respective source files.
Use:

file_permissions [-m mode (octal) | -u umask] [-o [owner[],][uid]] [-g [group[],][gid]]
directory     Status: Optional
Description: Specifies a source directory in which subsequently listed filenames are located (for this SD-Fileset only). The source directory can be either an absolute or relative pathname, relative pathnames will be interpreted relative to the current working directory in which swpackage is invoked.
Use:
directory source_dir [= destination_dir]

file          Status: Required
Description: Specifies the files to be packaged into a SD-Fileset. When the directory keyword is used, pathnames are relative to the directory specified; otherwise, pathnames must be absolute. The directory keyword must be used for recursive file specification.
This keyword is also used with directories when explicit permissions are given. Permissions can be included on this line or previously with the file_permissions keyword. No explicit file entry is needed for a directory when the file_permissions and directory keywords are implemented in sequence.
Use:
For explicit naming:
file [-m mode (octal)] [-o [owner[,]][uid]] [-g [group[,]][gid]] source [destination]
For implicit (recursive) naming:
file *
For explicit naming of directories:
file [-m mode (octal)] [-o [owner[,]][uid]] [-g [group[,]][gid]] source_dir [destination_dir]

Combining the file definitions:
File definition keywords can be combined for many purposes. Some possibilities are shown here. To include all files from /src/develop/ddk/driver/qlispdrv directory on the build system in the /opt/ddk/sampldrvs/qlisp/ on the target system:
directory/src/develop/ddk/driver/qlispdrv=/opt/ddk/sampldrvs/qlisp/file
file *
To implement the previous example giving the directory qlisp and all of the files in * the permission set 755, owner root, and group users:
file_permissions -m 755 -o root -g users
directory/src/develop/ddk/driver/qlispdrv=/opt/ddk/sampldrvs/qlisp/file
file *
To explicitly provide the permission set 555, owner root, and group root to the SD-Product directory /opt/ddk/:
file_permissions -m 555 -o root -g root file /src/develop/ddk /opt/ddk
Control Scripts

These are the shell scripts which act on the product files. Their functions are self explanatory but can be chosen by the developer:

- preinstall
- postinstall
- preremove
- postremove
- configure
- request
- verify
- checkinstall, and so forth.

A product might not have all of the control scripts. The needed scripts must be specified in the PSF. Each Fileset in a product can have control scripts of its own.

Control Scripts Overview

Control scripts can be used to manage the SD-Product more efficiently. The use of control scripts is optional. This section describes in detail all the control scripts one needs to use during SD package. There are around nine named control scripts that have specific function within the SD environment. Each script should fulfill the requirements of the SD-UX software distribution tools/commands and of the particular fileset.

The control scripts are separate shell scripts which are included as attributes in the PSF for specific purpose and are to be used with SD commands/tools.

The following control scripts are available to be used in SD-UX:

- `checkinstall` Run from within a `swinstall` session.
- `preinstall` Run from within a `swinstall` session.
- `postinstall` Run from within a `swinstall` session.
- `configure` Run from within a `swinstall` session and also from `swconfig`.
- `verify` For use with `swverify`.
- `checkremove` Run from within a `swremove` session and also from `swconfig`.
- `unconfigure` Run from within a `swremove` session and also from `swconfig`; the reverse of `configure`.
- `preremove` Run from within a `swremove` session.
- `postremove` Run from within a `swremove` session.
- `request` Run from `swinstall` and `swremove` session; user interaction script.
Control Script Levels

The control scripts can be used at two levels of SD-UX:

- Product level
- Fileset level

Product Level Control Scripts

Control scripts for use with SD-UX are associated both with filesets and with products. A product can have its own set of control scripts; each fileset within that product can also have its own control scripts. The product level control scripts will be run whenever any fileset within that product is selected for installation, removal, or verification. Therefore the activities in product level scripts must pertain to all software delivered in that product, but not to any fileset in particular.

Any actions which might be included in a particular control script for every fileset in a product should instead be in the appropriate product level control script.

Fileset Level Control Scripts

The scripts for a particular fileset must pertain only to the installation, verification, configuration, or removal of that fileset, and not to any other fileset or to the product that the fileset is a part of.

Special Types of Filesets

There are several types of filesets based on the special actions that need to be taken on the system.

Reboot Fileset

A reboot fileset is one which requires that the target system be rebooted as part of software installation and configuration. The fileset's `is_reboot` attribute in the PSF containing the fileset must have a value of TRUE for a reboot fileset. If one or more selected filesets has a reboot flag, the system will be rebooted after all selected filesets have been installed or removed.

Kernel Fileset

A kernel fileset contains components for inclusion in the system kernel. Selection of one or more kernel filesets requires that the system kernel be rebuilt as part of the installation process. The fileset's `is_kernel` attribute in the PSF containing the fileset must have a value of TRUE for a kernel fileset. For HP-UX, all kernel filesets are also reboot filesets.

Prerequisite Fileset

A prerequisite fileset is one whose correct installation is required for proper install-time operation. A fileset should specify a prerequisite fileset only if it needs the prerequisite fileset configured before the fileset configure script is run.

Corequisite Fileset

A corequisite fileset is one whose correct installation is required for proper run-time operation.
Control Script Format

The file should be a shell script (as opposed to a binary) and written to be interpreted by the Posix.2 shell /sbin/sh. Korn shell (formerly /bin/ksh) syntax is acceptable to the Posix.2 shell, however the script must be run in the Posix shell (that is, the first line of any script must be "#!/sbin/sh" - not "#!/bin/ksh" or "#!/usr/bin/ksh"). A script written in “csh” will not be supported.

The file should have a simple header similar to the following example. The first line must indicate the interpreter shell by using the “#!” convention. Included in the header should also be comment lines which state the Product and Fileset to which the script belongs, the name of the script, the revision string as required by the what(1) command, and a simple copyright statement.

Example:

```bash
#!/sbin/sh

# Product:
# Fileset:
# configure

#!/sbin/sh

# (c) Copyright Hewlett-Packard Company, 2002

Scripts without a specific shell identifier (that is, #!/sbin/sh) will be run under /sbin/sh by default. This is not recommended, because it may result unambiguous coding and testing approaches.

Execution Environment

This section contains details both of the file system location where the control scripts are delivered, and the environment variables to be called in those control scripts.

Location of the File System

When installing from magnetic tape or from a depot across the network, the checkinstall, preinstall, postinstall for a particular Fileset will be downloaded to a temporary directory from which they will be invoked: /var/tmp/<CATALOG_DIR>/catalog/<PRODUCT>/<FILESET>/control_script.

The form of the <CATALOG_DIR> is: AAAa<pid>, where <pid> is the swinstall process ID number. These files are delivered to that location from the depot immediately after product selection has completed, at the beginning of the Analysis phase and before any system checks have begun. The temporary directory is removed automatically upon exiting swinstall. After successful fileset installation, these control scripts and all other control scripts will be located in the Installed Product Database (IPD). They will be delivered to that location from the depot as part of the installation of the fileset’s other files:

```
/var/adm/sw/products/<PRODUCT>/<FILESET>/control_script
```

The location of the IPD is relative to the root directory under which the software installation is done. If the installation is to an alternate root, /mnt/disk2 for example, then the IPD for that software will be under:

```
/mnt/disk2/var/adm/sw/products/<PRODUCT>/<FILESET>
```

All necessary directories under /var/adm/sw will be created by the SD process. All files under those directories will be filled by SD initiated processes.
Environment Variables

The control scripts are invoked as the super user, that is, with an effective uid of 0. All of the control scripts are invoked without arguments. Information is learned by executing commands from within the scripts, by calling other utilities, and by environment variable values. The following environment variables will be passed to each control script. They can be tested for a particular value or used to construct another environment variable but should never be set or altered within a control script.

**SW_PATH**
A path to commands which can be called from within control scripts. The path for HP-UX 11.* is “/usr/lbin/sw/bin:/var/adm/sw/sbin:/sbin:/usr/bin:/usr/ccs/bin”.

**PATH**
The path to commands which can be called from within control scripts, this includes **SW_PATH** and is set when **control-utils** is sourced. The path for HP-UX 11.* is “${SW_PATH%:}:/usr/sbin:/usr/lbin/sw”.

**SW_ROOT_DIRECTORY**
The path to the root directory under which the software is being installed. This will normally be “/”, but will be different in the case of an alternate root install. It will always be “/” for configure, swverify, and unconfigure. This value must be prefixed to the path of any installed file but not to commands to be executed. When prefixing this variable to a path, do not use a “/” as the leading character in the path, since that will expand to “//”. Most of the time this is harmless, but some uses of the resulting path involve string comparisons, and the “//” could cause a failure. The value is set on the command line as a suffix to the target system name, which is distinguished by the “@” symbol.

```
swinstall \* @ hpmysys:/mnt/disk30...
```

Be aware that the installation to an alternate root requires that the “-r” option be provided when the SD command is invoked, both in interactive and noninteractive mode. Also, changing product location (described below) has no effect on this variable.

**SW_LOCATION**
The directory where the product is located. This will normally be “/” but will be different in the case of a relocated product. When prefixing this variable to a path, do not use a “/” as the leading character in the path, since that will expand to “/”.

The value is derived from a path which can be appended to the product name of each locatable product on the command line or in the software selection file.

```
swinstall... Accounting:/opt/alternate1/acct...
```

In the GUI, a menu button labeled “Change Product Location” (accessible from the Actions pull-down menu) permits the user to redirect the installation of a product to a nondefault directory. This action is denied if the product is not locatable. If no path is specified or the product is not locatable, $SW_LOCATION is the default path defined on the media.

**SW_SOFTWARE_SPEC**
A string containing the full, unambiguous specification of the current software. The format is PRODUCT[.FILESET], l=$SW_LOCATION, r= revision, a= architecture, v= vendor.

**SW_CONTROL_DIRECTORY**
The directory path where the script currently being executed is located. See the “Location of the File System” section for locations of control scripts. This value is necessary if the control script invokes a subscript which is shipped along with the fileset.
**Creating a Software Depot**

**Software Depot Package Components**

---

**SW_SESSION_IS_KERNEL**

This variable is set to TRUE only if there are one or more kernel filesets which will be installed. It is unset at all other times. Its value should be tested by control scripts to learn whether a kernel rebuild and subsequent system reboot will be done.

**SW_SESSION_IS_REBOOT**

This variable is set to TRUE only if there are one or more reboot filesets which will be installed. It is unset at all other times. This variable will tell that reboot filesets have been selected, not that they have been installed or that the system has been rebooted because of them.

---

**Relocating the Product**

Many non-OS SD-Products can be installed to a directory other than the default directory specified by the developer. This action is known as relocating the SD-Product. In the GUI, the menu button labeled “Change Product Location” (accessible from the Actions pull-down menu) can be used to implement this redirection. In the command line mode, the product directory can be specified by appending that directory to the name of the product to be installed:

```
swinstall -s /opt/ddk/sampldrvrs/qlisp/qlispdoc QLISPDOC:/opt/docs.
```

This will relocate the product QLISPDOC to new location /opt/docs. Here SW_LOCATION is now /opt/docs. It is possible to relocate products while installing to an alternate root:

```
swinstall -s /opt/ddk/sampldrvrs/qlisp QLISP:/opt/ddk/sampldrvrs/qlisp@newhost:/opt/ddk
```

Here both SW_LOCATION and SW_ROOT_DIRECTORY are changed /opt/ddk/ and “newhost”.

---

**System Commands**

Script developers should use only commands and syntax which comply to the IEEE Standard 1003.2 (Posix.2) standard. Compliance will not guarantee that all operating systems on which the scripts might run will conform to Posix.2.

Since control scripts are run under the POSIX shell, any built in command that works for the shell should work in a control script. A list of commands are shown in Table 19-1, “System Commands.”

---

**Table 19-1**

<table>
<thead>
<tr>
<th>alias</th>
<th>export</th>
<th>let</th>
<th>shift</th>
<th>unset</th>
</tr>
</thead>
<tbody>
<tr>
<td>bg</td>
<td>fc</td>
<td>newgrp</td>
<td>test</td>
<td>until</td>
</tr>
<tr>
<td>case</td>
<td>fg</td>
<td>print</td>
<td>time</td>
<td>wait</td>
</tr>
<tr>
<td>cd</td>
<td>for</td>
<td>pwd</td>
<td>times</td>
<td>whence</td>
</tr>
<tr>
<td>command</td>
<td>getopt</td>
<td>read</td>
<td>trap</td>
<td>while</td>
</tr>
<tr>
<td>continue</td>
<td>hash</td>
<td>readonly</td>
<td>typeset</td>
<td></td>
</tr>
<tr>
<td>echo</td>
<td>if</td>
<td>return</td>
<td>ulimit</td>
<td></td>
</tr>
<tr>
<td>exec</td>
<td>jobs</td>
<td>set</td>
<td>unalias</td>
<td></td>
</tr>
<tr>
<td>exit</td>
<td>kill</td>
<td>select</td>
<td>unmask</td>
<td></td>
</tr>
</tbody>
</table>
There is a very brief list of non-built-in commands which can be used safely by checkinstall, preinstall, and postinstall scripts. The commands in this list and the syntax used in control scripts must be available on all supported architectures and operating systems, and they should comply with the Posix.2 standard. This document will refer to the set of non-built-in commands as Posix.2 commands, refer to Table 19-2, “Posix.2 Commands for All Control Scripts,” even though some of them are HP specific and are not Posix compliant.

### Table 19-2  Posix.2 Commands for All Control Scripts

<table>
<thead>
<tr>
<th>Command</th>
<th>Command</th>
<th>Command</th>
<th>Command</th>
<th>Command</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>awk</td>
<td>cmp</td>
<td>In</td>
<td>pa</td>
<td>tail</td>
<td></td>
</tr>
<tr>
<td>ch-rc</td>
<td>cp</td>
<td>ls</td>
<td>rep</td>
<td>uname</td>
<td></td>
</tr>
<tr>
<td>chgrp</td>
<td>ioscan</td>
<td>mkboot</td>
<td>rm</td>
<td>wc</td>
<td></td>
</tr>
<tr>
<td>chmod</td>
<td>lifcp</td>
<td>mkdir</td>
<td>rmdir</td>
<td></td>
<td></td>
</tr>
<tr>
<td>chown</td>
<td>lifls</td>
<td>mv</td>
<td>sed</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

When an OS update is done, a situation which could place incompatible versions of essential commands on the system, the user must first fetch the correct version of SD-UX. Part of that process includes getting usable versions of these commands to a location on the system where control scripts will access them.

These commands will be preserved for the duration of the installation process for use by the checkinstall, preinstall, and postinstall scripts. Use of $SW_PATH in all control scripts to set PATH, as shown in the sample control scripts, ensures that the preserved version of the command is accessed rather than the newly installed, possibly incompatible versions.

In addition to the commands previously listed, the commands shown in Table 19-3, “Additional Preserved Commands,” will be preserved during an OS update. These commands are listed separately because they are not appropriate for use in control scripts.

### Table 19-3  Additional Preserved Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Command</th>
<th>Command</th>
<th>Command</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>ar</td>
<td>ccom</td>
<td>cpp</td>
<td>make</td>
<td>reboot</td>
</tr>
<tr>
<td>cc</td>
<td>config</td>
<td>ld</td>
<td>mount</td>
<td>umount</td>
</tr>
</tbody>
</table>

### SW DIST Commands

In addition to the SD executables there are tools that are used by SD-initiated processes and are delivered in the SW-DIST product. These include:

- `/usr/lbin/sw/control_utils`  General purpose utility functions for all control scripts
- `/usr/sbin/update-ux`  Utility used to update to a newer OS.

The installation of system software, especially during development cycles, must be preceded by importing the most recent version of the SW-DIST product by swinstall-ing it from the OS-Core depot. update-ux installs the latest SW-DIST automatically.
General Control Script Actions

This section covers the details of the control script actions such as inputs from the user, what the control script output is, and any exit values returned from the control scripts.

Input

A control script must never require interaction from a user. All control scripts must be designed to run to completion, whether successful or not, without intervention (an exception being “request” script). There are commands for which no input is required under all normal circumstances, yet in some unique configuration require user confirmation. These situations must be anticipated by the developer.

Output

Standard output and standard error from control scripts are directed to the `swagent` log file:

```
$SW_ROOT_DIRECTORY/var/adm/sw/swagent.log
```

The log file is located relative to the root directory under which the software is installed. The output consists of messages initiated by the `swagent` process and messages resulting from the control script actions. If there are any intermediate events of which the user should be informed, the resulting messages are appended to the log file. Output from the control script, including output from commands the control script invokes, is appended to the log file. Generating those messages is the responsibility of each control script. Notice there are no further messages from `swagent` if the script succeeds, that is, if it exits with a “zero” value. The script must not write directly to the console or attempt any other method of writing to the display. Commands called by control scripts must not write to the console. Instead, the output from the script will be handled by the calling process and appended to the log file. Output should be simple echo statements to `stdout`; they will be captured by the parent process and appended to the log file.

For example, the output to the log file if a warning is returned from the `checkinstall` script might look like:

```
* Running the "checkinstall" script for "FUEL.UDMH"

WARNING: The product "OXIDIZER.N2O4" is already present on the system. Proceeding with the installation of FUEL.UDMH can result in a volatile configuration. If used with the Instant Ignition product, results could include locally elevated levels of temperature and noise, reduced visibility, loss of volatile memory, burned monitor phosphor, and severe disk fragmentation. Your mileage may vary.

WARNING: The "checkinstall" script for "FUEL.UDMH" had a warning. (exit code "2"). The script location was: "/var/tmp/AAAa01234/catalog/FUEL/UDMH/checkinstall"

* This script had warnings but the execution of this product will still proceed. Check the above output from the script for further details.
```

At the end of the `checkinstall` execution in the Analysis Phase, the log will have a message:

```
WARNING: The Analysis Phase had warnings. See the above output for details.
```

Exit Values

The calling process acts upon the value returned by each control script. The action triggered by the return value differs depending on the type of script being run. The prescribed return values are:

- **0 — SUCCESS**
  - No impediment to installation or removal was found. The process can proceed to the next task in the sequence.

- **1 — FAILURE**
  - The action differs greatly according to script type. Refer to the individual script discussions for details.
2 — WARNING

A condition exists or an event was detected which does not impede the installation or removal process but should be presented to the user. This message type is also to be used if the possibility of a failure exists, yet there is not enough information to guarantee a failure.

3 — EXCLUDE

The action differs according to script type. This exit value is only valid in the checkinstall, and checkremove. Refer to these individual script discussions for details.

In the checkinstall and checkremove script scenarios, the EXCLUDE exit value has far reaching ramifications when used with filesets which are dependencies for other filesets. It is highly recommended that this return value be implemented only for isolated filesets, no other fileset should depend (prerequisite or corequisite) on a fileset which might be excluded.

11 — GLOBAL_ERROR

A serious condition exists which demands that the install process not continue past the Analysis Phase. This will not cause an immediate exit of the process, but will prevent any selected software from being installed. The remaining check scripts will be executed and they may emit messages to the log file. The system will be checked for available disk space. The session will exit after cleaning up.

Required Actions

The SD requirements do not insist that any one control script be present for correct operation. An absent control script file is functionally equivalent to an existing script which returns SUCCESS. If a fileset’s requirements do not include any operations in a particular control script, then that script must be omitted from the fileset’s delivery. It is the packager’s responsibility to verify that a control script is needed or not. A control script must be re-executable. The developer must include safeguards against a failure for an invocation of a script that has already been executed. A script which makes a change to the system the first time it is executed must have protection against error for subsequent executions.

Permitted Actions

The set of actions that a particular type of script is allowed to take is discussed in the details for each script type in the “Guidelines for Control Scripts” section.

Prohibited Actions

There are some actions that must not be initiated from any control script:

- Do not shut down or reboot the system.
- Do not change the system’s INIT state.
- Do not initiate a kernel build.
- Do not remove any control scripts after execution.
- Do not leave debugging enabled. The -u and -x shell options must be unset.
- Do not remove any of the fileset’s files in the Installed Product Database.
- Do not alter files owned by a different fileset.

Do not modify startup/shutdown scripts either directly or indirectly. Each fileset that has a startup/shutdown script should have a configuration file under /etc/rc.config.d which contains system configuration parameters unique to that fileset. The startup/shutdown script should be completely data free and should
require no modifications. Modifications to the configuration data file should be made with extreme caution. Changing system configuration parameters that may have been established deliberately by the user is not something to be done casually or silently.

Do not change any links between startup/shutdown scripts. The links should exist on the media, and should remain as shipped after configuration.

When an application installed under /opt requires a startup script, moving files and creating links are valid components of the install process. This scenario presents an exception to the above prohibition.

control_utils

The function library /usr/lbin/sw/control_utils is delivered with the SW-DIST product. It is a library of Posix.2 shell compatible functions which perform commonly used tasks. The control_utils file should be sourced by each control script. If the control script does not require anything from the control_utils file, omit this. The establishment of this function library allows a standardized set of functions to be available to all filesets.

Installed State

The result of a software installation attempt will leave the product or fileset in a particular state on the system. The state of the installed software can be displayed by executing the following command:

swlist -l fileset -a state <PRODUCT>.<FILESET>

The output will look like:

#  Target: hpmysys:

<PRODUCT>.<FILESET> <installed_state>

The function get_install_state in control_utils can also be used to retrieve a fileset's installed state.

Some of the several possible installed states:

- Configured — the software has been installed, and the configure script has been run successfully
- Installed — the software has been installed, and:
  — the configured script has not yet run
  — the configured script has run and failed
  — the unconfigure script has been run
- Transient — the software in the fileset's product is in the process of being installed or removed. This state is established for each of the product's filesets that is marked for installation. The state of all marked filesets in a product is changed to transient when the first fileset in that product enters Load (or Remove) phase. The state changes to installed when the last of the product's marked filesets has been successfully installed.
- Corrupt — the software currently on the system is an incomplete state. This might be the result of a software installation that had an abnormal termination while the fileset was in transient state.
Recommendations

There are number of conditions that a developer should not assume when developing a control script. Many of them are mentioned throughout this document, but they bear repeating:

- The directory under which software is being installed might not be “/”. The script should prefix \$\{ROOT\} from `control_utils` to ensure that paths are accurately defined.
- One of more volumes on the system might not be configured for long file names. Limit all file names to 14 or fewer characters including all extensions.
- The system might not be quiescent. A binary might be busy (e.g., open for execution) while its fileset is being updated. In this event, the file's basename will be renamed to \$\{basename\} and the incoming file will be placed on the system.
- The current working directory (\$\{CWD\}) must not be assumed or considered in a control script.
- Non-standard features in the super-user’s home environment will not be available.
- A command might not execute correctly. Checking the return value for success is prudent.
- A command’s functionality might have changed from a previous release. When in doubt, refer to the manpages for the new OS release.
- Other filesets might (or might not) be already present on the system. Test for the installed state of a fileset before acting on it.
- One or more filesets might not have been installed correctly. Be sure to explicitly specify all needed requisite filesets.
- When using \$\{ROOT\}, \$\{SW_ROOT_DIRECTORY\}, \$\{SW_LOCATION\}, or \$\{SW_CONTROL_DIRECTORY\} as a prefix to a path, do not also use “/” as the first character in the rest of the path. Each of these variables ends in a “/”, which would produce a “//” in the expanded path. That causes problems if the pat is used in string comparisons.
- Use \$\{SW_SYSTEM_FILE_PATH\} instead of defining the path for the location of the kernel.
- Use the `chrc` command to modify parameters within `/etc/rc/config`.
- Use utilities provided in `control_utils` whenever possible. For example, use the `newconfig_msgs` function instead of writing your own and placing it in your control script.

Performance Considerations

- Use Posix.2 shell built-ins wherever possible. The Posix.2 shell has a rich set of capabilities which can be used in place of many common commands. Creative use of built-ins, Input/Output manipulations, pattern matching and conditional expressions make scripts more code efficient and quicker.
- Source `control_utils` only when using the functions it provides.
- The `/etc`, `/dev` and `/var` directories are guaranteed to exist for cold install, diskless install and all forms of update. You should never create these directories.

Backward Compatibility Considerations

Several new functions are added to the `control_utils` file for the 11.11 release. These functions will not be available on systems earlier than 11.11 that haven’t been updated with the 11.11 version of SD. If the developer wants to use a function from the 11.11 `control_utils`; the easiest thing to do is to copy the functions wanted to a file and then deliver those functions as a `control_file`.
As an example, the **msg** function is being used for the 11.11 version of `control_utils`, the developer could copy the function to your packaging directory. Then the file containing the function (let’s call it “foo” to keep with UNIX tradition) would be included as a `control_file` within the product or fileset that has control scripts needing this new functionality.

To include “foo” you would add this line to your PSF:

```
control_file foo
```

Then with your control scripts needing the new functionality you would source your `control_file` like this:

```
. $SW_CONTROL_DIR/foo
 UTILS= "/usr/lbin/sw/control_utils"
 if [[ ! -f $UTILS ]]
 then
   echo "ERROR: Cannot find $UTILS"
   exit 1
 fi
 . $UTILS
 exitval=$SUCCESS
```

### Guidelines for Control Scripts

This section covers the guidelines for all the control scripts of SD. This involves a description of all aspects required to write and execute a control script in the SD environment. It describes in detail the purpose and order of script execution, and actions of all control scripts in SD.

#### Checkinstall

**Purpose of checkinstall script**

The purpose of a checkinstall script is to ensure that the target system has no product or fileset specific conditions which would cause either an installation failure or a runtime failure. This specifically excludes conditions that are tested elsewhere in the `swinstall` process such as inadequate disk space, unmounted volumes, unresolved fileset dependencies, inappropriate architecture, and so on.

**Order of script execution**

The checkinstall scripts are called while the installation is in the Analysis phase. Execution occurs after the check for mounted volumes and the check of currently installed software has completed, and before the check for product and fileset dependencies and the analysis of available disk space. The checkinstall script is run only when the user has invoked `swinstall`, and not during `swcopy` or `swremove`. The checkinstall script for a particular product or fileset is invoked in series with the checkinstall scripts of all other selected products and filesets.

The checkinstall scripts for selected products and filesets are all executed before any installation is begun. If there is a product level checkinstall script, it is called prior to the calling of any checkinstall scripts belonging to that product’s filesets. Unlike the preinstall and postinstall scripts, checkinstall works with all filesets in one product before moving on to the next product.

The explanation of the phases mentioned can be found in “Step 5: Packaging the Components” and the “Installing a Depot” section.

**Checkinstall system commands**

The commands available to the checkinstall script under all conditions are relatively few. They include the Posix.2 commands listed in the “System Commands” section.
Commands that are part of the SW-DIST product. For example, the `get_sysfile` command can be used to extract the `/stand/system` file to check for a certain kernel configuration. The path to the command might not be established in the environment’s `$PATH`. In this case the command must be specified with its full path.

**Checkinstall actions**

**Output:**

As mentioned in the “Output” section.

**Exit values:**

The calling process acts upon the value returned by each `checkinstall` script. These values correspond to environment variables set in the `control_utils` code. The prescribed return values are:

**0 — SUCCESS**

No impediment to installation found. Installation will proceed immediately if in non-interactive mode. If in the interactive mode, the process will wait for the user to select the `start_install` action item.

**1 — FAILURE**

In a product level `checkinstall` script, the entire product will be unselected, along with all other products which had any of this product’s filesets as a requisite. In a fileset level `checkinstall` script, the fileset will be unselected along with all other filesets which had any of this fileset as a requisite. The error message should explain why. Any remaining filesets and products unaffected by the failure will remain selected for installation. Any remaining `checkinstall` scripts will be run in install order.

**2 — WARNING**

Possible conflict explained in warning message. The selection of the current fileset and its prerequisite and corequisite filesets will remain intact, barring other failures. The warning message should explain the condition.

**3 — EXCLUDE**

The current fileset will be unselected. A message to the log file should explain the unselection. Any remaining filesets unaffected by the unselection will still be selected for installation. Any remaining `checkinstall` scripts will be run in install order.

**11 — GLOBAL_ERROR**

This value will cause the install session to exit before any software installation occurs. The Analysis Phase will proceed through all the checks in order to accumulate all analysis information. This exists to allow a `checkinstall` script prevent installation if test results warrant such action.

**Required Actions:**

The actions of a `checkinstall` script must be extremely unobtrusive since there is no commitment to installing at the time the scripts are run. The `checkinstall` script, like all control scripts, must be re-executable. It is possible for a `checkinstall` script to be executed numerous times within a single install session.

**Permitted Actions:**

Typical actions valid from within a `checkinstall` script at any time are:

- Test for selected software.
- Test the system’s hardware configuration.
- Test kernel configuration.
- Test the init state of the system.
- Test the system’s I/O structure.
Creating a Software Depot
Software Depot Package Components

- Test for software already installed.
- Test the revision, or other attributes of installed software.

Prohibited Actions: Mentioned in the “Required Actions” section. In addition the checkinstall script must not:

- Create new files, other than temporaries that must be removed before exiting.
- Copy, move, remove, or modify existing files.
- Kill processes.
- Spawn processes that will linger after the script exits.

Configuration Directory and Functions

The newconfig directories and functions are used in configure and preinstall scripts. The handling of files delivered to the newconfig directory is done by newconfig functions. These functions are required for the configure, pre- and post-install scripts. The newconfig directory for the OS products is /usr/newconfig.

For example, if /etc/disktab is fileset that is to be changed with each major release, then the “working” file is the file at the ultimate path. This file may or may not have been modified by the system administrator after being placed there by a previous installation.

Example: /etc/disktab

The “previous” file is the file delivered by the last software installation of the current fileset. This file should be unchanged from the way it was delivered.

Example: /usr/old/usr/newconfig/etc/disktab

The third file of this discussion is the one being installed by the current invocation of swinstall:

The “new” file is the file that is newly installed by swinstall.

Example: /usr/newconfig/etc/disktab

Preinstall

Purpose of Preinstall control script

The purpose of a fileset’s preinstall script is to prepare the system for installation of the fileset. By the time the preinstall script for a particular fileset is executed, the actions of that fileset’s checkinstall script have determined that there are no fileset-specific impediments to installing the fileset. Then the Analysis Phase has ensured that there are no known system-specific impediments to installing the current fileset. All that remains is to prepare the system and to begin installing files. The steps to prepare the system are done by the fileset’s preinstall script. A typical preinstall script is intended to kill processes that could interfere with installation, for example daemons that keep an executable file opened. Removal of obsolete software is done in a preinstall script.

Order of script execution

The preinstall scripts are called while the installation is in the Execution Phase. Each fileset’s preinstall script is executed just before that fileset’s files are installed onto the target system. A product level preinstall script is called prior to the calling of any preinstall scripts belonging to that product’s filesets.
Creating a Software Depot

Chapter 19

Preinstall system commands

The commands available to the preinstall script under all conditions are relatively few. They include the Posix.2 commands that are part of the SW-DIST product listed in the “System Commands” section.

Preinstall actions

Output:

AS mentioned in the “Output” section.

Exit Values:

The calling process acts upon the value returned by each preinstall script. The prescribed return values are:

0 — SUCCESS All processes were run successfully.
1 — FAILURE A FAILURE occurred, indicating that the installation and configuration is certain to fail in some way. The failure message should explain what the failure was. If a kernel fileset’s preinstall script return FAILURE, the install process will exit if in noninteractive mode and will suspend and wait for user input if in interactive mode. When the fileset is neither a kernel fileset nor a prerequisite for a kernel fileset, installation of the files will proceed even if a preinstall FAILURE occurs.
2 — WARNING A WARNING was sent to the log file. A condition was detected which might result in installation or configuration failure. The warning message should explain the condition. The installation of selected software will proceed.

Permitted Actions: Typical actions valid from within a preinstall script are:

Call the newconfig_prep function to set aside files that will be delivered under /usr/newconfig as follows:

The fileset’s preinstall script should handle all files delivered under /usr/newconfig by that fileset. It should invoke the control_utils function newconfig_prep once per newconfig file, in a loop if more than one file is involved. The function requires one argument — the absolute path of the file’s ultimate location, not including a $SW_ROOT_DIRECTORY or $SW_LOCATION prefix.

Example: newconfig_prep /etc/disktab

The function will copy the previous file from its place under /usr/newconfig to a corresponding path under /usr/old/usr/newconfig (see the “Configuration Directory and Functions” section). The file under /usr/newconfig will remain. Results are unspecified if the working file is linked to the previous file, or if either is linked to any other file. The newconfig_prep function must be called for every file that is handled by newconfig_cp.

newconfig_cp:

The newconfig_cp function is usually called from a fileset’s configure script. It must be called for every file that is handled by newconfig_prep. Before the fileset’s postinstall or configure script is executed, the previously installed file will have been copied (by the newconfig_prep function) from under /usr/newconfig to /usr/old/usr/newconfig, the new file will have been installed under /usr/newconfig, and the working (that is, currently installed) file will be unaffected. The task of the newconfig_cp function is to decide whether the newly installed file should overwrite the working file, and to do so if necessary. The
newconfig_cp function should be invoked once per newconfig file. Like the newconfig_prep function, the newconfig_cp function requires one argument: the absolute path of the file's ultimate location, not including a $SW_ROOT_DIRECTORY or $SW_LOCATION prefix.

Example: newconfig_cp /etc/disktab

The function first checks for existence of the new file, and exits with an error value if the new file is not found. A likely cause of this error is if the file is delivered directly under /usr/newconfig rather than as a pseudo-root, or if the argument is passed as a relative path rather than an absolute path.

Prohibited Actions: The list of prohibited actions given in the “Required Actions” section, also applies here.

**Postinstall**

**Purpose of Postinstall Script**

Postinstall scripts are used to prepare for a kernel build when required by the install conditions. They also can drive events that must occur before a system reboot occurs.

**Order of Postinstall script execution**

Each fileset's postinstall script is run just after the fileset's files are loaded onto the target system during the Execution Phase of the installation process. A product level postinstall script is invoked after all of that product's filesets have been installed.

**Postinstall System Commands**

The commands available to the preinstall script under all conditions are relatively few. They include the Posix.2 commands listed in the “System Commands” section, and commands that are part of the SW-DIST product.

**Postinstall Actions**

**Output:**

As mentioned in the “Output” section.

**Exit Values:**

The calling process acts upon the value returned by each postinstall script. The prescribed return values are:

0 — SUCCESS  
The script completed successfully.

1 — FAILURE  
The fileset installation experienced a fatal error. Although the files were installed onto the target system successfully, a required post-installation action failed. This could jeopardize the correct functionality of the fileset.

If a kernel fileset's postinstall script returns FAILURE, the install process will exit if in non-interactive mode and will suspend and wait for user input if in interactive mode.

When the fileset is neither a kernel fileset nor a prerequisite for a kernel fileset, installation of all other files will proceed even if a postinstall FAILURE occurs.

2 — WARNING  
Installation succeeded, but a condition exists of which the user should be notified.

**Required Actions:**  
A postinstall script is an essential part of every kernel fileset and must be delivered with each kernel fileset because kernel configuration is done from within a postinstall script.
Permitted Actions: Typical actions valid from within a postinstall script at any time are:

- Copy, move, or remove files.
- Conditionally create links to another location.
- Use `newconfig_cp` to conditionally copy files delivered to `/usr/newconfig` to a working location.
- Use `mod_systemfile` to modify `/stand/system`.
- Perform other actions that affect the successful build of a kernel.

Prohibited Actions: The list of prohibited actions listed in the “Required Actions” section are also applicable.

**Configure**

**Purpose of Configure Script**
A configure script is used to perform product or fileset installation actions that cannot be accomplished by simple unconditional file extraction from the software source media. Configure scripts are typically used to alter system specific files. The configure script is invoked either as part of the `swinstall` process, or else as the result of the user invoking the `swconfig` command without the `-u` option. The configure script is not run when the user has invoked `swcopy` or `swremove`.

**Configure Execution Environment**

Since the configure scripts are run on the target system, `$SW_ROOT_DIRECTORY` will be “/” in all cases when configure is run.

**Order of Configure Script Execution**

If one or more reboot (including kernel) filesets has been successfully installed, then the execution of the configure scripts is postponed until after the system has rebooted. If all of the selected filesets are non-reboot filesets, then the configure scripts are invoked from within the `swinstall` session as the last step of the Execution Phase. The configure scripts are executed in product order, not install order.

The order is only affected by prerequisites and kernel filesets. Within any one software product, the configure script is executed first for kernel filesets and their prerequisites and then for other non-kernel filesets. Configure scripts for a fileset’s prerequisites are run before the fileset’s own configure script to ensure all features of a prerequisite fileset are available during the configuration of the current fileset. If a software product has a configure script, that product’s configure script is called prior to the calling of any configure scripts belonging to that product’s filesets.

**Configure System Commands**

The configure script, in comparison to the preinstall and postinstall scripts, has a greater selection of commands available to it under all conditions. They include:

- The Posix.2 commands listed in the “System Commands” section.
- Commands that are part of the SW-DIST product.
- Commands that are included in any prerequisite filesets.
- Commands included in the script’s own fileset that do not need to be configured before using.
Configure Actions

Output:

As mentioned in the “Output” section.

Exit values:

The calling process acts upon the value returned by each configure script. The prescribed return values are:

0 — SUCCESS Fileset configuration succeeded. The state of the fileset is changed to be CONFIGURED.

1 — FAILURE The fileset configuration experienced a fatal error. Although the files were extracted onto the system successfully, a required configuration action failed, and one or more features of the fileset will not work. The state of the fileset remains INSTALLED.

2 — WARNING Configuration succeeded, but a condition exists of which the user should be notified. The state of the fileset is CONFIGURED.

3 — EXCLUDE Fileset configuration must be run once again to complete this configuration. The fileset state remains INSTALLED. The next invocation of the configure script (by swconfig at system reboot) will complete the configuration process.

Permitted Actions: Typical actions valid from within a configure script at any time are:

- Create special device files.
- Append to existing files such as the fileset's rc.config.d file.
- Conditionally establish symbolic links.
- Change file attributes.
- Conditionally copy, move, or link files which have been delivered to a location under a private directory (that is, one only known to the packager).

Prohibited Actions: The list of prohibited actions listed in the “Required Actions” section, also applies here.

Unconfigure

Purpose of Unconfigure Script The purpose of the unconfigure script is to undo most configuration changes that were made to the system by the corresponding configure script. A typical action in an unconfigure script would be the removal of device special files or the changing of a system configuration parameter in a file under /etc/rc.config.d.

Order of Unconfigure Execution The unconfigure script is invoked as part of the swremove process, or as the result of the user giving the -u option to the swconfig command. Execution of each fileset's unconfigure script occurs just after the user has elected to proceed with removing the selected products and filesets, and just before the selected products and filesets are actually removed. It is the first step in the commitment to remove functionality. If a software product has an unconfigure script, that product's unconfigure script is called after the calling of any unconfigure scripts belonging to that product's filesets. The unconfigure scripts for any swremove session are invoked in reverse prerequisite order. The unconfigure script for all
prerequisite filesets that have been selected for removal will be run after the unconfigure script for the current fileset. The presence of kernel and/or reboot filesets has no effect on the order of execution.

Unconfigure System Commands

The commands available to the preremove script under all conditions include:

- The Posix.2 commands listed in the “System Commands” section.
- The commands that are part of the SW-DIST product.
- Commands that are included in any prerequisite filesets.
- Commands that are included in the script's own fileset.
- Core system command.

Unconfigure Actions

Output:

As mentioned in the “Output” section.

Exit Values:

The calling process acts upon the value returned by each unconfigure script. The prescribed return values are:

0 — SUCCESS
The script completed successfully. The install state is now INSTALLED.

1 — FAILURE
The script experienced a fatal error. If called as part of an swremove process, the attempt to remove the files in the product or fileset will proceed regardless. The install state is now INSTALLED.

2 — WARNING
A nonfatal condition was detected and should be reported to the user. If called as part of a swremove process, the attempt to remove the files in the product or fileset will proceed regardless. The install state is now INSTALLED.

3 — EXCLUDE
The fileset was unselected. Unconfiguration did not take place. The install state is unchanged.

Permitted Actions:

Typical actions valid from within a unconfigure script at any time are:

- Kill processes, including daemons, owned or spawned by files in the current fileset.
- Move or remove files and directories that were created by the corresponding configure script.
- Alter a value in the system's configuration files.
- Remove client specific files such as log files.
- Use IPD_addfile() and IPD_delfile() (control_utils) when copying, moving, or removing files.

Prohibited Actions:

The list of prohibited actions listed in the “Required Actions” section, also applies here.
Preremove

Purpose of Preremove Script

The purpose of the preremove script is to perform any necessary actions not done in the unconfigure script in preparation for the removal of the fileset's files. It should undo any actions taken in a postinstall script. For kernel filesets, it should modify the /stand/system file to delete entries such as driver names and configurable parameters whose functionality is part of the fileset.

Order of Preremove Execution

The preremove scripts are called during the Execution Phase. Execution of each fileset's preremove script occurs just before that fileset's files are actually removed. If a software product has a preremove script, that product's preremove script is called prior to the calling of any of that product's fileset level preremove scripts. The preremove scripts for any swremove session are invoked in reverse prerequisite order. The preremove script for all prerequisite filesets that have been selected for removal will be run after the preremove script for the current fileset. The presence of kernel and/or reboot filesets has no effect on the order of execution.

Preremove System Commands

The commands available to the preremove script under all conditions include:

- The Posix.2 commands listed in the “System Commands” section.
- Commands that are part of the SW-DIST product. Commands that are included in any prerequisite fileset.
- Commands that are part of the script's own fileset, provided the unconfiguration has not rendered them unusable.
- Core system commands.

Preremove Actions

Output:

AS mentioned in the “Output” section.

Exit Values:

The calling process acts upon the value returned by each preremove script. The prescribed return values are:

0 — SUCCESS  The script completed successfully.
1 — FAILURE  The script experienced a fatal error. The attempt to remove the files in the product or fileset will proceed regardless.
2 — WARNING  A nonfatal condition was detected and should be reported to the user. The attempt to remove the files in the product or fileset will proceed regardless.

Permitted Actions:

Typical actions valid from within a preremove script are:

- Move or remove files and directories.
- Modify the /stand/system file.

Prohibited Actions:

The list of prohibited actions listed in the “Required Actions” section, also applies here.
Postremove

Purpose of Postremove Script

The purpose of the postremove script is to perform any necessary cleanup actions after the fileset's files have been removed. Any files which might have been created by the fileset, and which might not have been added to the IPD should be removed by the postremove script. A typical action in a postremove script would be the removal of newly emptied directories when those directories are the exclusive property of the fileset.

Order of Postremove Script Execution

The postremove script for a fileset is called in the Execution Phase immediately after that fileset's files have been removed from the system. If a software product has a postremove script, that product's postremove script is called after the calling of any postremove scripts belonging to that product's filesets. The postremove scripts for any swremove session are invoked in reverse prerequisite order. The postremove script for all prerequisite filesets that have been selected for removal will be run after the postremove script for the current fileset. The presence of kernel and/or reboot filesets has no effect on the order of execution.

Postremove System Commands

The commands available to the preremove script under all conditions include:

- The Posix.2 commands listed in the “System Commands” section.
- Commands that are part of the SW-DIST product.
- Commands that are included in any prerequisite fileset.
- Core system commands which have not been affected by the current removal.

Postremove Actions

Output:

As mentioned in the “Output” section.

Exit Values:

The calling process acts upon the value returned by each postremove script. The prescribed return values are:

0 — SUCCESS The script completed successfully.
1 — FAILURE The script experienced a fatal error. The error message is logged and swremove continues.
2 — WARNING A nonfatal condition was detected and should be reported to the user.

Permitted Actions: Typical actions valid from within a postremove script at any time are:

- Remove newly emptied directories that are owned and used exclusively by the fileset.
- Replace any files that were set aside by the installation of the fileset.
- Any other actions that will bring the system closer to the condition it was in prior to the fileset’s installation.

Prohibited Actions: The list of prohibited actions listed in the “Required Actions” section, also applies here.
Files to Package

The files needed to create a product are the Object/Source files or document files on which the PSF and control scripts act to install/uninstall/verify a product. These files can be drivers, libraries, tools or a document. The pathnames of these files, both in the package and their final destinations when a product gets installed on a host, have to be specified in the PSF.
Creating a Software Depot

There are various steps that are involved in creating an SD package:

- “Step 1: Design an SD Structure”
- “Step 2: Selecting the Product Directory Structure”
- “Step 3: Writing a PSF”
- “Step 4: Writing Control Scripts”
- “Step 5: Packaging the Components”
- “Step 6: Registering the Depot”

Step 1: Design an SD Structure

As explained in the “Software Depot Package Components” section, packaging can be done at SD-Product level or at Bundle level. It is up to the user to select what level best suits his/her package. If the user intends to have more than one product as part of a package then he/she can choose Bundle level packaging. If not SD-Bundle can be omitted and packaging can also be done at SD-Product level. Depending upon the usage (Product contains several filesets, subproducts can be used to group logically related filesets.) of the SD depot SD-Subproduct objects should be used. The user also has to decide whether the packaging calls out for any specific control scripts.

Step 2: Selecting the Product Directory Structure

Once the SD package structure is decided, product directory structure (location) has to be selected. This is the directory structure where all the source/object/bin/docs files reside once installed. This has to be decided before writing the PSF as directory information is required as one of the attributes in SD-Product and SD-Fileset category of the PSF.

Step 3: Writing a PSF

After the SD structure and product directory structures are decided, it is time to write a PSF. As mentioned earlier a PSF is a master file which has the configuration attributes of the package. These attributes are to be filled in, they take effect accordingly once the product is installed.

An overview and all the attribute policies are explained in detail under the “Product Specification File” section, and the “SD Objects Attributes Classification and Flow” section.

Based on the selection of structure refer to (step 1) a PSF has to be written. To omit the bundle and consider product level packaging the bundle attribute for such a package must not be included. When considering Bundle level packaging both bundle and product attributes need to be included. In both cases Fileset attributes are needed. The Subproducts’ attributes consideration is optional.

The attributes mentioned in the “SD Objects Attributes Classification and Flow” section are as required (check the status) and should be filled in, as mentioned in the policies.

NOTE

Policies are the recommendations suggested. If not followed accordingly the PSF will generate errors during package creation.
Writing a PSF Vendor Object

In the PSF, the vendor information of the package needs to be filled in first. The information filled in should be unique. Information such as name of the products’ company.

Under this object fill in all attributes:

- **tag**: The vendor’s trademark.
- **title**: Vendor’s trademark expansion or company name.
- **description**: Description of the vendor.

Based on the vendor attribute policies discussed under the “Vendor Attributes” section, a sample template of vendor object of a PSF.

```perl
vendor
  tag <.....>
  title <.....>
  description <.....>
end
```

**Example:**

```perl
#vendor information (considered an comment as in case "/*" for 'C " code)
vendor
  tag HP
  title Hewlett-Packard
  description A Technology company
end #vendor
```

Writing a PSF Category Object

The category attributes are defined outside the SD-Bundle, SD-Product or SD-Fileset. The category_tag attribute within the SD-Bundle, SD-Product and/or SD-Fileset definition is used to refer to the associated category object. Category class attribute definitions are “global”, a particular category tag and description applies to all referring objects in the depot/on the media.

The list of attributes and the policies of filling in these attributes are mentioned under the “Category Attributes” section:

- **tag**: Identifier of the category object.
- **title**: Detailed name for the category.
- **Description**: A text string describing the category.

A Category object template:

```perl
category
  tag <.....>
  title <.....>
  description <.....>
end
```

**Example:**

```perl
#category information
Category
  tag TrialUseApps
  title Trial Use Software Applications
  description "Trial Use Software Applications"
end #category
```
Writing a PSF Bundle Object

The package Bundle information follows the vendor information. All products and filesets that are considered under the bundle are mentioned in this object.

The list of attributes and the policies of filling in these attributes are also mentioned under the “SD-Bundle Attributes” section:

- **tag**: Identifier of the bundle.
- **title**: Detailed name of the bundle.
- **Description**: A text string describing the bundle.
- **Revision**: Revision number of the bundle.
- **Architecture**: Target systems on which the bundle will be installed.
- **os_name**: The OS's on which the bundle will run.
- **os_release**: OS releases on which the bundle will run.
- **os_version**: (optional) OS versions on which the bundle will run.
- **vendor_tag**: Short name of the vendor (refer to the “Writing a PSF Vendor Object” section).
- **contents**: Software contained in the bundle.

A Bundle object template:

```plaintext
bundle
tag <.....>
title <.....>
description <.....>
revision <.....>
vendor_tag <.....>
architecture <.....>
os_name <.....>
os_release <.....>
os_version <.....>
contents <.....>
end
```

Example:

```plaintext
#bundle information

bundle
tag  B11_11QLISP
        title  "11.11 QLISP sample driver"
        description  "11.11 QLISP sample driver version 1.0"
        revision  11_11.1.0
        vendor_tag HP
        architecture HP-UX_B.11.11_32/64
        os_name HP-UX
        os_release ?.11.11
        os_version *
        contents  QLISP11_11_1_0.qlisp1_0src,r=11_11.1.0,
a=HPUX_B.11.11_32/64,v=HP
end #bundle
```
Writing a PSF Product Object

The Product information should follow the bundle information. If we consider product level packaging then bundle information should be omitted and product object should follow the vendor one. All filesets of the products should be included under this object.

**NOTE**  This document considers a fileset as a separate object.

The following is a list of all the product object attributes:

- **tag**: Identifier or name of the product.
- **title**: Detailed name of the product.
- **description**: A text string describing the product.
- **revision**: Revision (release) number of the product.
- **Architecture**: target systems on which product is installed.
- **os_name**: OS's on which Product will be installed.
- **os_release**: OS release on which product is installed.
- **os_version**: (optional), OS versions on which product will be handled.
- **directory**: The default, absolute pathname of the directory under which product will be installed.
- **is_locatable**: Product alternate directory.
- **vendor_tag**: Short name of the vendor (refer to the “Writing a PSF Vendor Object” section).
- **preinstall**: (optional) Path name of install pre-load product control script (holds same for all other control scripts).

Refer to the policies mentioned in the “SD-Product Attributes” section.

**NOTE**  The attributes revision, architecture, os_name, os_release, os_version are the same as in the bundle object if we consider bundle level packaging.

A product object template:

```
product
tag <......>
title <......>
revision <......>
vendor_tag <......>
directory <......>
preinstall <......>
is_locatable <......>
architecture <......>
os_name <......>
os_release <......>
os_version <......>
```

Example:

```
#product information
product
tag QLISP11_11_1_0
title "QLISP Sample drivers 1.0.0 for 11.11 DDK 1.0 "
revision 11_11.1.0
vendor_tag HP
```
NOTE The “end” of the product should be written after filesets.

Writing a PSF Subproduct Object

Subproduct object is optional. Subproducts information should be filled in here, following product information.

The following is a list of attributes under subproduct object:

- **tag**: The identifier for the Subproduct
- **title**: More detailed name of the Subproduct
- **description**: Text string describing the Subproduct

The policies are described in the “SD-Subproduct Attributes” section.

Since Subproduct object is optional, so are all its attributes.

A subproduct object template:

```xml
subproduct
tag <......>
title <......>
description <......>
contents
end
```

Example:

```xml
#Subproduct information
subproduct
tag QLISP11_11_1_0
title "QLISP Sample drivers 1.0.0 for 11.11 DDK 1.0"
description "This subproduct contains a collection of all sources of QLISP 1.0"
contents QLISP11_11_1_0,qlispdisp1_0,r=11_11.1.0,
a=HPUX_B.11.11_32/64,v=HP
end #subproduct
```

Writing a PSF Fileset Object

Filesets object should follow product object if we do not consider subproducts. Since filesets are the atomic units, all the information of files that are to be packaged should be filled in here.

A list of all the attributes under Fileset object:

- **tag**: The identifier for the Fileset
- **title**: Detailed name of the Fileset
- **Description**: A text string describing the Fileset
- **revision**: Revision number of the Fileset
- **Architecture**: Target systems on which Fileset is installed.
### Creating a Software Depot

#### Chapter 19

**os_name**  
OS’s on which Fileset will be installed

**os_release**  
OS releases on which the Fileset is to be installed.

**os_version**  
(optional) OS versions on which Fileset will be installed

**ls_kernel**  
(optional) Defined if the Fileset contains OS kernel files

**ls_reboot**  
(optional) Defined if the Fileset requires a system reboot after installation

**preinstall**  
(optional) path name of install pre-load Fileset control script (holds same for all other control scripts)

**file_permissions**  
Explicitly specifies default permissions for the files being packaged into the Fileset

**directory**  
Specifies a source directory in which subsequently listed filenames are located (for this Fileset only)

**file**  
Specifies the files to be packaged into a Fileset

The policies are mentioned under the "SD-Fileset Attributes" section.

---

**NOTE**  
The revision, architecture, os_name, os_release, and os_version attributes are the same as in product object.

---

A Fileset object template:

```plaintext
fileset
tag <......>
description <......>
title <......>
revision <......>
arquitecture <......>
os_name <......>
os_release <......>
os_version <......>
file_permissions <......>
directory <......>
file <......>
end #fileset
end #product (if this is the last fileset under product)
```

Example:

```plaintext
fileset
tag qlispsrc1.0
description "DDK11.11.1.0 MS HBA sample drivers"
title QLISP 1.0 for DDK11_11
revision 11.11.1.0
architecture HP-UX_B.11.11.32/64
os_name HP-UX
os_release ?.11.11
os_version *
file_permissions -m 0555
directory ../../../ddkpack/sampldrvs/ms/scsihba/src=/opt/ddk/11.11/
sampldrvs/massstorage/qlisp/11.11.1.0/src/
file *
end #fileset
end #product (if this is the last fileset under product)
```
Step 4: Writing Control Scripts

This is (optional) and this section presents in detail the usage, templates and samples of all the SD control scripts.

Writing a Checkinstall Script

The purpose of a checkinstall script is to ensure that the target system has no product or fileset specific conditions which would cause either an installation failure or a runtime failure. This specifically excludes conditions that are tested elsewhere in the swinstall process such as inadequate disk space, unmounted volumes, unresolved fileset dependencies, inappropriate architecture, and so forth.

The details such as order of script execution, commands available for checkinstall script, its actions, etc. are explained in detail in the “Checkinstall” section.

The actions of a checkinstall script must be extremely unobtrusive since there is no commitment to installing at the time the scripts are run. The checkinstall script, like all control scripts, must be re-executable. It is possible for a checkinstall script to be executed numerous times within a single install session.

The checkinstall script should be used to test:

- Selected software.
- The system’s hardware configuration.
- Kernel configuration.
- The init state of the system.
- The system’s I/O structure.
- Software already installed.
- Revision, or other attributes of installed software.

Checkinstall Script template:

```
#!/sbin/sh

########
# Product: DEMO
# Fileset: LATEST
# checkinstall
# @(#) $Revision: 11.11 $

########

(c) Copyright 2003, Hewlett-Packard Company
```

Sample Checkinstall Script:

```
1.
```
###
UTILS="/usr/lbin/sw/control_utils"
if [[ ! -f $UTILS ]]
then
   msg ERROR "Cannot find $UTILS"
   exit 1
fi
. $UTILS
exitval=$SUCCESS

PROD_VER=1            # Current version of the product
PROD_LIB=/usr/conf/lib/libenet.a
WHAT_BIN=/usr/bin/what

# Check if we are trying to install over a more recent version. If so, skip this
# installation. B.11.11 is hard-coded as the release

if [[ -x $WHAT_BIN && -e $PROD_LIB ]]; then
   what_str=
   ''$WHAT_BIN $PROD_LIB"
   print $what_str | /usr/bin/grep -q 'B.11.11'
   if [[ $? = 0 ]]; then
      INST_VER=
      ''print $what_str | /usr/bin/awk '{print
         substr($0,index($0,"B.11.11.")+8,2);}''
      if [[ $INST_VER -gt $PROD_VER ]]; then
         echo "ERROR: A newer version of the PCI FDDI has already been installed"
         echo " on this system (perhaps from a patch)."
         echo " The version in the kernel is B.11.11.$INST_VER."
         echo " The version attempted is B.11.11.$PROD_VER."
         echo " To force the installation, you must first swremove this"
         echo " product from the kernel, then retry the installation."
      fi
   fi
exit $exitval

2.

#!/sbin/sh
###
# Product: DEMO
# Fileset: LATEST
# checkinstall
# $Revision: 11.11 $
###
#
# (c) Copyright 2003, Hewlett-Packard Company
#
###
UTILS="/usr/lbin/sw/control_utils"
if [[ ! -f $UTILS ]]
then
   msg ERROR "Cannot find $UTILS"
exit 1
fi
. $UTILS
exitval=$SUCCESS #Anticipate success

# Make sure that product is not installed on an alternate root.
# Exclude the product from installation if it is alternate root.
if [[! $SW_ROOT_DIRECTORY != "/"]]
then
   echo "ERROR product cannot be installed on alternate root"
   exitval = $ERROR
fi
exit $exitval
Writing a Preinstall Control Script

The purpose of a fileset’s preinstall script is to prepare the system for installation of the fileset. By the time the preinstall script for a particular fileset is executed, the actions of that fileset’s `checkinstall` script have determined that there are no fileset-specific impediments to installing the fileset. Then the Analysis Phase has ensured that there are no known system-specific impediments to installing the current fileset. All that remains is to prepare the system and to begin installing files. The steps to prepare the system are done by the fileset’s preinstall script.

A typical preinstall script is intended to kill processes that could interfere with installation, for example daemons that keep an executable file opened. Removal of obsolete software is done in a preinstall script.

The details such as order of script execution, commands available for `checkinstall` script, its actions, and so forth are explained in detail in the “Preinstall” section.

Preinstall Script template:

```bash
#!/sbin/sh

#########
Script information and what it is used for
#########

Setup the SD environment by calling control utils


Preinstall functions like newconfig...


Sample Preinstall control script:

1.  
```bash
#!/sbin/sh

        # Do the newconfig preparation step for configuration files.
        for file in \
            /etc/rc.config.d/demo
        do
```
Writing a Postinstall Control Script

Postinstall scripts are used to prepare for a kernel build when required by the install conditions. They also can drive events that must occur before a system reboot occurs.

The details such as order of script execution, commands available for checkinstall script, its actions, etc. are explained in detail under the "Postinstall" section. A postinstall script is an essential part of every kernel fileset and must be delivered with each kernel fileset because kernel configuration is done from within a postinstall script.

A postinstall script can be used to:

- Copy, move, or remove files.
- Conditionally create links to another location.
- Use newconfig_cp to conditionally copy files delivered to /usr/newconfig to a working location.
- Use mod_systemfile to modify /stand/system.
- Perform other actions that affect the successful build of a kernel.
Template Postinstall script:

```bash
#!/sbin/sh
############
# Product: DEMO
# Fileset: LATEST
# postinstall
# $Revision: 11.11 $
#
############
# (c) Copyright 2003, Hewlett-Packard Company
#
UTILS="/usr/lbin/sw/control_utils"
if [[ ! -f $UTILS ]]
then
  echo "ERROR: Cannot find $UTILS"
  exit 1
fi
$UTILS
exitval=$SUCCESS
# here is an example that adds the static driver "mydriver"
# to the target system file.
mod_systemfile ${SW_SYSTEM_FILE_PATH} -a mydriver
retval=$?
if [[ $retval -ne $SUCCESS ]]
then
  msg ERROR "Could not enter 'mydriver' in the 
  ${SW_SYSTEM_FILE_PATH}"
  [[ $exitval -ne $FAILURE ]] && exitval=$retval
fi
exit $exitval
```

Writing a Configure Control Script

A configure script is used to perform product or fileset installation actions that cannot be accomplished by simple unconditional file extraction from the software source media. Configure scripts are typically used to alter system specific files. The configure script is invoked either as part of the swinstall process, or as the result of the user invoking the swconfig command without the -u option. The configure script is not run when the user has invoked swcopy or swremove.

A configure control script can be used to:

- Create special device files.
- Append to existing files such as the fileset's rc.config.d file.
- Conditionally establish symbolic links.
- Change file attributes.
Creating a Software Depot

Conditionally copy, move, or link files which have been delivered to a location under a private directory (that is, one only known to the packager).

Configure script template:

```
!/sbin/sh
#########
# Script information and what it is used for
#########
Setup the SD environment by calling control utils

Configure script action, for example, configure nettl

Sample Configure script:
```

```bash
#!/sbin/sh
#
# Product: DEMO
# Fileset: LATEST
#
# (c) Copyright 2002, Hewlett-Packard Company
#
 UTILS="/usr/lbin/sw/control_utils"
if [[ ! -f $UTILS ]]
then
    echo "ERROR: Cannot find $UTILS"
    exit 1
fi
.
exitval=$SUCCESS #Anticipate Success

#Configure nettl ( configure network tracing and logging command subsystem database)
nettlconf -S -id 179 -name enet -class 12 -kernel ..............

exit $exitval
```

Writing an Unconfigure Control Script

The purpose of the unconfigure script is to undo most configuration changes that were made to the system by the corresponding configure script. A typical action in an unconfigure script would be the removal of device special files or the changing of a system configuration parameter in a file under /etc/rc.config.d.

An unconfigure script can be used to:

- Kill processes, including daemons, owned or spawned by files in the current fileset.
- Move or remove files and directories that were created by the corresponding configure script.
- Alter a value in the system’s configuration files.
- Remove client specific files such as log files.
Unconfigure script template:

```bash
#!/sbin/sh

#########
Script information and what it is used for

#########
Setup the SD environment by calling control utils

#########

Configure script action, for example, remove nettl entry

Sample Unconfigure script:

```bash
#!/sbin/sh

#########
# Product: DEMO
# Fileset: LATEST
# configure
# @(#) $Revision: 11.11 $
#########

#
# (c) Copyright 2003, Hewlett-Packard Company
#

UTILS="/usr/lbin/sw/control_utils"

if [[ ! -f $UTILS ]]
  then
    echo "ERROR: Cannot find $UTILS"
exit 1
fi

. $UTILS

exitval=$SUCCESS # Anticipate success

# Remove nettl entry.
nett1conf -delete 179

exit $exitval
```

Writing a Preremove Control Script

The purpose of the preremove script is to perform any necessary actions not done in the unconfigure script in preparation for the removal of the fileset's files. It should undo any actions taken in a postinstall script. For kernel filesets, it should modify the `/stand/system` file to delete entries such as driver names and configurable parameters whose functionality is part of the fileset.

A preremove script can be used to:

- Move or remove files and directories.
- Modify the `/stand/system` file.

Preremove script template:

```bash
#!/sbin/sh

#########
Script information and what it is used for

#########
Setup the SD environment by calling control utils

#########

preremove script action, for example, remove system files for drivers
```
Sample Preremove Script:

```bash
#!/sbin/sh

########
# Product: DEMO
# Fileset: LATEST
# preremove
# @(#) $Revision: 11.11 $
########
#
# (c) Copyright Hewlett-Packard Company 2003
#
# Utilitys=/usr/lbin/sw/control_utils
if [ -f $UTILS ]
then
    . $UTILS
else
    echo "ERROR: Cannot find $UTILS"
    exit 1
fi
exitval=$SUCCESS

#############################################################
###########
# FUNCTIONS
########
# DeleteDriverEntry()
# Purpose: To delete from the system file all the drivers that this fileset
# has delivered. Set the $exitval value based on the return value from
# mod_systemfile(). If multiple deletions, ensure that a FAILURE value
# of $exitval is not reduced by a later SUCCESS return.
#
DeleteDriverEntry()
{
    typeset retval

    for driver_name in 
        driver_1 
        driver_2 
        last_driver
    do
        mod_systemfile ${SW_SYSTEM_FILE_PATH} -d $driver_name
        retval=$?
        if [[ $retval -ne $SUCCESS ]] then
            [[ $retval -ne $SUCCESS && $exitval -ne $FAILURE ]] &&
            exitval=$retval
        fi
    done

} # DeleteDriverEntry()

exit $exitval
```
Writing a Postremove Control Script

The purpose of the postremove script is to perform any necessary cleanup actions after the fileset’s files have been removed. Any files which might have been created by the fileset, and which might not have been added to the IPD should be removed by the postremove script. A typical action in a postremove script would be the removal of newly emptied directories when those directories are the exclusive property of the fileset.

A postremove script can be used to:

- Remove newly emptied directories that are owned and used exclusively by the fileset. Replace any files that were set aside by the installation of the fileset.
- Any other actions that will bring the system closer to the condition it was in prior to the fileset’s installation.

Postremove script template:

```bash
#!/sbin/sh
#
# Product: DEMO
# Fileset: LATEST
# postremove
# @(#) $Revision: 11.11 $
#
# (c) Copyright Hewlett-Packard Company 2003
#
UTILS="/usr/lbin/sw/control_utils"
if [[ ! -f $UTILS ]]
then
  echo "ERROR: Cannot find $UTILS"
  exit 1
fi
. $UTILS exitval=$SUCCESS

for file in installd ninstall
do
  rm -f /usr/share/man/cat1m.Z/${file}.1m
done

for file in /opt/demo/app/log* /dev/demo
do
  rm -f $file >/dev/null 2>&1
```

Sample Postremove Script:
done

**********
Remove empty directories that this fileset owned exclusively.
for dir in \
   /opt/demo/app \
   /opt/demo/config \
   /opt/demo/data \
   /opt/demo
do
   rm -rf $dir >/dev/null 2>&1
done

exit $exitval

**Step 5: Packaging the Components**

Once PSF and control scripts are written, its time to package the components. SD provides the `swpackage` command to package these components. This section focuses on some of the basic usage of `swpackage` command. Details of which can be found as part of man page on `swpackage`.

---

**NOTE**

All SD commands are provided as manpages on the host as part of OS.

---

**SD Components Packaging**

As explained in the “Creating a Software Depot” section, a package consists of; a PSF file, the control scripts and the source/object files of product that needs to be packaged for distribution.

The command used to create a package into a target depot:

**Syntax**

`swpackage -s *.psf`

**Inputs**

PSF file, depot directory

**Options**

- `[-s]` — PSF file
- `[-d]` — depot directory
- `[-p]` — Previews the package without creating a depot

**Usage1**

`swpackage -s “PSF file path”`

**Usage2**

`swpackage -s “PSF file path” -d “depot directory path”`

**Syntax:**

`swpackage -s *.psf`

**Usage2:**

`swpackage -s “PSF file path” -d “depot directory path”`

For example, if the PSF file of the package is in directory `/home/qlisp/` and depot directory is `/home/qlisp/driversource` then:

```
#swpackage -s /home/qlisp/qlispdrv.psf
```

- or -

```
#swpackage -s /home/qlisp/qlispdrv.psf -d /home/qlisp/driversource
```

By default the depot directory is `/var/spool/sw`. If the user does not specify the depot directory as in **usage1**, then `swpackage` puts the depot into `/var/spool/sw`; or if the user specifies as in **usage2**, then `swpackage` puts the depot into user specified directory.
swpackage Process

Phase 1  Selection

When running swpackage, specify a PSF and any other options to include. The swpackage command begins the session by telling what source, target, software selections, and options used:

- Determine the product, subproduct, and fileset required for the structure.
- Determine which files are contained in each fileset.
- Determine the attributes associated with each objects.
- Check PSF syntax and terminates the session if any are encountered.

Phase 2  Analysis

The swpackage performs these checks during this phase:

1. Check for unresolved dependencies

   For every fileset in each selected product, swpackage checks to see if a requisite of the fileset is not also selected or not already present in the target depot. Unresolved dependencies within the product generate errors. Unresolved dependencies across products produce notes.

2. Check for software being repackaged

   For each selected product, swpackage checks to see if the product already exists in the target depot.
   - If it does exist, swpackage checks to see which filesets are being added (new filesets) or modified.
   - If it exists and all filesets are selected, swpackage checks to see if any existing filesets have been obsoleted by the new product.

3. Performing Disk Space Analysis (DSA)

   The swpackage verifies that the target depot has enough free disk space to package the selected products.
   - If adequate disk space is available for the packaging operation to proceed, swpackage writes a note to the log file to note the impact on disk space.
   - An error results if the package will encroach into the disk’s minfree space.
   - An error results if the package phase requires more disk space than is available.

4. Build

   When packaging a product, if the target depot does not exist, swpackage creates it. If it does exist, swpackage will merge new product(s) into it. Before a new depot directory is created, swpackage checks to see if this product version has the same identifying attributes as an existing product version. If all the identifying attributes match, you are re-packaging (modifying) an existing version. Otherwise, swpackage creates a new version in the target distribution.

   Each product is packaged in its entirety and when all specified products have been packaged successfully, the distribution’s global INDEX file is built/rebuilt.
   - It checks if the product is new or already exists. If it is new, create the product’s storage directory.
- For each fileset in the product, copy the fileset’s files into their storage location (within the product’s storage directory), and create the fileset’s catalog (database information) files.
- After the individual filesets, create the product’s informational files (meta-files).

**NOTE**
The `swpackage` does not register the depot created. It asks you to register. This is notified at the end of the `swpackage` process.

### Step 6: Registering the Depot

After the depot is created, we need to register the depot to install and manage the depot further. The command used to register the depot is `swreg`.

**Command usage:** `swreg -l depot`

**options:** `-l` specifies the object to register or unregister. Exactly one level must be specified. For registering or unregistering a depot, use “-l depot”, and for a root use “-l root”.

**For example:** `swreg -l /home/qlisp/qlispdrv`

In this case `qlispdrv` is the target depot directory.
Installing a Depot

Once a SD depot is created and registered, it is ready for distribution. If a user wants to use the distribution, the depot should be installed first on a host. The command used to install a depot is `swinstall`.

Features of `swinstall`:

- Optional GUI.
- Compatibility filtering to ensure the software will run on the installed system.
- Ability to perform kernel rebuilding or rebooting.
- Automatic use of dependencies to automatically select software on which to operate (in addition to any software specified directly).
- Ability to run control scripts as part of the installation like all install and request scripts.

Command Usage

For GUI based `swinstall`, startup the GUI:

`#/usr/bin/swinstall` to generate TUI based
- or -
`#/usr/bin/swinstall -i` to generate a GUI based

Select Source

Specify the source depot that contains the software to install. The Specify Source dialog automatically lists the local host and default depot path.

Optionally, to specify another host system, type a source host name, or:

1. Click on the Source Host Name button. The system displays a dialog that lists all host system names contained in the `defaults.hosts` file (`$HOME/.sw/defaults.hosts` or `/var/adm/sw/defaults.hosts`).
2. Choose a host name from the list.
3. Click OK. The host name appears in the appropriate box in the Specify Source dialog.

Optionally, to specify the path to the depot, type a new path, or:

1. Click on the Source Depot Path button to display a list of registered depots on the source host.
2. Highlight one of the depots.
3. Click OK to make it appear in the Specify Source dialog.

Select Software

Use the Software Selection window to select the software to be installed.
Analysis
SD-UX analyzes the software that has been selected. The Analysis window displays status information about the analysis process. When the analysis is complete and the host status shows Ready, click OK to start the actual installation.

Installation
In this step, SD-UX proceeds with the actual installation.

swinstall usage, Command Line based:

```
syntax          swinstall -s "depot source" "depot"

Options
[-s source]
[-p preview]
[-x command_option=value]
[@target_selections]
```

Basic Operation
In the command:

```
swinstall -s /home/qlisp/ qlispdepot
```

The "/home/qlisp" is the depot source, where the depot resides or is downloaded. This command installs qlispdepot under the directory specified in the PSF file.

Other useful operations are, for example:

```
swinstall -p -s /home/qlisp/ qlispdepot
```

will only preview the installation, not install it.

To change the target install directory:

```
swinstall -s /home/qlisp/ qlispdepot:/home/elsewhere
```

will change the install directory specified in the PSF file to /home/elsewhere.

**NOTE**
When creating a depot using swpackage the developer should have set “is_locatable TRUE”. Only then can we change the target directory with the swinstall command.

Target Selection
For example, the command:

```
swinstall -s /home/qlisp qlispdepot:otherhost
```

will install the depot into some other host.
**command_options**

Change the behavior of the command by changing its options. These can be set with a value of “true” or “false” and some of them require respective settings as attributes in the PSF files also. Some of the useful ones are:

- **ask=true**
  
  Executes a request script, which asks for a user response. The ask option has two possible values:
  
  - true — executes the request script (if one exists for the selected software) and stores the user response in a file named response.
  
  - false — (default for `swinstall` and `swconfig`), does not execute request scripts.

- **autoreboot=false**
  
  Normally set to false, indicating that installation of software requiring a reboot is not allowed from the command line. If set to true, this option allows installation of the software and automatically reboots the local host.

- **reinstall=false**
  
  Prevents SD-UX from reinstalling (overwriting) an existing revision of a fileset. If set to true, the fileset are reinstalled.

- **allow_downdate=false**
  
  Normally set to false, so installing an older version of software than already exists is disallowed. This prevents installing older versions by mistake. Additionally, many software products do not support this “downdating”.

  If set to true, a previous version can be installed but SD-UX issues a warning message.

- **allow_multiple_versions=false**
  
  Normally set to false, so installed or configured multiple versions (for example, the same product, but a different revision, installed into a different location) are disallowed. If set to true, install and manage multiple versions of the same software.
Managing the Depot Software

Once the depot is installed, we do need to manage the software of the depot to suit our needs. SD provides the following `swlist` and `swcopy` depot managing commands.

**swlist**

The `swlist` command can display lists of registered depots residing on a host.

**Command Usage** `swlist`

**Options**

- `-l` depot @ hostA — to list all depots on remote host A.
- `-l` product Bundle — to list all products in the specific Bundle.
- `-l` fileset Bundle — to list all filesets in a specific Bundle.
- `-l` fileset Product — to list all filesets in a specific product. For example:

```
swlist -l depot
swlist -l product QLISPBUNDLE
swlist -l fileset QLISPBUNDLE
swlist -l fileset QLISPDRVSRVC
```

**swcopy**

The `swcopy` command copies or merges software selections from a software source to one or more software depot target selections. This is a GUI and command line based command. The GUI steps are same as for `swinstall commands`.

**Command Usage** `swcopy [-s] source [\@ target selections]`

**Options**

- `-s` — Source depot ([host]:[/directory) specifies the source depot from which software is installed or copied. The default source type is directory.

```
\@ target selections: ([host]:[/directory)
```

**Example:**

```
sCOPY -s /home/ddk/depots/QLISP/QLISPdrvsrc /opt/qlisp
```

Here, `/home/ddk/depots/QLISP/QLISP drvsrc` is a depot path of QLISP driver sources and `swcopy` will copy the QLISP `drvsrc` product to `/opt/qlisp` depot path on local host.

**Example:**

```
sCOPY -s /home/ddk/depots/QLISP/QLISPdrvsrc @ hostA:/opt/qlisp
```

Will copy the QLISP `drvsrc` product to `/opt/qlisp` depot path on host A.
Removing the Installed Software

Any installed software can be removed by using the `swremove` command.

**Command usage:** `swremove Bundle/product/product`.

- `#swremove Bundle` Removes all software installed from the Bundle.
- `#swremove Bundle.product` Removes all software installed from the product contained in the bundle.
- `#swremove Bundle.product.fileset` Removes all software installed from the fileset contained in a product which part of a bundle.
- `#swremove product` Removes all software in the product.
- `swremove QLISPBUNDLE` Removes all driver sources, binaries, documents from the QLISP bundle.
- `swremove QLISPBUNDLE.QLISPDRVSRC` Removes only the driver sources but not binaries and documents.
- `swremove QLISPDRVSRC` If `QLISPDRVSRC` is not part of any bundle.

---

**NOTE** When `swremove` is done, it only removes the files (software). It does not remove the directory path where the files are installed.

For example, if qlisip driver sources are installed on a system at `/opt/ddk/src/qlisp/*.[ch]`, `#swremove QLISPDRVSRC` will remove `*.ch` sources, but can still see an empty `/opt/ddk/src/qlisp` directory there.

Command Messages Logging

The SD commands' messages are logged in `/var/adm/sw/swagent.log`. 

Creating a Package

PSF

For example, qlisp.psf

```bash
vendor
  tag   HP
  title Hewlett-pakard
  description Hewlett
end

bundle
  tag   B11_11QLISP
  title "11.11 QLISP sample driver Version 1.0" with only sources"
  description "11.11 QLISP sample driver Version 1.0 with only sources"
  vendor_tag HP
  architecture HP-UX_B.11.11_32/64 ( for 32 and 64 bit systems)
  os_name HP-UX
  os_release ?.11.11
  os_version *
  contents QLISP11_11_1_0.qlispsrcfiles1_0,r=11_11.1.0,a=HP
             UX_B.11.11_32/64,v=HP
end

product
  tag   QLISP11_11_1_0
  title "QLISP 1.0 Sample drivers sources"
  revision 11_11.1.0
  vendor_tag HP
  directory /opt/ddk/11.11 ( where the QLISP sample sources product is
  installed)
  is_locatable TRUE ( while installing the user can relocate the
  above install path)
  preinstall scripts/preinstall (scripts is a directory)
  postinstall scripts/postinstall
  architecture HP-UX_B.11.11_32/64
  os_name HP-UX
  os_release ?.11.11
  os_version *
  fileset
    tag    qlispsrcfiles1_0
    description "QLISP 1.0 sources only"
    title   QLISPSOURCES
    revision 11_11.1.0
    architecture HP-UX_B.11.11_32/64
    os_name HP-UX
    os_release ?.11.11
    os_version *
    file_permissions -m 0555
    directory /samlrdvs/ms/scsihba=/opt/ddk/11.11/samlrdvs/
              massstorage/qlisp/11.11.1.0
    file *
end #fileset
end #product
```
Control Scripts Files

Preinstall
This script checks if there are any obsolete qlisp configuration files, removes them and continues the installation:

```bash
#!/sbin/sh
#
# Product: QLISP11_11_1_0
# Fileset: qlispsrcfile1_0
# preinstall
# @(#) $Revision: 11.11 $
#
#
# (c) Copyright Hewlett-Packard Company 2003
#
#
# set up the SD environment
UTILS="/usr/lbin/sw/control_utils"
if [[ ! -f $UTILS ]]
    then
        echo "ERROR: Cannot find $UTILS"
        exit 1
    fi
.
$UTILS
exitval=$SUCCESS # Anticipate success
```

Postinstall
This script installs the driver into kernel by adding qlisp into the system file:

```bash
#!/sbin/sh
#
# Product: QLISP11_11_1_0
# Fileset: qlispsrcfile1_0
# postinstall
# @(#) $Revision: 11.11 $
#
# (c) Copyright Hewlett-Packard Company 2003
#
#
# UTILS="/usr/lbin/sw/control_utils"
if [[ ! -f $UTILS ]]
    then
        echo "ERROR: Cannot find $UTILS"
    fi
.
$UTILS
exitval=$SUCCESS # Anticipate success
```
 Creating a Software Depot

Examples

```bash
# $DRV_NAME=qlisp
if [[ $SW_ROOT_DIRECTORY = '/' && -z $SW_DEFERRED_KERNBLD ]]
then
  mod_systemfile ${SW_SYSTEM_FILE_PATH} -a $DRV_NAME
  retval=$?
  if [[ $retval -ne $SUCCESS ]]
  then
    echo 'ERROR: Could not enter 'qlisp' in the
    ${SW_SYSTEM_FILE_PATH}"
    exitval=$retval
  fi
  fi
```

Commands

Create a Package

```bash
#swpackage -s qlisp.psf -d /home/user/qlispdrvdepots
```

Creates a depot under /home/user/qlispdepots.

```bash
#ls /home/user/qlispdrvdepots
```

Will have:

```
B11_11QLISP
QLISP11_11_1_0
catalog
swagent.log
```

```bash
# ls /home/user/qlispdrvdepots/QLISP11_11_1_0
```

Has qlisp sources under directory specified in the “=target dir path” across the “directory” filesystem of the PSF.

```bash
#ls /home/user/qlispdrvdepots/catalog
```

Has INDEX file with creation time and all information set in PSF.

```bash
#cd B11_11QLISP
```

Has the pfiles (product files) in turn will have INDEX and INFO files which are self explanatory.

```bash
#cd QLISP11_11_1_0
```

Has pfiles same as in for B11_11DDK.

Register the Depot

```bash
#swreg -l depot /home/user/qlisdrvdepots
```

Registers the qlisp depot.

Installing the Depot

```bash
#swinstall -s /home/user/qlispdrv/depots B11_11QLISP
```

Installs the qlisp sources under /opt/ddk/11.11/sampldrvs/massstorage/qlisp/11.11.1.0.
List the Product

#swlist
Lists B11_11QLISP with revision and its title.

#swlist -l product B11_11QLISP
Lists QLISP11_11_1_0 product.

#swlist -l fileset B11_11QLISP
Lists qlispsrcfiles1_0.

Remove the Sources from the Product

#swremove B11_11QLISP
Removes all files under /opt/ddk/11.11/sampldrvs/massstorage/qlisp/11.11.1.0/.

#swremove B11_11QLISP.QLISP11_11_1_0

NOTE If we have more than one product in a bundle and wish to remove only one product then we can use the above command. If we have only one product in an bundle, removing bundle itself will remove all the products, therefore the above command is not useful in that case.

Removes the files from the same location.

#swlist
Will not show B11_11QLISP and product and filesets under it.
A

32-Bit Program A program compiled to run in 32-bit mode. For example, programs compiled for the PA RISC 1.X processors.

64-Bit Program A program compiled to run in 64-bit mode. For example, programs compiled for the PA-RISC 2.0 processor in wide mode.

100BT 100BASE-T is the technical term for the Fast Ethernet or IEEE802.3u standard. See also, Fast Ethernet.

Adapter Card Physical hardware, under software control, which is typically attached either directly to an I/O bus or to an auxiliary bus (e.g., SCSI) attached to a directly connected adapter. A device typically combines a hardware controller with the mechanism (e.g., disk controller with disk).

Area Allocator The memory attribute based allocated in HP-UX kernel which replace the old MALLOC()//FREE() interface. The advanced features include object caching, improved fault isolation, reduced memory fragmentation and better scaling.

ARP Address Resolution Protocol.

Attach Chain A linked list of driver attach routines (<drv>_attach). As a hardware module is being configured, this list is walked to allow each driver in the system a chance to recognize and claim the hardware module.

B

BAR Base Address Register. On a PCI card, one of the registers in PCI configuration space that contains the size and alignment requirements needed to map the card's registers. Each BAR also contains information (encoded in the low-order bits of the register) indicating whether they are base registers for PCI memory space or for PCI I/O space. The system reads and decodes this information and writes a PCI address back into these registers when it initially maps them in. BARs contain PCI addresses when properly set up.

BDR Boot Data Record.

Beta Semaphores Mutually-exclusive, blocking semaphores. When a thread acquires a beta semaphore, it is released. The owning thread may subsequently block (i.e., sleep) and still keep ownership. Threads waiting to acquire an owned beta semaphore are blocked.

Big Endian A format for storage or transmission of binary data in which the most significant bit or byte comes first. See also, Little Endian.

bit mask A pattern of binary values, typically used in bitwise operations.

Bit An atomic unit of data representing either a 0 or a 1.

Bitwise Operation A bitwise operation treats its operands as a vector of bits rather than a single number.

BN-CDIO Bus Nexus CDIO, low-level kernel software that manages platform-dependent bus connection hardware.

Broadcast Address A well-known multicast address signifying the set of all stations.

Bundle A collection of filesets, possibly from several different products, “encapsulated” for a specific purpose. Bundles can consist of groups of filesets or products.

Bus Mastering The act of taking over a bus and generating cycles on it. A bus master is any piece of hardware that creates read or write cycles on the PCI bus. Typical cards become bus masters only when they perform DMA operations, although any card-initiated cycle (for example, a peer-to-peer transaction) is an example of bus mastering.

Bus Nexus Connection between two buses.

C

Cache Coherence Consistency of data in host memory as viewed by processor caches and I/O devices.

Cacheline The smallest unit of memory that can be transferred between the main memory and the cache. Typically, Cacheline is hardware dependent.
Canonical format Synonymous with Little Endian format.

ccNUMA Cache-Coherent Non-Uniform Memory Architecture. See also, NUMA.

CDB Command Descriptor Block.

CDIO Context Dependent I/O module. A module in GIO framework which contains all bus specific and/or driver environment specific functionality.

Central Bus CDIO (CB-CDIO) The BN-CDIO which is responsible for discovering and initializing CEC components.

CKO Checksum Offload.

Class A logical grouping of device or hardware modules by type. For instance the class “tape” would include all tape devices regardless of bus interface.

Coherent I/O Accesses to data in host memory by I/O devices are consistent with accesses by CPU caches. Hardware in the platform maintains the consistent view of data in host memory as DMA transactions flow through the hardware.

Continuous DMA A type of DMA that makes a host memory buffer continuously available to an I/O device. This type of DMA is mainly used for control structures and circular queues that are shared between the device driver and the hardware device.

Core Electronics Complex (CEC) The chipset which interfaces directly to the processor in the processor-memory interconnect. In simple systems, this usually includes memory controllers and I/O adapters. On more complex systems, it might include high-speed interconnects and coherency controllers.

CPU Central Processing Unit

CSMA/CD Carrier Sense, Multiple Access with Collision Detection.

Datagrams (1) A frame or packet transferred using connectionless communications. (2) A frame or packet sent using best-effort service. (3) An IP packet.

Decapsulation The process of removing protocol headers and trailers to extract higher-layer protocol information carried in the data payload. See also, Encapsulation.

Depot A repository of software products managed by SD/UX. A depot consists of a directory or physical media such as tapes, CD-ROMS, or DVDs.

Device Driver The software used to provide an abstraction of the hardware details of a network or peripheral device interface. Device drivers allow higher-layer entities to use the capabilities of a device without having to know or deal with the specific implementation of the underlying hardware.

DLKM Dynamically Loadable Kernel Module.

DLPI Data Link Provider Interface.

DLSAP Data Link Service Access Point.

DMA Direct Memory Access. I/O transactions for which the device interacts directly with memory without processor intervention.

Driver Software module which controls a device, interface card or bus-nexus. See also, Device Driver

DSAP Destination Service Access Point.

Encapsulation The process of taking data provided by a higher-layer entity as the payload for a lower-layer entity and applying a header an trailer as appropriate for the protocol in question. See also, Decapsulation.

ENET HP-UX sample Native STREAMS DLPI network interface driver.

Ethernet The popular name for a family of LAN technologies standardized by IEEE 802.3.
F

Fast Ethernet  An Ethernet system operating at 100 Mb/s.

Filesets  Include all the files and scripts that make up a product. They can only be part of a single product. They are the lowest level object managed by SD.

Fragmentation  A technique whereby a packet is subdivided into small packets so that they can be sent through a network with a smaller MTU. See also, Reassembly.

Frame  The Data Link layer encapsulation of transmitted or received information.

Frame Check Sequence  A block check code used to detect errors in a frame. Most LANs use a CRC-32 polynomial as their FCS.

Full Duplex  A mode of communication whereby a device can simultaneously transmit and receive data across a communications channel. See also, Half-duplex.

G

Gigabit Ethernet  An Ethernet system operating at 1000 Mb/s.

GIO  General I/O System.

Group Address  Synonymous with multicast address.

H

Half duplex  A mode of communication in which a device can either transmit or receive data across a communications channel, but not both simultaneously. See also, Full duplex.

HBA  Host Bus Adapter.

Header  A protocol-specific field or fields that precede the encapsulated higher-layer data payload (e.g., the MAC addresses in a Data Link frame). See also, Trailer.

High Availability (HA)  Used to describe a computer system that has been designed to allow users to continue with specific applications even though there has been a hardware or software failure.

HP-DLPIs  HP’s own implementation of the DLPI layer.

I

ICMP  Internet Control Message Protocol.

IELAN  HP-UX sample non-native HP-DLPI based network interface driver.

IHV  Independent Hardware Vendor.

ILP32  C language data model where int, long and pointer data types are 32 bits in size.

Init Function  This is an attribute in driver modules modmeta file. An “initfunc” statement specified an initialization function, provided y the module, that the system should call during driver initialization.

Init List  A linked list of device driver init routines (<drv>_init) which is built as the drivers configure themselves and run as the I/O system configuration is completed to perform any device driver-specific initialization.

Installed Product Database (IPD)  SD uses the Installed Product Database (IPD) to keep track of what software is installed on a system. The IPD is a series of files and subdirectories that contain information about all the products that are installed under the root directory (/). For depots, this information is maintained in catalog files beneath the depot directory. The SD commands automatically add to, change and delete IPD and catalog information as the commands are executed.

Interface Select Code (ISC)  Each instance of an adapter card has an ISC entry that the system maintains in an internal table. Each ISC entry is used by WSIO to maintain interface device driver information.

Interface Service Routine (ISR)  A function that handles interrupts that are received for a specific device driver. A pointer to this routine is linked to a system vector table. When an interrupt occurs, it is...
ruled to the ISR that is placed in the section of the Interrupt Vector Table that corresponds to the received interrupt.

**Interrupt Service Routine** A function that handles interrupts that are received for a specific device driver. A pointer to this routine is linked to a system vector table. When an interrupt occurs, it is routed to the ISR that is placed in the section of the Interrupt Vector Table that corresponds to the received interrupt.

**Instance** A number assigned to an I/O tree node. The number is unique within a driver class.

**I/O Adapter** Hardware to provide IOVA translation between an I/O bus and the processor/memory interconnect devices on the I/O bus issue bus transactions to IOVA memory targets and the I/O adapter translates IOVA memory targets to physical addresses. The I/O adapter also participates in the coherency protocol of the processor caches for platforms that are coherent or semicoherent.

**I/O Bus** Interconnect bus for I/O cards and devices. PCI is an example of an I/O bus.

**I/O Node** An element of an I/O tree which includes all relevant information needed for configuring a single hardware module.

**I/O PDIR** I/O Page Directory. Address translation table associated with an I/O adapter. The I/O PDIR is analogous to the PDIR used by CPUs for virtual-to-physical address translations. It is a table maintained by the kernel to provide mappings between IOVAs and physical addresses.

**I/O Tree** Data structure for recording the I/O subsystem configuration information.

**IOVA** I/O Virtual Address. Address used by I/O devices to access host memory. Platforms that are semicoherent or coherent, or where the processor/memory interconnect is greater than 32-bits wide, generally implement IOVAs.

**IPF** Itanium Processor Family.

**IRQ** Interrupt Request.

**ISC** Interface Select Code. Usually used as a pointer to an element of a table of *isc_table_type* structures (one per interface card). Each ISC entry is used by WSIO to maintain interface device driver information.

**ISCSI** SCSI over IP.

**ISR** Interrupt Service Routine. A driver-specific routine which handles interrupts from the device.

**ISV** Independent Software Vendor

**J**

**Jumbo Frame** A frame longer than the maximum frame length allowed by a standard. Specifically used to describe the 9-Kbyte frames on Ethernet LANs.

**K**

**Kcweb** The HP-UX Kernel Configuration tool user interface uses as a web browser.

**L**

**LAN** Local Area Network. A network with a relatively small geographical extent.

**LBI** Line Based Interrupt.

**Length Encapsulation** The Ethernet frame format where the length/Type field contains the length of the encapsulated data rather than a protocol type identifier. Length Encapsulated frames typically use LLC to multiplex among multiple higher-layer protocol clients. See also, Type Encapsulation.

**Little Endian** A format for storage or transmission of binary data in which the least significant bit or byte comes first. See also, Big Endian.

**LLC** Line Based Interrupt.

**Logical Address** See Multicast Address.

**LP64** C language data model where the `int` data type is 32 bits wide, but long and pointer data types are 64 bits wide.

**LSB** Least significant bit or least significant bit.
LUN Logical Unit Number.

LVM The Logical Volume Manager is a disk management subsystem that offers access to file systems as well as features such as disk mirroring, disk spanning, and dynamic partitioning.

MAC Medium Access Control.

MAC Address A bit string that uniquely identifies one or more devices or interfaces as the sources or destination of transmitted frames. IEEE 802 MAC addresses are 48 bits in length and may be either unicast (source or destination) or multicast (destination only).

MAC Algorithm The set of procedures used by the stations on a LAN to arbitrate for access to the shared communication channel (e.g., CSMA/CD, Token Passing).

Map PCI Device/Function The act of mapping a PCI device or function involves determining the size and alignment requirements for each memory or I/O range described by an implemented configuration-space base register. Using these requirements, PCI Services finds a suitable hole in the memory or I/O address space and updates the corresponding base register to point to this range. This is taken care of by the system (firmware and/or the kernel) at the time of the card's initialization.

Map PCI to Port Handle Mapping a PCI I/O space address to a port handle is the act which allows a driver to access the I/O space using pci_read_port_uintNN_isc() and pci_write_port_uintNN_isc(), passing in the port handle as a argument. The mapping is done through a call to pci_get_port_handle_isc().

Map PCI to Virtual Address Mapping a PCI memory space address to a virtual address is the act that allows a driver to access PCI space using READ_REG_UINTNN_ISC() or WRITE_REG_UINTNN_ISC() with that virtual address. The mapping is done through a call to map_mem_to_host().

mbuf Message buffer. Data structure used in mass storage stack.

Metadata The metadata for a module are used by the kernel configuration tools when configuring a module, they are also used by various kernel services while the module is in use.

Memory Mapped I/O (MMIO) I/O that occurs by mapping the device's I/O to system memory.

MP Multi-Processor

MP Safe Describes a module which is protected in an MP environment through the use of various spinlocks and semaphores. Note that MP-safeness does not imply any performance considerations due to the granularity of the semaphores (e.g., use of a single I/O Empire semaphore or separate semaphores for each instance all imply MP-safeness).

MP Scalable Describes an MP module which may add components without causing more drain on other MP modules. An MP-scalable driver will provide a separate spinlock for each instance of the driver. Non MP-scalable drivers may still be MP-safe but perhaps only provide a single semaphore and spinlock for all instances of the driver. Adding more instances of a non MP-scalable driver will therefore cause additional taxing of those resources for each instance added to the system.

Modwrapper The additional code and data structures added to a DLKM module to make it dynamic.

MSB Most significant bit, or most significant byte.

MTU Maximum Transmission Unit.

MTU Discovery A process whereby a station can determine the largest frame or packet that can be transferred across a internetwork without requiring fragmentation.

Multicast Address A method of identifying a set of one or more stations as the destination for transmitted data. Also known as logical address or group address.

N

NFS Network File System.
NIC Network Interface Card or Network Interface Controller.

Noncoherent I/O Accesses to data in host memory by I/O devices are not made consistent with processor caches by hardware. Software must explicitly flush the processor caches prior to starting a DMA transaction by an I/O device; and, in the case of data read from an I/O device, purge the processor caches after the DMA transaction completes.

NUMA Non-Uniform Memory Architecture. A memory architecture, used in multiprocessors, where the access time depends on the memory location. A processor can access its own local memory faster than non-local memory (memory which is local to another processor or shared between processors). See also, ccNUMA.

O

On-Line Addition, Replacement and Delete (OLA/R/D) The ability to insert adapter cards and replace such cards while a system is in use (Hot Plug).

Operating System The low-level software responsible for managing the underlying hardware in a computer, scheduling tasks, allocating storage.

OSI Open Systems Interconnect.

P

PA Precision Architecture.

Package A collection of files that need to be distributed. It is created by a SD command.

Packet The Network layer encapsulation of transmitted or received information.

Packet DMA A type of DMA that maps a host memory buffer temporarily. This is used when pre-existing memory objects must be mapped for DMA, or when a mapping only needs to be temporary.

PCI Peripheral Component Interconnect. An industry standard bus used mainly by current generations of HP platforms as a means of providing expansion I/O.

PCI Address An address in the PCI memory or I/O space. This is the type of address found in a PCI memory or I/O base address register. It is NOT a virtual address or an I/O port handle, which a driver could use to access a card.

PCI Card A PCI bus can have up to 32 devices; each device can have up to eight functions. A PCI card can have single or multiple devices; each device can have single or multiple functions. For example, a four-port LAN card is a multi-device PCI card, but none of these devices is multi-functional. On the other hand, a dual-port SCSI card is a single device, but it has two functions.

PCI Configuration Space This always-accessible space allows a driver to configure and obtain status from PCI devices or functions.

PCI I/O Space The space that is addressed by an I/O cycle on the PCI bus. This is a less often used way to access card registers on cards who choose to respond to PCI I/O accesses. Most cards have registers that are in PCI memory space instead of I/O space (i.e., they respond to PCI memory cycles, not PCI I/O cycles).

PCI Memory Space The space that is addressed by a memory cycle on the PCI bus. It is called memory space to indicate that it is memory-mapped input/output, as opposed to a special I/O style of input/output. The current PA Workstation I/O architecture allows the PA processor to directly access PCI memory space (i.e., a single instruction). Typical cards map their registers into PCI memory space, meaning they can only be accessed by PCI memory cycles.

Physical Address Real address by which host memory or an I/O device register is accessed.

Physical Layer The lowest layer of the seven-layer OSI model, responsible for transmission and reception of signals across the communication medium.

Ping A utility program used to test for network connectivity by using the Echo Request and Echo Response mechanisms of ICMP.
**Port Handle** The kernel resource associated with a mapped range of PCI I/O space. This handle is used to access the I/O space addresses by calling `pci_read_port_uintNN_isc()` and `pci_write_port_uintNN_isc()`.

**Port I/O (PIO)** Communication with an I/O device using the device's ports.

**PPA** Physical Point of Attachment

**Product** Collections of filesets and (optionally) subproducts and control scripts.

**Product Specification File (PSF)** A master file where all bundle configuration information (attributes) exists.

**Promiscuous Mode** A mode of operation of a network interface in which it receives (or attempts to receive) all traffic regardless of Destination Address.

**Protocol** A set of behavioral algorithms, message formats, and message semantics used to support communications between entities across a network.

**pSCSI** Parallel SCSI. See also, SCSI.

**Q**

**QLISP** HP-UX sample parallel

**R**

**Reassembly** The process of reconstructing a packet from its fragments. See also, Fragmentation.

**Root** In SD, a system on which depot software is installed.

**S**

**SAM** System Administration Manager. A GUI based application for HP-UX system administration.

**SAP** Service Attach Point

**SCSI** Small Computer System Interface. An industry standard external I/O bus available on all HP9000 systems.

**SDTR** Synchronous Data Transfer Request.

**Semicoherent I/O** Similar to coherent I/O. However, for the case of data read from an I/O device, software must synchronize the data that have been read into host memory after the DMA transaction completes.

**Server I/O (SIO)** I/O environment for port-server drivers with origins in S/800 systems.

**SNAP** Sub-Network Access Point

**Software Distributor (SD)** The software distributor tool for HP-UX operation system.

**Software Objects** Can be a bundle, product or filesets.

**Spinlock** Basic locking primitive used by the kernel for short-term locks. When a thread acquires a spinlock, the thread's current processor becomes the effective owner until the spinlock is released. Threads (processors) waiting to acquire an owned spinlock will spin while waiting; they do not block. For the duration that a processor owns a spinlock, external interrupts to the processor are disabled.

**Stream** A connection supported by the STREAMS facilities between a user process and a device driver. It is a structure made up of linked modules, each of which processes the transmitted information and passes it to the next module. Use STREAMS to connect to a wide variety of hardware and software configurations, using building blocks, or modules, that can be stacked together. STREAMS drivers and modules are similar in that they both must declare the same structures and provide the same interface. Only STREAMS drivers manage physical hardware and must therefore be responsible for handling interrupts if appropriate.

**T**

**Target** Either a host (the host’s file system) or a depot that resides on a host.

**TBI** Transaction Based Interrupt.

**TCP** Transmission Control Protocol.

**Token Ring** A LAN whose MAC algorithm uses token passing among stations on a logical ring topology (i.e., IEEE 802.5).
**Topology** The physical or logical layout of a network.

**Trailer** A protocol-specific field or fields that follow the encapsulated higher-layer data payload (e.g., the FCS in a Data Link Frame). See also, Header.

**Type Encapsulation** The Ethernet frame format in which the Length/Type field identifies the protocol type of the encapsulated data rather than its length. See also, Length Encapsulation.

**UDP** User Datagram Protocol.

**Virtual Address** Address used by processors, when executing in virtual mode, to access host memory. Address translation hardware converts a virtual address to a physical address before host memory is accessed. Virtual addresses may also be used to map and access I/O device registers.

**WDTR** Wide Data Transfer Request.

**WSIO** Workstation and Server I/O. A CDIO, also the HP-UX Driver Development Environment (DDE).
Symbols
$CONFIGURE, 447
$INTERFACE, 447
$LOADABLE, 447
$TUNABLE, 448
$TYPE
  misc, 447
  streams_drv, 447
  streams_mod, 447
  wsio_class, 447
  wsio_intfc, 447
$VERSION, 447

(BDR) Bus Device Reset, 326
(BOT) Beginning of Tape, 361
(BSS) Block Started by Symbol, 439
(CDB) Command Description Block, 304
(CDBs) SCSI Command Blocks, 284
(CDIO) Context Dependent Input/Output, 452
(CDIO) Context-Dependent I/O, 26, 27
(CKO) checksum onload, 266
(CORE CDIO) Core Context-Dependent I/O, 27
(DCLM) Dynamically Configured Loadable Module, 456
(DDE) Device Driver Environment, 280
(DKLM) Dynamically Loadable Kernel Module, 55
(DLKM) Dynamically Loadable Kernel Modules, 422
(DLI) Data Link Provider Interface, 183
(DLS) Data Link Service, 262
(DMA) Direct Memory Access, 281
(DSA) Disk Space Analysis, 551
(EISA CDIO) EISA Context-Dependent I/O, 27
(EOD) End of Data, 361
(EOF) End of File, 361
(FC) Fibre Channel, 279, 283
(FIFOs) First In First Out, 335
(GIO) General I/O, 56
(GIO) General I/O System, 26
(GUI) Graphical User Interface, 496
(HBA) Host Bus Adapter, 286
(HBAs) Host Bus Adapters, 279
(IA) Itanium Architecture, 219
(ICOD) Instant Capacity on Demand, 488
(ICS) Interrupt Control Stack, 57, 72, 179, 285
(IHV) Independent Hardware Vendor, 211
(INET) Internet, 163
(IOCB) I/O Control Blocks, 342
(IOVA) I/O Virtual Address, 52, 53
(IOVA) I/O Virtual Address View, 52
(IPD) Installed Product Database, 516
(IRQ) Interrupt ReQuest, 44
(ISC) Interface Select Code, 27, 40
(ISR) Interrupt Service Routine, 37, 75, 332
(ISVs) Independent Software Vendors, 422
(KDT) Kernel Device Table, 26
(KIS) Kernel ID String, 442
(KRS) Kernel-Specific, 432
(KRS) kernel-specific, 441
(LBI) Line Based interrupt, 481
(LBIs) Line Based interrupts, 480
(LLC) Link Level Control, 169, 179, 180
(LUN) Logical Unit, 312
(LUN) Logical Unit Number, 355
(LVM) Logical Volume Manager, 33
(MBLKS) Message Blocks, 266
(MP) Multiprocessor, 355
(MP) multiprocessor, 72
(MTU) Maximum Transmission Unit, 216
(mtu) maximum transmission unit, 169
(NBPG) Number of Bytes per Page, 312
(NIC) Network Interface Card, 265
(NLS) National Language Support, 247
(NLS) Native Language Support, 247
(OLA) On-Line Addition, 405, 412
(OLA/R) On-Line Addition/Replacement, 403
(OLR) On-Line Replacement, 412
(PA CDIO) Precision Architecture
  Context-Dependent I/O, 27
(PA) Precision Architecture, 219, 327
(PCI CDIO) PSI Context-Dependent I/O, 27
(PDIR) Page DiRectory, 55
(PDU) Protocol Data Units, 229
(PIO) Programmed I/O, 281
(PPA) Physical Point of Attachment, 169, 220
(pSCSI) Parallel SCSI, 286
(pSCSI) parallel SCSI, 279
(PSF) Product Specification File, 492, 496
(RPC) Remote Procedure Call, 496
(RTE) Real Time Extension, 488
(SAM) System Administration Manager, 167, 253, 412
(SCLM) Statically Configured Loadable Module, 456
(SD) Software Depot, 248, 491, 492
(SDTR) Synchronous Data Transfer Request, 288, 326
(TBI) Transaction Based interrupt, 484
(TBI) Transaction Based interrupts, 480
(TBIs) Transaction Based interrupts, 480
(UP) single processor, 355
(UP) uniprocessor, 72
(VLAN) Virtual Local Area Network, 263
(WDTR) Wide Data Transfer Request, 326
(WDTR) Wide Data Transport Request, 304
(WSIO) Workstation I/O, 27
(WSIO) Workstation Input/Output, 443
/usr/contrib/bin/intctl command, 479
__LP64__, 475

Numerics
64-bit data module, 472
64-bit safety, 471

A
Abort function, 310
Aborting an I/O Request, SCSI, 333
ACKs, 265
adding a Series 700 driver to configuration files, 152
Agent
  source, 496
  target, 496
Analysis Phase, 526
ANSI function prototypes, 477
architecture attribute
  os bits, 501
  os name, 501
  os release, 501
Arena Allocator, 59
ARP, 169
ARP configuration, 184
Index

ARP STREAMS, 169
arpcom structure, 168
attribute
  vendor.tag, 502
Autoneg, 220

B
Beginning of Tape (BOT), 361
Beta Semaphores, 73
  overview, 76
binding and unbinding, protocol, 179
Block Device Features, 34
block devices, 33
block I/O, 35
Block Started by Symbol (BSS), 439
bound SAP data fields, 197
buf structure, 40
buf Structure, Device Queue Management, 135
buffering data using uiomove(), 119
build environment, 156
bundle.category_tag, 502
bundle.category_title, 502
bundle.contents, 503
bundles, 493
Bus
  Device Reset, SCSI, 332
Lock, SCSI, 375
  Structure, SCSI, 315
Bus Device Reset, 310
Bus Device Reset (BDR), 326
Bus Reset, 310

C
C_ALLCLOSES Flag, 116
Cache Coherence, 50
Calling
  driver_attach(), SCSI, 323
  driver_install(), SCSI, 321
calling
uiomove(), 119
calling physio(), 119
category.description, 500
category.tag, 500
  HPUXAdditions, 500
  OrderedApps, 500
  OrderedHP-UX, 500
  TrialUseApps, 500
category.titles, 500
cnNUMA, 47
cdevsw, 35
CDIO
  basic components, 27
Character Device Driver Features, 34
character devices, 33
character I/O, 35
checksum
  IP header, 265
  TCP/UDP, 265
checksum offload (CKO), 266
Coherency, Driver Requirements for, 51
Coherent I/O Systems, 53
Command Discription Block (CDB), 304
command word
ioctl, 121
command, /usr/contrib/bin/intctl, 479
commands
  lanadmin, 211
  lanscan, 211
  linkloop, 211
  swaci, 496
  swagent, 496
  swagentd, 496
  swconfig, 496
  swcopy, 496
  swdepot, 496
  swinstall, 496
  swjob, 496
  swlstat, 496
  swreg, 496
  swremove, 496
  swverify, 496
commands and processing, driver control, 197
compiling a driver, 150
Concepts DLKM, 422
conf.h header file, 478
configuration
  ARP, 184
  IP, 184
  lanscan output, 183
configuration entry points, 284
Configuration Routines, Writing, 97
Configuration Space Accesses, 48
Configuration Space Services, 138
Context Dependent I/O Modules, 27
Context Dependent Input/Output (CDIO), 452
Context-Dependent I/O (CDIO), 26, 27
Control Block, SCSI, 303
crond, 496
core functions, network driver, 190
Controller commands, 496
Core Context-Dependent I/O (CORE CDIO), 27
Current Negotiated Information, SCSI, 326

D
data
  fields, bound SAP, 197
  fields, enet_dlpi_data_t, 188
  fields, enet_ifi, 195
  fields, enlan_ifi, 196
  link interface layer, 163
  protection
    SCSI, 375
  Structure Diagram, SCSI, 316
  Structures, SCSI, 303
Data Caching, 34
Data Link Provider Interface (DLPI), 183
Data Link Service (DLS), 262
data structures
  kernel, 40
Data Structures, Interface Card Management, 135
dd_close(), SCSI, 361
dd_ioctl(), SCSI, 362
dd_ioctl_okay(), SCSI, 364
dd_open(), SCSI, 365
dd_pass_thru_done(), SCSI, 367
dd_pass_thru_okay(), SCSI, 368
dd_read(), SCSI, 368
dd_start(), SCSI, 369
dd_strategy(), SCSI, 372
dd_write(), SCSI, 374
Deadlocks
  overview, 77
Defining Installation Structures, 86
Depot
  Installing, 492
  Managing, 492
  Packaging, 492
  Removing, 492
Description DLKM, 423
Description Service, 140
device
  /DLPI drivers synchronization, 186
  class header files, 348
  drivers, STREAMS, 167
  file, 163, 183
  switch table
    SCSI, 352
Device Driver Environment (DDE), 280
Device Driver Interface Management, 135
Device Driver Interrupt Driven Routines, 135
Device Driver, Sample Header, 90
device drivers, 33
  and major numbers, 35
  kernel association, 35
Device Queue Management, 135
Device Special File, create, 160
device special files
  major and minor numbers, 35
Device Switch Tables, 36
devices
  block, 33
  character, 33
  raw, 33
Direct Memory Access (DMA), 281
Directory Depot, 492
Disk Space Analysis (DSA), 551
disksort_enqueue() SCSI Service, 377
DLKM
  Advantages, 426
  Concepts, 422
  Description, 423
  Driver Loading Concepts, 426
  Driver Unloading Concepts, 427
  DriverConfiguration Concepts, 427
Kernel Configuration Tool Set Commands and Options, 431
  kernel configuration tool use, 429
Kernel Configuration Tools Description, 430
  Master File Definition, 428
  Miscellaneous Modules, 429
  Modstub.o File Definition, 428
  Module Packaging, 425
  Module Types, 426
Procedures for Dynamically Configured Loadable Modules, 433
  Procedures for Statically Configured Loadable Modules, 440
  space.h File Definition, 429
STREAMS Drivers, 429
  system File Definition, 428
  Terms and Concepts, 433
  Tools, 429
Tools for Building Static or Dynamic Kernels, 430
Tools that interface to DLKM, 430
DLKM Overview, 422
DLPI
  primitives and IOCTLs, 193
  protocol, primitives, 187
  protocol, sequence, 183
DLPI 2.0, 261
DLS, 262, 264
DLS providers
  ATM, 261
  HyperFabric, 261
  X.25, 261
DMA Mapping, 52
DMA Services, 139
dma.h header file, 478
dma_sync_IO, rules, 51
driver
  control commands and processing, 197
  environment, 253
  install routine
    SCSI, 351
Installation and Initialization, SCSI, 321
  set up structures
    SCSI, 349
Driver Configuration Concepts DLKM, 427
Driver Configuration Overview, 97
driver environment, 25
Driver Event Handling Services, 142
Driver Event Timeout Values, OLAR, 409
driver fragment skeleton, 46
Driver Handler, Registering, 144
Driver Loading Concepts DLKM, 426
Driver Master File, Creating, 152
Driver Master File, editing, 152
Driver Master File, example, 154
Driver Modifications for OLAR Support, 408
Driver Modifications Needed, OLAR, 405
Driver Name, Choosing, 83
Driver Probe Routines, Writing, 103
Driver Registering its Event Handler, OLAR, 408
Driver Registering its Event Mask, OLAR, 408
Driver Unloading Concepts DLKM, 427
driverAttach(), 175
driverAttach() routine, 100
driverAttach(), diagnostics, 101
driver_close()
  declaration, 114
  SCSI, 356
driver_close() routine, 114
driver_close(), Exclusive Open Device, 115
driver_close(), Multiple Open Device, 115
driver_close(), Shared Open Device, 115
driver_dev_init()
  SCSI, 352
driver_dev_init() routine, 102
driver_if_init() routine, 102
driver_install() routine, 98
driver_install(), Device and WSIO Pseudo Drivers, 99
driver_install(), Interface Drivers, 99
driver_install(), Monolithic Drivers, 100
driver_install(), non-WSIO Pseudo Drivers, 99
driver_ioctl() routine, 120
driver_isr() routine, 132
driver_minor_build() routine, 111
driver_minphys() routine, 125
driver_open() routine, 112
driver_open() routine, 112
driver_psize() routine, 134
driver_read() routine, 132
driver_read() routine, 137
driver_select() routine, 126
driver_strategy() routine, 128
driver_strategy() routine, 128
driver_write() routine, 117
Drivers
pre-11.0, 471
drivers
adding Series 700 driver to configuration files, 152
device, 33
interface, 33
interrupt context, 37
lower half, 37
pseudo, 33
two levels, 37
upper half, 37
drivers, LBI, 480
drivers, TBI, 480
drv_info_t Structure, 88
drv_ops_t Structure, 86
Dynamically Configured Loadable Module (DCLM), 456
dynamically assigned major numbers, 35
Dynamically Loadable Kernel Module (DLKM), 55
Dynamically Loadable Kernel Modules (DLKMs), 422

enet_dlipi_wakeup(), 186
enet_hwrq(), 200
enet_if() data fields, 195
enet_if() structure, 194
enet_isr() Interrupt Service Routine, 204
enet_process_looper() routine, 203
enet_process_packet() routine, 202
enet_receive_pkts() routine, 202
enet_slow_hw_req(), 201
enet_transmit_compl(), 201
ENET_TRANSMIT_FRAME, 201
enet_transmit_pended_frames(), 201
enlan_if structure, 196
data fields, 196
entry point routines, SCSI, 356
Entry Point Routines, Writing, 112
environment layer, STREAMS, 164
EPIC bridge, 205
Error handling
SCSI, 358
event flags, LBI, 481
event flags, TBI, 484
Event Handling Function, OLAR, 411
Event Timeout Values, Setting, 146
example
I/O read request, 39
exclusive open, 112
Execution Phase, 526
External Data Structures, 303
extracting the system file, 156

F
fast path data transmission, 192
Fastpath, 262, 264
Fibre Channel (FC), 279, 283
figures
Driver Configurations, 28
WSIO CDIO as a buffer zone, 29
File System Support, 34
Files, Device Special, 35
fileset
is_kernel, 515
is_reboot, 515
filesets, 493, 494
First In First Out (FIFOs), 335
flow, interrupt migration, 488
format, packet protocol, 180
Functions
Probing, 282
WSIO I/O Tree Management, 282
functions, message service, 190

G
General I/O (GIO), 56
General I/O System (GIO), 26
Generic Function Switch, 40
Glance/UX, 74
Graphical User Interface (GUI), 496
Index

**H**
- handling interrupts, 44
- Handling Timeouts, SCSI, 333
- Hardware Considerations, 64-bit safety, 476

<table>
<thead>
<tr>
<th>Header file</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>conf.h,</td>
<td>478</td>
</tr>
<tr>
<td>dma.h,</td>
<td>478</td>
</tr>
<tr>
<td>uio.h,</td>
<td>478</td>
</tr>
</tbody>
</table>

**header files**
- device classes, 348
- general, 348
- SCSI, 348

**Header Files, System Defined, 84**
- Host Bus Adapter, 283
- Host Bus Adapter (HBA), 286
- Host Bus Adapters (HBAs), 279
- HP-UX Logging and Tracing, network, 229
- HPUXAdditions, 502, 503
- hw_dlpis structure, 168, 170
- hw_ifts structure, 168
- protection, 171
- HyperFabric, 261

**I**
- I/O
  - flow of a request, 37
  - kernel data structures, 40
  - path, SCSI, 318
  - system calls, 37
  - within SCSI Device Driver, 319
  - within SCSI Interface Layer, 320
  - within SCSI Service Layer, 320
- I/O Adapters, 53
- I/O Control Blocks (IOBC), 342
- I/O Port Access Services, 49
- I/O Port Space Services, 138
- I/O subsystem structure, 26
- I/O Switch Tables, 40
- I/O Synchronization Service, 141
- I/O virtual, 52
- I/O Virtual Address (IOVA), 52, 53
- I/O Virtual Address View (IOVA), 52
- ID number, subsystem, N, network, 229
- Identifying Routines for Driver, 94
- ifEntry, 208
- ifIndex, 208
- ifTable, 208
- ILP32 data models, 472
- inbound path, 201
- Independent Hardware Vendor (IHV), 211
- Independent Software Vendors (ISVs), 422
- Installed Product Database (IPD), 516
- Instant Capacity on Demand (ICOD), 488
- int32_t, 474
- int64_t, 475
- Integrated Controller, 496
- interface
  - layer, data link, 163
  - name, 183
- Interface Driver, Sample Header, 91
- interface drivers, 33
- Interface Function Switch, 41
- interface node, 282
- Interface Select Code (ISC), 27, 40
- interface select code table, 40
- Internet (INET), 163
- interrupt
  - Processing, SCSI, 332
  - Service Routine (ISR), 288
  - Interrupt Control Stack (ICS), 57, 72, 179, 285
  - interrupt line sharing, 480
  - Interrupt Migration, 480
  - interrupt migration flow, 488
  - Interrupt ReQuest (IRQ), 44
  - Interrupt Service Routine
    - enet_isr(), 204
  - interrupt service routine, 132
  - Interrupt Service Routine (ISR), 37, 75, 332
  - Interrupt Services, 140
  - interrupts
    - handling, 44
    - interrupts, transaction based, 480
    - interrupts: line based, 480
  - intptr_t, 475
  - ioctl command word, 121
  - IOCTL processing, 191
  - IODIR, 55
  - iovc structure, 41
  - IP, 264
  - IP configuration, 184
  - is_kernel, 515
  - is_reboot, 515
  - ISC, SCSI, 307
  - isc_table_type Structure, 135
  - iSCSI, 279
  - isrlink(), 489
  - isrunlink(), 489
  - Itanium Architecture (IA), 219

**K**
- kernel and drivers association, 35
- Kernel Config.
  - Tool Set Commands and Options DLKM, 431
- kernel configuration tool, 429
- Kernel Configuration Tools Description DLKM, 430
- Kernel Containing Driver, build, 157
- kernel data structures, 40
- Kernel Device Table (KDT), 26
- Kernel ID String (KIS), 442
- Kernel-Specific (KRS), 432
- kernel-specific (KRS), 441
- kget_log_instance(), network, 235
- KLOG_CK(), network, 235
- klogg_write(), network, 235
- ktrc_write(), network, 233

**L**
- lanadmin, 211
- lanscan, 211
- output configurations, 183
- LaserRX, 74
- LBI drivers, 480
- LBI event flags, 481
- LBI processing, 488
- Line Based interrupt (LBI), 481

575
Index

line based interrupts, 480
Line Based interrupts (LBIs), 480
line sharing, interrupt, 480
Link Level Control (LLC), 169, 179, 180
linkloop, 211
LLC Header template, 264
lock
  orders, SCSI, 375
Lock acquisition rules, 77
Lock orders, 77
logged_info, 197
logged_link, 197
logging, network, 225
logging/tracing
  assigning subsystem ID, network, 229
  classifying log data, network, 230
  classifying trace data, network, 229
  formatting trace data, network, 230
HP-UX, network, 229
makefiles, network, 232
network, 226
source files, network, 232
what and when, network, 231
Logical Unit (LUN), 312
Logical Unit Number (LUN), 355
Logical Unit Structure, SCSI, 312
Logical Volume Manager (LVM), 33
LP64 considerations, 121, 472
LP64 data models, 472
LUN Lock, SCSI, 375

M
m_scsi_lun() SCSI Service, 377
machine_type, 504
major and minor numbers
  in device special files, 35
major numbers
  and device drivers, 35
  defined, 35
  dynamic assignment and ranges, 35
Mapping I/O Space, 47
master
  agent, 207
master file, 447
Master File Definition DLKM, 428
master file keywords, 447
$DRIVER_DEPENDENCY, 447
$DRIVER_INSTALL, 447
$INTERFACE, 447
$LOADABLE, 447
$TUNABLE, 447
$TYPE, 447
$VERSION, 447
stub, 447
Maximum Transmission Unit (MTU), 216
maximum transmission unit (mtu), 169
mechanism
  software trigger, 45
Mechanisms synchronization, 73
Memory Allocation, 47
Memory Allocation Services, 47, 142
Memory Mapped Services Registers, 48
Menu Mode, 218
message
  block reception, 192
  block transmission, 192
  service functions, 190
Message Blocks (MBLKS), 266
MIB-II, 218
Migration, Interrupt, 480
milliseconds between retries (msecs), 359
minor numbers
  defined, 35
Miscellaneous Modules DLKM, 429
mod.o file, 448
modlink, 444
Modstub.o, 448
Modstub.o file definition, 448
Module Packaging DLKM, 425
Module Types DLKM, 426
modwrapper, 444
MSGCKO, 266
MTU
  resetting, 217
multiple open, 112
Multiprocessing
  Idle Time, 74
  Timing hazards, 74
Multiprocessing to Uniprocessing Comparison, 72
Multiprocessor (MP), 355
multiprocessor (MP), 72
Multiprocessor implications
  SCSI, 355

N
N, subsystem ID number, network, 229
National Language Support (NLS), 247
Native Language Support (NLS), 247
netfmt(1M), network, 228
NetTL Team, 251
nettl(1M), network, 228
nettlconf(1M), network, 228
nettlgen.conf(4), network, 227
Network
  Assign Subsystem ID, 229
  Calling driver_attach(), 175
  Calling driver_install(), 174
  Classify Log Data, 230
  Classify Trace Data, 229
  Control Functions, 190, 197
  Data Link Interface Layer, 163
  Data Structures, 194
  Data Structures and Interfaces, 168
  Device Driver/DLPI Driver Synchronization, 186
  Device File, Interface Name, and PPA Number, 183
  DLPI Interface to Upper Layers, 183
  Driver Development Steps, 166
  Driver Structure Protection, 171
  Driver Support for DLPI, 187
  Driving the NIC, 194

576
Entering STREAMS from ICS, 187
Format Trace Data, 230
HP-UX Networking Interface Architecture, 162
hw_dp1 Structure, 170
hw_ift Structure Protection, 171
hw_ift_t Structure, 168
Inbound Path, 201
Initializing Networking Device Drivers, 173
IOCTL Processing, 191
IP and ARP Configuration, 184
Logging Code Sample, 236
Major Data Structures, 187
Management Support, 207
Message Block and Queue Functions and Macros, 182
Network Protocol Layer, 163
Networking Driver Structure, 165
Opening and Closing a Driver, 189
Outbound Path, 199
Packet Protocol Format, 180
Passing Data to HP-UX Tracing/Logging, 232
Platform Specifics, 205
Protection and Synchronization, 171
Protocol Binding and Demultiplexing, 179
Protocol Interface Layer, 163
Reception of Message Blocks, 192
STREAMS DLPI and Network Driver, 185
STREAMS Environment, 164
STREAMS Model Device Drivers, 167
STREAMS Synchronization, 187
Summary of DLPI Primitives and IOCTSs, 192
Tracing and Logging, 225
Transmission of Message Blocks, 192
V Class, 205
network driver
closing, 189
core control functions, 190
driver_attach(), 175
driver_install(), 174
initializing, 173
opening, 189
sequence, STREAMS DLPI, 185
Network Interface Card (NIC), 265
network interface, architecture, 162
network management ID, 169
newconfig, 526
NIC, 194, 265
node
interface, 282
transparent, 282
Noncoherent I/O Systems, 53
Number of Bytes per Page (NBPG), 312

O

OLAR
Driver Event Timeout Values, 409
Driver Modifications for OLAR Support, 408
Driver Modifications Needed, 405
Driver Registering its Event Handler, 408
Driver Registering its Event Mask, 408
Event Handling Function, 411
Handling, 412
Introduction, 404
Miscellaneous Changes Required at the Driver, 418
On Line Addition Process, 412
On Line Replacement Process, 413
Overview, 403
Performing OLAR of PCI I/O Cards, 419
Resume, 416
Supported Hardware and Software, 404
Suspend, 415
WSIO Event Handling Structures, 406
WSIO Interfaces, 406
wsio_drv_info_t, 406
On Line Addition Process, OLAR, 412
On Line Replacement Process, OLAR, 413
On-Line Addition (OLA), 405, 412
On-Line Addition/Replacement (OLA/R), 403
On-Line Replacement (OLR), 412
Ordered Apps, 503
Ordered Interrupts Service, 141
OrderedApps, 499, 502
os_name, 504
os_release, 504
outbound path, 199
Overview DLKM, 422

P

packet
protocol, format, 180
type, network, 179
Page Directroy (PDIR), 53
Parallel SCSI (pSCSI), 286
parallel SCSI (pSCSI), 279
Patch, 499
PCI
Automatic IRQ determination, 386
Blocking transactions, 381
Bus-dependent implementation, 392
Bus-independent functionality, 392
byte swapping, 383
Configuration space restrictions, 387
Configuration Spaces, 380
Constants and data structures, 396
Device operation, 389
Device setup, 385
DMA, 390
driver entry point routines, 402
Endian issues, 383
I/O base registers, 386
I/O space, 381
Leveraging existing drivers, 392
Mapping base address registers into I/O space, 385
Mapping base address registers into PCI memory, 385
Mapping the memory base register, 386
Mastered reads and writes to host memory, 382
Masters and Coherence, 391
Memory base registers, 385
Index

Multiprocessor safety, 395
Nonblocking transactions, 381
PCI_LITTLE_ENDIAN_ONLY flag, 389
pre-swapping, 384
Processor and card mastered reads and writes to host memory, 382
Processor mastered reads and writes, 381
Register Spaces, 380
sample driver_attach(), 399
sample driver_install(), 399
Transaction ordering, 381
Unsupported WSIO functions, 394
Using base address registers, 385
Write side-effects, 381

PCI Context-Dependent I/O (PCI CDIO), 27
PCI OLAR Support, 144
Physical Address View, 52
Physical Point of Attachment (PPA), 169, 220
physical processor, 52
PPA number, 183
Precision Architecture (PA), 219, 327
Precision Architecture Context-Dependent I/O (PA CDIO), 27
primitives and IOCTLs, DLPI, 193
Procedures for Dynamically Configured Loadable Modules DLKM, 433
Procedures for Statically Configured Loadable Modules DLKM, 440
processing
Completed I/O Requests, SCSI, 330
packets, 179
Product Specification File (PSF), 492, 496
products, 493
Programmed I/O (PIO), 281
Promiscuous, Inbound and Outbound, 181
Protection and Synchronization, SCSI, 328
protocol
binding and unbinding, 179
demultiplexing, 179
interface layer, 163
kind, network, 179
layer, 163
value, 179
Protocol Data Units (PDU), 229
pSCSI, 279
Pseudo Driver, Sample Header, 92
pseudo drivers, 33
PSF, 496
ptr32_t, 475

Q
Qlisp 12160 card, 345
qlisp driver, SCSI, 340
qlisp_build_cmd 64, SCSI, 342
qlisp_build_cmd, SCSI, 342
qlisp_call_cbfn, SCSI, 344
qlisp_fast_complete, SCSI, 344
qlisp_get_req_iocb, SCSI, 343
qlisp_if_abort, SCSI, 346
qlisp_if_bdr, SCSI, 346
qlisp_if_lun_open, SCSI, 346
qlisp_if_reset_bus, SCSI, 346
qlisp_if_start, SCSI, 343
qlisp_init_interruptions, SCSI, 341
qlisp_init_alloc_queues, SCSI, 341
qlisp_init_attributes, SCSI, 341
qlisp_init_common, SCSI, 341
qlisp_init firmware, SCSI, 341
qlisp_init_ifsw, SCSI, 342
qlisp_init_master, SCSI, 342
qlisp_init_pci_cfg, SCSI, 340
qlisp_init_reset_card, SCSI, 340
qlisp_init_slave, SCSI, 342
qlisp_init_start_queues, SCSI, 342
qlisp_iocb_cnt, SCSI, 343
qlisp_isr, SCSI, 345
qlisp_mailbox_cmd, SCSI, 342
qlisp_multi driver, SCSI, 336
qlisp_multi_dev_probe, SCSI, 338
qlisp_pci_attach, SCSI, 342
qlisp_pdk_init_common(), SCSI, 341
qlisp_pdk_load_nvram_values(), SCSI, 341
qlisp_pdk_nvram_read(), SCSI, 341
qlisp_pdk_nvram_send_bits(), SCSI, 341
qlisp_post_reset_delay, SCSI, 344
qlisp_process_rsp, SCSI, 345
qlisp_reset_bus, SCSI, 346
qlisp_send_marker, SCSI, 346

R
raw devices, 33
Real Time Extension (RTE), 488
Register
Mapping and DMA, SCSI, 327
Register Services, 137
regular data path transmission, 192
Releasing Pending Timeouts, 204
Remote Procedure Call (RPC), 496
request entry points, 284
Reset Bus, SCSI, 333
Resume, OLAR, 416
RFC 1213, 218
RFC 1398, 218
routines
calling physio(), 119
driver_strategy() read, 130
driver_strategy() write, 129
software trigger, 45
writing a driver_attach(), 100
writing a driver_close(), 114
writing a driver_dev_init(), 102
writing a driver_if_init(), 102
writing a driver_install(), 98
writing a driver_ioctl(), 120
writing a driver_isr(), 132
writing a driver_minor_build(), 111
writing a driver_minphys(), 125
writing a driver_open(), 112
writing a driver_psize(), 134
writing a driver_read() routine, 117
writing a driver_select(), 126
writing a driver_strategy(), 128
writing a driver_write() routine, 117
Routines and services
Index

SCSI, 360
Rules for lock acquisition, 77

S
Sample Driver
SCSI, 335
SCSI
Abort an I/O Request, 333
Bus Device Reset, 332
Bus Lock, 375
Bus Structure, 315
Calling driver_attach(), 323
Calling driver_install(), 321
Control Block, 303
Current Negotiated Information, 326
Data protection, 375
Data Structure Diagram, 316
Data Structures, 303
dd_close() Routine, 361
dd_ioctl(), 362
dd_ioctl1Okay(), 364
dd_open(), 365
dd_pass_thru_done(), 367
dd_pass_thru_okay(), 368
dd_read(), 368
dd_start(), 369
dd_strategy(), 372
dd_write(), 374
Device switch table, 352
Driver install routine, 351
Driver Installation and Initialization, 321
Driver set up structures, 349
driver_close(), 356
driver_dev_init(), 352
driver_ioctl(), 357
driver_open(), 356
driver_psize(), 357
driver_read(), 356
driver_strategy(), 357
driver_write(), 356
Entry point routines, 356
Error handling, 358
Handling Timeouts, 333
I/O Path, 318
I/O within SCSI Device Driver, 319
I/O Within SCSI Interface Layer, 320
I/O Within SCSI Service Layer, 320
Include header files, 348
Interrupt Processing, 332
ISC, 307
Lock and semaphore orders, 375
Logical Unit Structure, 312
LUN Lock, 375
Multiprocessor implications, 355
Parameters, Setting, 325
Processing Completed I/O Requests, 330
Protection and Synchronization, 328
qlisp driver, 340
qlisp_build_cmd, 342
qlisp_build_cmd 64, 342
qlisp_call_cbfns, 344
qlisp_fast_complete, 344
qlisp_get_req_iocb, 343
qlisp_if_abort, 346
qlisp_if_bdr, 346
qlisp_if_lun_open, 346
qlisp_if_reset_bus, 346
qlisp_if_start, 343
qlisp_in init interrupts, 341
qlisp_in init_alloc_queues, 341
qlisp_in init_attributes, 341
qlisp_in init_common, 341
qlisp_in init_firmware, 341
qlisp_in init_ifsw, 342
qlisp_in init_master, 342
qlisp_in init_pciCfg, 340
qlisp_in init_reset_card, 340
qlisp_in init_slave, 342
qlisp_in init_start_queues, 342
qlisp_iocb_cnt, 343
qlisp_isr, 345
qlisp_mailbox_cmd, 342
qlisp_multi driver, 336
qlisp_multi_dev_probe, 333
qlisp_pci_attach, 342
qlisp_pdk_in init_common(), 341
qlisp_pdk_load_nvram_values(), 341
qlisp_pdk_nvram_read(), 341
qlisp_pdk_nvram_send_bits(), 341
qlisp_post_reset_delay, 344
qlisp_process_rsp, 345
qlisp_reset_bus, 346
qlisp_send_marker, 346
Register Mapping and DMA, 327
Reset Bus, 333
Routines and services, 360
Sample Driver, 335
SDTR/WDTR Negotiation, 326
select_q, 316
Sending SCSI I/O Requests to the Target, 329
Services, 377
disksort_enqueue(), 377
m_scsi_lun(), 377
scsi_action(), 377
scsi_cmd(), 377
scsi_cmdx(), 377
scsi_ddsw_init(), 377
scsi_dequeue(), 377
scsi_dequeue_bp(), 377
scsi_init_inquiry_data(), 377
scsi_ioctl(), 377
scsi_log_io(), 377
scsi_lun_open(), 377
scsi_read(), 377
scsi_sense_action(), 377
scsi_snooze(), 377
scsi_strategy(), 377
scsi_write(), 377

579
Index

Sleep and Wakeup Mechanism, 334
Special Routines, 332
Specific Part, 321, 323
struct buf, 310
Subsystem Lock, 375
Target Lock, 375
Target Structure, 313
Transport Driver Development, 321
Underlying routines, 360
What to Negotiate, 326
When to Negotiate, 326
SCSI Class Drivers, 279
SCSI Command Blocks (CDBs), 284
SCSI Interface Drivers, 279
SCSI Transport Drivers, 279
scsi_action() SCSI Service, 377
scsi_cmd() SCSI Service, 377
scsi_cmdx() SCSI Service, 377
scsi_dequeue() SCSI Service, 377
scsi_dequeue_bp() SCSI Service, 377
SCSI_GET_INITIATOR_PARAMS, 325
scsi_init_inquiry_data() SCSI Service, 377
scsi_ioctl() SCSI Service, 377
scsi_log_io() SCSI Service, 377
scsi_lun_open() SCSI Service, 377
scsi_probe(), 109
scsi_read() SCSI Service, 377
scsi_sense_action() SCSI Service, 377
scsi_snooze() SCSI Service, 377
scsi_strategy() SCSI Service, 377
scsi_write() SCSI Service, 377
SD Components
Agent, 496
Controller, 496
Packager, 496
SD Object Attributes
SD-Bundle, 498
SD-Category, 498
SD-Fileset, 498
SD-Product, 498
SD-Subproduct, 498
Vendor, 498
SD-Bundle, 535
SD-Bundles, 495
SD-Fileset, 535
SD-Filesets, 494, 495
SD-Product, 494, 535
attributes, 494
filesets, 494
packaging, 494
Subproducts, 494
SD-Products, 495
SD-Subproduct, 535
SD-Subproducts, 495
SDTR/WDTR Negotiation, SCSI, 326
select code
interface, 40
select queue, 285
select_q, SCSI, 316
Semaphore orders
SCSI, 375
Sending SCSI I/O Requests to the Target, 329
Services
SCSI, 377
Setting SCSI Parameters, 325
shared open, 112
single processor (UP), 355
skeletons
driver fragment, 46
Sleep and Wakeup Mechanism, SCSI, 334
sleep(), 78
Software Considerations, 64-bit safety, 476
Software Depot (SD), 248, 491, 492
software trigger mechanism, 45
software trigger routine, 45
space.h, 448
space.h file definition, 448
space.h File Definition DLKM, 429
Special Routines, SCSI, 332
Specific Part, SCSI, 321, 323
Spinlock routines
overview, 75
Spinlocks, 73
overview, 75
spinlocks, 73, 288
deadlock avoidance, 171
protection model, 171
Statically Configured Loadable Module (SCLM), 456
STREAMS
device driver, 167
DLPI Network Driver Sequence, 185
environment layer, 164
head, 163
synchronization, 187
STREAMS Drivers DLKM, 429
struct buf, SCSI, 310
structure
uio, 41
structures
buf, 40
enet_dlpi_data_t, 187
enet_if_t, 194
enlan_if_t, 196
hw_dlpi, 168, 170
hw_if_t, 168
iovec, 41
stubs, 448
Subagent, 207
subproduct.tag
Demonstration, 508
Development, 508
Help, 508
HelpByLang, 508
Manuals, 508
ManualsByLang, 508
MessagesByLang, 508
ReleaseNotes, 508
subproducts, 493
subsystem
ID number, N, network, 229
Lock, SCSI, 375
superuser privilege, 216, 217, 219
support modules
Driver, 265
Hardware, 265
Transport Stack, 265
Supported Hardware and Software, OLAR, 404
Suspend, OLAR, 415
swpackage, 496
synchronization
device/DLPI drivers, 186
Mechanisms, 73
sleep(), 78
STREAMS, 187
wakeup(), 78
Synchronization Mechanisms, 73
Synchronous Data Transfer Request (SDTR), 288, 326
System Administration Manager (SAM), 167, 253, 412
System Attribute Services, 141
system calls, 37
system file, 447
System File Definition DLKM, 428
system file keywords, 447
$CONFIGURE, 447
$LOADABLE, 447
$TUNABLE, 448
$VERSION, 447
system file, extracting the, 156
System Header Files, Choosing, 84

T
Tape Depot, 492
Target
Lock, SCSI, 375
Structure, SCSI, 313
TBI drivers, 480
TBI event flags, 484
TBI processing, 488
TCP, 264, 265
TCP/IP, 264
TCP/UDP, 265
template
Link Layer Header, 264
Terms and Concepts DLKM, 423
timeout mechanisms, 42
timeout() routine, 42
Timeouts, Explanation, 42
Token Ring interfaces, 215
Tools DLKM, 429
Tools for Building Static or Dynamic Kernels DLKM, 430
Tools that interface to DLKM, 430
tracing, network, 225
tracing/logging
assigning subsystem ID, network, 229
classifying log data, network, 230
classifying trace data, network, 229
formatting trace data, network, 230
makefiles, network, 232
network, 226
source files, network, 232
Transaction Based interrupt (TBI), 484
transaction based interrupts, 480
Transaction Based interrupts (TBIs), 480
transmission
fast data path, 192
message block, 192
regular data path, 192
transparent node, 282
transport
Driver Development, SCSI, 321
TrialUseApps, 499, 502, 503
troubleshooting, HP approach, network, 225

U
UDP, 265
uio structure, 41
uo.h header file, 478
uiomove(), 119
Underlying routines
SCSI, 360
uint32_t, 474
uint64_t, 475
unintptr_t, 475
Uniprocessing to Multiprocessing comparison, 72
uniprocessor (UP), 72
unt imeout() routine, 43
usage1, 550
usage2, 550
using physio(), 118

V
V class PCI I/O architecture, 205
vendortag, 502
virtual, 52
Virtual Address View, 52
Virtual Local Area Network (VLAN), 263

W
wakeup(), 78
Wide Data Transfer Request (WDTR), 326
Wide Data Transport Request (WDTR), 304
Workstation I/O (WSIO), 27
Workstation Input/Output (WSIO), 443
Writing a Driver, Overview, 82
WSIO Event Handling Structures, OLAR, 406
WSIO Interfaces, OLAR, 406
WSIO Mapping Services, 54
WSIO Services, new for 11.11, 136
wsio_completion_cb(), 487
wsio_drv_data_t Structure, 89
wsio_drv_info_t Structure, 90
wsio_drv_info_t OLAR, 406
WSIO_EVENT_LBI_INTR_MIGR, 481
WSIO_EVENT_OFFLINE_CPU, 484
WSIO_EVENT_ONLINE_CPU, 487
wsio_generic_event_t, 481
wsio_intr_activate(), 489
wsio_intr_alloc(), 489
wsio_intr_deactivate(), 486, 489
wsio_intr_enable(), 486, 487
wsio_intr_free(), 489
wsio_intr_get_assigned_cpu(), 486, 489
wsio_intr_get_irq_line(), 489
wsio_intr_get_txn_info(), 486, 489
wsio_intr_migr_info_t, 481
wsio_intr_set_cpu_spec(), 486, 489
wsio_reg_drv_capability_mask(), 481
wsio_register_addr_probe(), 104
wsio_register_dev_probe(), 103