# Table of Contents

## 1 Driver Environment Overview

- I/O Subsystem Structure ................................................................. 29
- General I/O System ........................................................................ 29
- Context Dependent I/O Modules (CDIOs) ..................................... 30
  - Basic Components of a CDIO .................................................. 30
  - How the Driver Environment Works ...................................... 31

## 2 HP-UX I/O Subsystem Features

- Driver Types .................................................................................. 33
  - Device, Pseudo, and Interface Drivers ..................................... 33
  - Block Devices ............................................................................. 34
  - Character Devices ..................................................................... 34
- Major and Minor Numbers ............................................................. 34
  - Assigning Major Numbers ...................................................... 34
  - Using Minor Numbers ............................................................ 35
  - Device Special Files ............................................................... 35
  - Device Switch Tables ............................................................ 35
- System Privileges .......................................................................... 36
- System Calls .................................................................................. 36
  - Parts of a Driver ....................................................................... 36
  - I/O Request Flow ...................................................................... 37
  - I/O Read Request Sample ...................................................... 37
- Kernel Data Structures ................................................................... 38
  - buf Structure ............................................................................ 38
  - isc_table_type Structure ........................................................ 38
  - I/O Switch Tables ..................................................................... 39
    - Generic Function Switch .................................................... 39
      - Interface Function Switch ................................................. 39
  - uio Structure ........................................................................... 39
  - iovec Structure ......................................................................... 39
  - io_post_event_req Structure ................................................. 40
  - wsio_reg_node Structure ....................................................... 40
  - wsio_legacy_info Structure .................................................... 40
- Timeout Mechanisms ..................................................................... 40
  - How Timeouts Work ............................................................... 40
  - timeout Routine ....................................................................... 40
  - untimout Routine ..................................................................... 41
- Interrupt Handling ........................................................................ 41
  - Interrupt Services ..................................................................... 41
  - Interrupt Migration ................................................................. 42
- Memory Allocation and Mapping I/O Space ................................. 43
  - Memory Allocation Services .................................................... 43
  - I/O Space Services .................................................................... 43
    - Register Services ................................................................... 43
    - Configuration Space Services ............................................. 44
    - I/O Port Space Services ....................................................... 44
- Cache Coherence ........................................................................... 45
3 Multiprocessing.......................................................................................................................... 69
Uniprocessing and Multiprocessing Comparison..............................................................................69
Synchronization Mechanisms........................................................................................................... 69
  Adaptive Locking.......................................................................................................................... 70
  Spinlocks ..................................................................................................................................... 70
    Spinlock Routines......................................................................................................................71
  Mutexes......................................................................................................................................... 71
    Mutex Routines.......................................................................................................................... 71
  Reader-Writer Locks...................................................................................................................... 72
    Reader-Writer Lock Routines...................................................................................................... 72
  Reader-Writer Spinlocks................................................................................................................ 72
    Reader-Writer Spinlock Routines............................................................................................... 72
Deadlocks........................................................................................................................................... 73
  Lock Acquisition Rules.................................................................................................................. 73
  Lock Orders.................................................................................................................................. 73
Synchronization with Condition Variables....................................................................................... 73
  Condition Variable Routines......................................................................................................... 74
Counting Semaphores..........................................................................................................................74
  Counting Semaphore Routines...................................................................................................... 74

4 Writing a Driver............................................................................................................................ 77
Step 1: Choosing a Driver Name ...................................................................................................... 77
Step 2: Choosing System Header Files.............................................................................................77
  Header Files for All Drivers..........................................................................................................78
  Header Files for Disk Drivers........................................................................................................ 78
  Header Files for Tape Drivers........................................................................................................ 78
Step 3: Defining Installation Structures............................................................................................78
  drv_ops_t Structure......................................................................................................................79
  drv_info_t Structure..................................................................................................................... 80
  wsio_drv_data_t Structure............................................................................................................ 81
  The wsio_drv_info_t Structure Type........................................................................................... 82
Step 4: Identifying Routines for the Driver....................................................................................... 85
  Configuration Routines.................................................................................................................. 86
  Entry Points.................................................................................................................................. 86
  Other Routines............................................................................................................................. 87
Step 5: Writing Configuration Routines.............................................................................................87
  Configuration Steps....................................................................................................................... 88
Writing a driver_install Routine..................................................................................................... 89
Writing a driver_attach Routine........................................................................................................91
Writing a driver_dev_init Routine......................................................................................................92
Writing a driver_if_init Routine...........................................................................................................92
Writing Driver Probe Routines.............................................................................................................93
  wsio_register_dev_probe Routine............................................................................................... 93
  wsio_register_addr_probe Routine............................................................................................... 95
Writing a driver_minor_build Routine...............................................................................................102
Step 6: Writing Entry Point Routines................................................................................................102
Writing a driver_open Routine..............................................................................................................102

Finding the Physical I/O Address.....................................................................................................65
Write Coalescing Guidelines (Integrity platform only).....................................................................66
5 Writing a DLKM Driver..............................................................................................................137

Dynamically Loadable Kernel Modules......................................................................................137
Dynamic Loading Concepts........................................................................................................137
Module Load Times......................................................................................................................137
Dependencies Between Modules..................................................................................................138
Module Load Sequence................................................................................................................138
Synchronization with Other Operations......................................................................................138
Automatic Module Loads...........................................................................................................139
Module Metadata for DLKMs........................................................................................................139
Dynamic Loading Data Structures............................................................................................139
Dynamic Loading Functions.......................................................................................................140
Module Type-Specific DLKM Requirements...............................................................................141
WSIO Class and Interface Drivers..............................................................................................141
Dynamic Loading Data Structures............................................................................................141
Module Preparation Scripts for Drivers....................................................................................142
6 Writing a Kernel Module.................................................................157
   Step 1: Creating a Source Code Directory...........................................157
   Step 2: Creating a Module Metadata File.............................................157
      Module Metadata Fields.................................................................158
         Sample Device Driver Module Metadata File.................................163
   Step 3: Writing the Module Source Code.............................................164
   Step 4: Adding DLKM Support............................................................164
   Step 5: Adding Tunable Parameter Support..........................................164
   Step 6: Embedding Debug Information...............................................164
   Step 7: Embedding Version Information..............................................164
   Step 8: Creating Module Preparation Scripts.....................................165
      Sample modprep Script...................................................................166
   Step 9: Building a Kernel Module.....................................................167
      Makefile.........................................................................................168
         Makefile.bld..............................................................................168
      Command Line Options....................................................................169
   Step 10: Installing a Kernel Module....................................................170
   Step 11: Testing and Using a Kernel Module........................................170
      Updating an Older HP-UX Kernel Module........................................171

7 Kernel Registry Services.................................................................173
   KRS Structure...................................................................................173
      KRS Tree Structure........................................................................173
   Values...............................................................................................174
   Keys....................................................................................................174
   Nodes..................................................................................................174
   Links...................................................................................................175
# Table of Contents

## 8 Adding Tunable Parameters to Kernel Modules
- Step 1: Defining a Tunable Parameter .................................................. 183
- Step 2: Adding the Tunable to the Metadata File ................................. 185
- Step 3: Writing Tunable Handler Functions ......................................... 185
  - Handler Capabilities ........................................................................... 186
  - Validating Tunable Values ................................................................. 187
  - Constraint Handlers ........................................................................... 188
  - Computed Default Values ................................................................... 188
  - Dynamic Changes to Tunable Values ................................................. 188
  - Avoiding Race Conditions .................................................................. 189
  - Automatic or Self-Tuning Tunables ................................................... 189
  - Notification of Tunable Changes ....................................................... 190
  - Percentage Tunable Parameters ....................................................... 190
  - Tunable Handler Function Example .................................................. 190
- Step 4: Writing Tunable Initialization Functions ................................. 193
- Step 5: Writing Tunable Teardown Code ............................................. 194
- Step 6: Building and Testing the Kernel ............................................. 195
- Step 7: Creating Tunable Documentation .......................................... 195
- Additional Tasks .................................................................................. 195
  - Changing Tunable Default Values or Attributes ............................... 195
  - How to Make a Tunable Dynamic or Self-Tuning ............................. 195

## 9 Writing PCI Drivers
- PCI Overview ....................................................................................... 197
- PCI Register Spaces ............................................................................ 197
  - PCI Configuration Space ................................................................. 197
  - PCI Memory Space ........................................................................... 197
  - PCI I/O Space ................................................................................ 198
- PCI Transaction Ordering .................................................................. 198
  - Processor-Mastered PCI Transaction .............................................. 198
  - Blocking versus Nonblocking Transactions ..................................... 198
  - Write Side Effects ........................................................................... 199
  - PCI Card-Mastered Transactions ..................................................... 199
  - Interleaved PCI Transactions .......................................................... 199
- PCI Endian Issues ................................................................................ 200
  - Byte Swapping ................................................................................. 200
  - Preswapping .................................................................................... 200
- PCI Device Setup ................................................................................ 201
<table>
<thead>
<tr>
<th>Property Exchange with Drivers</th>
<th>229</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registration</td>
<td>229</td>
</tr>
<tr>
<td>Outbound Processing</td>
<td>230</td>
</tr>
<tr>
<td>Inbound Processing</td>
<td>230</td>
</tr>
<tr>
<td>Control Processing</td>
<td>230</td>
</tr>
<tr>
<td>HP-UX Kernel Services</td>
<td>230</td>
</tr>
<tr>
<td>HP-UX LAN Commands</td>
<td>230</td>
</tr>
<tr>
<td>Driver Instance Data Structure</td>
<td>230</td>
</tr>
<tr>
<td>The ielan_if_t Data Structure</td>
<td>231</td>
</tr>
<tr>
<td>Property Exchange Between HP-DLPI and Drivers</td>
<td>233</td>
</tr>
<tr>
<td>Return Values</td>
<td>234</td>
</tr>
<tr>
<td>Rules for Using dlpi_propp()</td>
<td>234</td>
</tr>
<tr>
<td>HP-DLPI Properties and Associated Data Structures</td>
<td>234</td>
</tr>
<tr>
<td>Information for dl_hp_getinfo_t</td>
<td>235</td>
</tr>
<tr>
<td>Deallocating a Multicast Address List Example</td>
<td>235</td>
</tr>
<tr>
<td>Initializing Non-Native Networking Device Drivers</td>
<td>236</td>
</tr>
<tr>
<td>Step 1: Setting Up the Data Structures and the WSIO Event Handler</td>
<td>236</td>
</tr>
<tr>
<td>Step 2: Installing the Driver</td>
<td>237</td>
</tr>
<tr>
<td>Step 3: Calling driver_attach()</td>
<td>238</td>
</tr>
<tr>
<td>Step 4: Calling driver_init()</td>
<td>239</td>
</tr>
<tr>
<td>Step 4a: Verifying HP-DLPI Version Compatibility</td>
<td>240</td>
</tr>
<tr>
<td>Step 4b: Initializing the Card Instance Data</td>
<td>240</td>
</tr>
<tr>
<td>Step 4c: Registering a Driver Instance with HP-DLPI</td>
<td>241</td>
</tr>
<tr>
<td>Step 4d: Retrieving HP-DLPI Callback Functions and NM_ID</td>
<td>245</td>
</tr>
<tr>
<td>Step 4e: Initializing a MIB Data Structure</td>
<td>246</td>
</tr>
<tr>
<td>Step 4f: Informing HP-DLPI about the Current Administrative and Operational Status</td>
<td>247</td>
</tr>
<tr>
<td>Step 4g: Setting Up ISR for a Driver Instance</td>
<td>248</td>
</tr>
<tr>
<td>Step 5: Creating a Driver Metadata File</td>
<td>248</td>
</tr>
<tr>
<td>IELAN Metadata File Example</td>
<td>249</td>
</tr>
<tr>
<td>Driver Control Function</td>
<td>249</td>
</tr>
<tr>
<td>Control Flow Processing</td>
<td>250</td>
</tr>
<tr>
<td>Implementation Guidelines for driver_control()</td>
<td>251</td>
</tr>
<tr>
<td>Control Request Types</td>
<td>251</td>
</tr>
<tr>
<td>Handling a Primitive</td>
<td>251</td>
</tr>
<tr>
<td>Primitives Required to be Supported by Drivers</td>
<td>252</td>
</tr>
<tr>
<td>64-Bit MIB Statistics Support</td>
<td>254</td>
</tr>
<tr>
<td>Handling an Ioctl</td>
<td>254</td>
</tr>
<tr>
<td>Information on Transparent Ioctls</td>
<td>255</td>
</tr>
<tr>
<td>Information on dlpi_wakeupp()</td>
<td>256</td>
</tr>
<tr>
<td>IELAN Handling a DL_ENABMULTI_REQ Primitive</td>
<td>256</td>
</tr>
<tr>
<td>Step 1: Obtain the List of Currently Enabled Multicast Addresses</td>
<td>256</td>
</tr>
<tr>
<td>Step 2: Update Card Hardware to Handle Multicast Addresses</td>
<td>257</td>
</tr>
<tr>
<td>Step 3: Wake Up HP-DLPI when Processing Completes</td>
<td>257</td>
</tr>
<tr>
<td>DL_HP_SET_DRV_PARAM_IOCTL Ioctl</td>
<td>258</td>
</tr>
<tr>
<td>Driver-Specific Ioctls — IELAN_LINK_SPEED</td>
<td>260</td>
</tr>
<tr>
<td>Interrupt Service Routine</td>
<td>261</td>
</tr>
<tr>
<td>Rules for ISR</td>
<td>261</td>
</tr>
<tr>
<td>IELAN ISR</td>
<td>262</td>
</tr>
<tr>
<td>Inbound Frame Processing</td>
<td>266</td>
</tr>
<tr>
<td>Inbound Path Processing Rules</td>
<td>264</td>
</tr>
<tr>
<td>Updating Inbound MIB Statistics</td>
<td>264</td>
</tr>
<tr>
<td>IELAN Driver Inbound and Receive Routine</td>
<td>265</td>
</tr>
<tr>
<td>Processing Received Frames</td>
<td>266</td>
</tr>
<tr>
<td>Sending the Frame to HP-DLPI</td>
<td>267</td>
</tr>
<tr>
<td>Updating MIB Statistics</td>
<td>267</td>
</tr>
</tbody>
</table>
13 LAN Commands .......................................................................................................................... 337
   The nwmgr Command Architecture............................................................................................ 338
   How nwmgr Works..................................................................................................................... 338
   nwmgr Command Syntax ......................................................................................................... 338
14 Tracing and Logging in LAN Drivers.................................353

Overview...........................................................................353
  The nettlgen.conf File......................................................354
  The nettlconf Command..................................................354
  The nettl Command........................................................355
  The netfmt Command.......................................................355

Using HP-UX Logging and Tracing for Troubleshooting Support..................................................................355
  Assigning a Subsystem ID...............................................355
  Classifying Trace Data....................................................356
  Formatting Trace Data....................................................356
  Classifying Log Data.......................................................357
  What and When to Log.....................................................357

Passing Data to HP-UX Tracing and Logging.................................................................358
  KTRC_CK Macro..............................................................358
  The ktrc_write Routine....................................................359
  Tracing Code Example.....................................................360
  KLOG_CK Macro...............................................................361
  The kget_log_instance Routine......................................361
  The klogg_write Routine...............................................361
  Logging Code Example....................................................362

Formatting Networking Trace and Log Messages........................................................364
  Subformatter Registration with NetTL....................................364

Formatter Responsibilities and Features.......................................................364
  Formatting Routines........................................................365
  Utility Routines..............................................................365
  Trace Formatting Routines................................................366
  Examples...........................................................................367
  Nice Format Output........................................................369
  Terse Format Output........................................................370
  Raw Format Output..........................................................370

Designing a Subformatter...................................................370
  Subformatter Responsibilities..............................................371
  Subformatter Requirements...............................................371
  Building and Installing the Subformatter................................371

Internationalization and Message Catalog Support........................................................372
  Configuring Subsystems into the System.................................372
  Subformatter Configuration Script Example............................373
  Subsystem Installation Testing............................................375
  Testing Procedure............................................................375
  Troubleshooting Installation and Configuration.........................375
## 15 OOP and Transport IOCTLs

- DLPI IOCTL and Primitives ................................................................. 377
- DL_IOC_DRIVER_OPTIONS ................................................................. 377
- DL_IOC_HDR_INFO ioctl ................................................................. 377
- DL_HP_NOTIFY_EVENT_REQ Primitive .................................................. 379
- DL_LINK_UP_IND and DL_LINK_DOWN_IND Primitives ......................... 379

### OOP Data

- Outbound Processing ........................................................................ 383
  - Algorithm ..................................................................................... 384
- Inbound Processing .......................................................................... 384
  - Algorithm ..................................................................................... 384
- STREAMS Modules ........................................................................... 384
- Packet Trains ................................................................................... 385

## 16 Supporting the HP SMH NIC Tool in LAN Drivers

- NIC Tool Overview ........................................................................... 387
- How the NIC Tool Driver Shared Library Works ......................... 388
- Creating the NIC Tool Driver Shared Library ..................................... 393
  - Requirements for the Driver to Work With the NIC Tool ............. 393
  - Developing the Shared Library .................................................. 393
  - Building the Shared Library ...................................................... 395
  - Installing the Shared Library ..................................................... 395

## 17 Mass Storage Stack Architecture

- Mass Storage Stack Features .............................................................. 397
- Overview of the HP-UX Mass Storage I/O Stack ............................... 397
- Major Data Structures ...................................................................... 400
- I/O Flow .......................................................................................... 402
- SCSI Addressing Paradigm ............................................................... 402
- Probe Functions ............................................................................. 403
  - SCSI Class Driver .................................................................... 403
  - SCSI Services ............................................................................ 403
  - SCSI Interface Driver .................................................................. 403
- Interface Driver Migration to HP-UX 11i v3 .................................. 404

## 18 Writing a SCSI Interface Driver

- External Interfaces to a SCSI Interface Driver .................................. 405
  - Entry Points ............................................................................... 405
    - Mandatory Entry Points ......................................................... 405
    - Conditional Entry Points ....................................................... 406
  - Service Functions and Macros .................................................... 406
    - Registration and Probe Services ............................................ 406
    - Asynchronous Event Services ............................................... 407
    - Interface Handle Lookup Services ....................................... 407
    - Tag Management and Allocation Services ............................. 407
    - DMA Mapping Services ....................................................... 408
    - Tracing Services .................................................................... 409
    - SCSI Command Services ..................................................... 410
    - Linked List Services ............................................................. 410
  - External Data Structures ............................................................... 411
    - Controller/Target Registration .............................................. 411
    - Lunpath Open/Close .............................................................. 412
19 Device Special File Support in Administration Tools.................................445

Overview of *sf Commands............................................................................445
The lssf Command............................................................................................445
Examples............................................................................................................445
The insf Command.............................................................................................445
Examples............................................................................................................445
The mksf Command..........................................................................................446
Examples............................................................................................................446
The rmsf Command..........................................................................................446
Examples............................................................................................................446
Command Flow.................................................................................................446
Shared Library Creation.................................................................447
Shared Library Name and Location..............................................448
Example..................................................................................448
Shared Library Supported Interfaces........................................449
Compiling a Shared Library.......................................................449
Sample Code..........................................................................450

20 OL* and PCI Error Recovery Support in Interface Drivers........455
OL* Support Requirements for Interface Drivers.......................455
Driver Requirements for OL* Support........................................455
WSIO Interfaces........................................................................456
WSIO Generic Event Structure..................................................456
WSIO Callback Function..........................................................457
Registering an Event Handler....................................................458
Registering an Event Mask.........................................................458
Writing the Event Handler........................................................460
Configuring Event Timeout Values..........................................461
Miscellaneous Required Driver Changes.................................461
Managing OL* Operations in the Driver......................................462
   Online Addition....................................................................462
   MP Safe...............................................................................462
   Resource Allocation Failures..............................................462
   Online Replacement..........................................................463
   Suspend..............................................................................463
   Resume...............................................................................465
   Online Deletion....................................................................466
   Pred delete Phase...............................................................466
   Post delete Phase..............................................................466
   PCI Card Online Deletion Flow..........................................466
   PCI Card Online Deletion Failure Scenarios........................468
OL* Scripts..............................................................................469
PCI Error Recovery Support Requirements for Interface Drivers...469
Overview..............................................................................470
PCI Error Recovery Capability.................................................470
Registering Event Handler and PCI Error Recovery Capability....471
Detecting a PCI Error Condition..............................................472
Reporting a PCI Bus Error.........................................................473
Automatic PCI Error Recovery Stages......................................474
Manual PCI Error Recovery....................................................475

21 Critical Resource Analysis....................................................477
CRA Framework........................................................................477
Infrastructure CRA Module......................................................477
Subsystem CRA Modules........................................................477
Module Interaction....................................................................477
CRA Interface..........................................................................477
Infrastructure CRA Call Semantics..........................................478
Subsystem CRA Interface........................................................478
Common CRA Context Parameters.........................................480
CRA Return Flags....................................................................480
CRA Header File and Library Details........................................481
Input Parsing Guidelines........................................................481
Subsystem CRA Modules and Execution Order........................482
CRA Log File .................................................................................................................. 482
SD Package Components ................................................................................................. 514
SD Structure Capabilities ................................................................................................. 512
Software Depot Overview ................................................................................................. 511
Assumptions and Dependencies ....................................................................................... 508
Interrupt Migration Event Masks and Registration with WSIO ......................................... 497
LBI Event Flags and Migration Algorithm ......................................................................... 497
TBI Event Flags and Migration Algorithm ......................................................................... 498
MSI and MSI-X Event Flags and Migration Algorithm ..................................................... 502
LBI Event Flags and Migration Algorithm ......................................................................... 497
WSIO_EVENT_LBI_INTR_MIGR .................................................................................... 497
TBI Event Flags and Migration Algorithm ......................................................................... 498
WSIO_EVENT_OFFLINE_CPU ....................................................................................... 499
MSI and MSI-X Event Flags and Migration Algorithm ..................................................... 502
WSIO_EVENT_OFFLINE_CPU ....................................................................................... 503
Subsystem CRA Locked Analysis ...................................................................................... 485
Performing CRA on a 4-Port LAN Card ........................................................................... 485
CRA Flow Examples ......................................................................................................... 487
PCI Card Delete - CRA Successful .................................................................................. 487
PCI Card Delete - CRA Detects Data Critical Resources .................................................. 488
PCI Card Delete - Internal Failure During Analysis ......................................................... 490
PCI Card Delete - Subsystem CRA Module Fails RELEASE Event ..................................... 492

22 Interrupt Migration ...................................................................................................... 495
Impact on Drivers ........................................................................................................... 495
LBI Drivers ..................................................................................................................... 495
TBI Drivers ..................................................................................................................... 495
MSI and MSI-X Drivers ................................................................................................. 495
CLM Services ................................................................................................................ 496
Interrupt Line Sharing .................................................................................................... 496
Interrupt Migration Event Masks and Registration with WSIO ......................................... 497
LBI Event Flags and Migration Algorithm ......................................................................... 497
WSIO_EVENT_LBI_INTR_MIGR .................................................................................... 497
TBI Event Flags and Migration Algorithm ......................................................................... 498
WSIO_EVENT_OFFLINE_CPU ....................................................................................... 499
WSIO_EVENT_ONLINE_CPU ....................................................................................... 501
MSI and MSI-X Event Flags and Migration Algorithm ..................................................... 502
WSIO_EVENT_OFFLINE_CPU ....................................................................................... 503
WSIO_EVENT_MSI_INTR_MIGR ............................................................................... 506
WSIO_EVENT_ONLINE_CPU ....................................................................................... 507
Interrupt Migration Flow .................................................................................................. 507
Assumptions and Dependencies ....................................................................................... 508

23 Creating a Software Depot .......................................................................................... 511
Software Depot Overview ................................................................................................. 511
Bundles ........................................................................................................................... 512
Products .......................................................................................................................... 512
Subproducts ..................................................................................................................... 512
Filesets ............................................................................................................................ 512
SD Structure Capabilities ................................................................................................. 512
Software Objects Nomenclature ....................................................................................... 513
SD Package Components ................................................................................................. 514
Product Specification File .................................................................................................. 515
SD Objects Attributes Classification and Flow ..................................................................... 516
Policies of the PSF Attributes ......................................................................................... 517
Vendor Attributes ........................................................................................................... 517
Category Attributes ........................................................................................................ 517
SD-Bundle Attributes ....................................................................................................... 522
SD-Product Attributes ..................................................................................................... 526
SD-Subproduct Attributes ............................................................................................... 530
SD-Fileset Attributes ...................................................................................................... 531
File Definitions ............................................................................................................... 535

Table of Contents 17
Control Scripts Overview...........................................................................................................536
Control Script Levels................................................................................................................537
Special Types of Filesets............................................................................................................537
  Reboot Fileset..........................................................................................................................537
  Kernel Fileset............................................................................................................................537
  Prerequisite Fileset....................................................................................................................537
  Corequisite Fileset.....................................................................................................................537
Control Script Format................................................................................................................537
Execution Environment.............................................................................................................538
  Location of the File System......................................................................................................538
  Environment Variables............................................................................................................538
Relocating the Product................................................................................................................539
System Commands....................................................................................................................540
  SW-DIST Commands.................................................................................................................541
General Control Script Actions.................................................................................................541
  Input.........................................................................................................................................541
  Output......................................................................................................................................541
  Exit Values...............................................................................................................................542
  Required Actions.....................................................................................................................542
  control_utils............................................................................................................................543
Configuration Directory and Functions.....................................................................................543
Control Script Guidelines..........................................................................................................543
  Checkinstall Guidelines..........................................................................................................543
  Preinstall Guidelines.................................................................................................................545
  Postinstall Guidelines..............................................................................................................546
  Configure Guidelines.............................................................................................................547
  Unconfigure Guidelines..........................................................................................................549
  Preremove Guidelines............................................................................................................550
  Postremove Guidelines............................................................................................................550
Files to Package........................................................................................................................551
Creating a Software Depot.........................................................................................................551
  Step 1: Designing an SD Structure.........................................................................................552
  Step 2: Selecting the Product Directory Structure...............................................................552
  Step 3: Writing a PSF...............................................................................................................552
    Writing a PSF Vendor Object...............................................................................................552
    Writing a PSF Category Object.............................................................................................553
    Writing a PSF Bundle Object...............................................................................................553
    Writing a PSF Product Object...............................................................................................554
    Writing a PSF Subproduct Object........................................................................................555
    Writing a PSF Fileset Object...............................................................................................556
  Step 4: Writing Control Scripts.............................................................................................557
    Writing a Checkinstall Script...............................................................................................557
    Writing a Preinstall Script.....................................................................................................558
    Writing a Postinstall Script.................................................................................................559
    Writing a Configure Script.................................................................................................560
    Writing an Unconfigure Script............................................................................................561
    Writing a Preremove Script.................................................................................................562
    Writing a Postremove Script...............................................................................................563
  Step 5: Packaging the Components.......................................................................................564
    SD Components Packaging...................................................................................................564
    The swpackage Process.........................................................................................................565
  Step 6: Registering the Depot..................................................................................................566
Installing a Depot......................................................................................................................566
  Using swinstall.......................................................................................................................566
  Step 1: Selecting the Source Depot.......................................................................................566
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>GIO-CDIO Organization</td>
<td>30</td>
</tr>
<tr>
<td>1-2</td>
<td>Same Driver in Different Configurations</td>
<td>31</td>
</tr>
<tr>
<td>1-3</td>
<td>WSIO CDIO as a Buffer Zone</td>
<td>32</td>
</tr>
<tr>
<td>2-1</td>
<td>Noncoherent System</td>
<td>45</td>
</tr>
<tr>
<td>2-2</td>
<td>Physical and I/O Virtual Addressing</td>
<td>47</td>
</tr>
<tr>
<td>3-1</td>
<td>Synchronization Areas</td>
<td>69</td>
</tr>
<tr>
<td>3-2</td>
<td>Synchronization Mechanism</td>
<td>70</td>
</tr>
<tr>
<td>7-1</td>
<td>Sample KRS Tree Structure</td>
<td>174</td>
</tr>
<tr>
<td>7-2</td>
<td>KRS Node Structure</td>
<td>175</td>
</tr>
<tr>
<td>7-3</td>
<td>Linked Nodes</td>
<td>175</td>
</tr>
<tr>
<td>7-4</td>
<td>Relationship between Linked Nodes and Their Descendants</td>
<td>176</td>
</tr>
<tr>
<td>10-1</td>
<td>HP-UX Network Interface Driver Architecture</td>
<td>214</td>
</tr>
<tr>
<td>10-2</td>
<td>Data Link Layer Architecture for a Tightly Coupled Driver</td>
<td>218</td>
</tr>
<tr>
<td>10-3</td>
<td>Data Link Layer Architecture for Loosely Coupled Drivers</td>
<td>219</td>
</tr>
<tr>
<td>11-1</td>
<td>HP-UX Architecture for Non-Native Drivers</td>
<td>222</td>
</tr>
<tr>
<td>11-2</td>
<td>Developing a Non-Native Networking Driver — Part 1</td>
<td>224</td>
</tr>
<tr>
<td>11-3</td>
<td>Developing a Non-Native Networking Driver — Part 2</td>
<td>225</td>
</tr>
<tr>
<td>11-4</td>
<td>Process Flow of a Control Request</td>
<td>250</td>
</tr>
<tr>
<td>11-5</td>
<td>The DL_GET_STATISTICS_REQ MBLK Chain</td>
<td>252</td>
</tr>
<tr>
<td>11-6</td>
<td>MBLK Chain for an Ioclt</td>
<td>254</td>
</tr>
<tr>
<td>12-1</td>
<td>Developing a Networking Driver</td>
<td>294</td>
</tr>
<tr>
<td>12-2</td>
<td>STREAMS DLPI Network Driver Sequence</td>
<td>306</td>
</tr>
<tr>
<td>12-3</td>
<td>Fastpath Negotiation Message Formats</td>
<td>313</td>
</tr>
<tr>
<td>12-4</td>
<td>Control Flowchart for Outbound Path</td>
<td>323</td>
</tr>
<tr>
<td>12-5</td>
<td>Control Flowchart for Inbound Path</td>
<td>325</td>
</tr>
<tr>
<td>13-1</td>
<td>LAN Command Flow</td>
<td>337</td>
</tr>
<tr>
<td>13-2</td>
<td>The nwmgr Command Architecture</td>
<td>338</td>
</tr>
<tr>
<td>14-1</td>
<td>Network Tracing and Logging Elements, and Data Flow</td>
<td>354</td>
</tr>
<tr>
<td>15-1</td>
<td>Fastpath Negotiation Request Message</td>
<td>378</td>
</tr>
<tr>
<td>15-2</td>
<td>Fastpath Negotiation Acknowledgement Message</td>
<td>378</td>
</tr>
<tr>
<td>15-3</td>
<td>Data Tuple</td>
<td>379</td>
</tr>
<tr>
<td>15-4</td>
<td>Sample Packet with OOP Data</td>
<td>381</td>
</tr>
<tr>
<td>15-5</td>
<td>Fragment Chain</td>
<td>385</td>
</tr>
<tr>
<td>16-1</td>
<td>HP-UX Network Interfaces Configuration Tool</td>
<td>388</td>
</tr>
<tr>
<td>16-2</td>
<td>SMH NIC Tool Architecture</td>
<td>388</td>
</tr>
<tr>
<td>16-3</td>
<td>Flow Diagram Showing How the Driver Shared Library Works</td>
<td>390</td>
</tr>
<tr>
<td>17-1</td>
<td>Mass Storage Stack</td>
<td>398</td>
</tr>
<tr>
<td>17-2</td>
<td>Mass Storage I/O Data Structures</td>
<td>401</td>
</tr>
<tr>
<td>17-3</td>
<td>I/O Flow Diagram</td>
<td>402</td>
</tr>
<tr>
<td>18-1</td>
<td>Controller Registration and Probing Sequence</td>
<td>421</td>
</tr>
<tr>
<td>19-1</td>
<td>Functional Interface</td>
<td>447</td>
</tr>
<tr>
<td>21-1</td>
<td>CRA Flow During PCI Card Delete - CRA Successful</td>
<td>488</td>
</tr>
<tr>
<td>21-2</td>
<td>CRA Flow During PCI Card Delete - CRA Detects Data Critical Resources</td>
<td>490</td>
</tr>
<tr>
<td>21-3</td>
<td>CRA Flow During PCI Card Delete - Internal Failure During Analysis</td>
<td>492</td>
</tr>
<tr>
<td>21-4</td>
<td>CRA Flow During PCI Card Delete - Subsystem CRA Module Fails RELEASE Event</td>
<td>494</td>
</tr>
<tr>
<td>22-1</td>
<td>LBI Interrupt Migration</td>
<td>498</td>
</tr>
<tr>
<td>22-2</td>
<td>TBI Interrupt Migration</td>
<td>500</td>
</tr>
<tr>
<td>22-3</td>
<td>MSI and MSI-X Interrupt Migration</td>
<td>504</td>
</tr>
<tr>
<td>23-1</td>
<td>SD Structure</td>
<td>513</td>
</tr>
<tr>
<td>23-2</td>
<td>SD Package</td>
<td>516</td>
</tr>
</tbody>
</table>
List of Tables

1  Publishing History Details......................................................................................................................25
2  HP-UX 11i Releases.................................................................................................................................27
2-1 Interrupt Service Routines......................................................................................................................41
2-2 Memory Allocation Service Routines........................................................................................................43
2-3 Register Service Routines.........................................................................................................................44
2-4 Configuration Space Service Routines......................................................................................................44
2-5 I/O Port Space Service Routines..............................................................................................................44
2-6 DMA Service Routines............................................................................................................................48
2-7 Endian Service Routines...........................................................................................................................50
2-8 Driver Event Handling Service Routines..................................................................................................51
2-9 Description Service Routine.....................................................................................................................52
2-10 Ordered Interrupts Service Routine.......................................................................................................52
2-11 System Attribute Service Routines.......................................................................................................52
2-12 I/O Synchronization Service Routine...................................................................................................52
2-13 Interface Property Flags (Create)............................................................................................................55
2-14 Interface Property Flags (Modify)............................................................................................................56
2-15 Kernel I/O Mapping Interfaces..............................................................................................................64
4-1 drv_ops_t Structure Field Descriptions................................................................................................79
4-2 drv_info_t Structure Field Descriptions................................................................................................81
4-3 wsio_drv_data_t Structure Field Descriptions....................................................................................81
4-4 Configuration and Entry Point Routines..................................................................................................85
4-5 Entry Points for a Block Driver................................................................................................................87
4-6 Entry Points for a Character Driver.........................................................................................................87
4-7 Device Probe Parameters.........................................................................................................................94
4-8 Address Probe Parameters.......................................................................................................................95
6-1 Module State Change..............................................................................................................................159
6-2 Sample Makefile.bld Compiler Options................................................................................................169
7-1 KRS Value Types..................................................................................................................................174
7-2 Attribute Flags......................................................................................................................................177
7-3 Action Flags..........................................................................................................................................177
11-1 Data Structures for HP-DLPI Properties..............................................................................................235
11-2 Required DLPI Primitives and Iocltls...................................................................................................252
11-3 Request Data Structures Associated with Primitives.........................................................................253
11-4 Response Data Structures of Primitives.............................................................................................254
11-5 Events Sent from HP-DLPI to Driver....................................................................................................277
11-6 Description of the Fields in mib_ifEntry.............................................................................................278
12-1 Packet Type, Protocol Kind, and Protocol Value.................................................................................304
12-2 Promiscuous Level and Mode.................................................................................................................305
12-3 enet_dlpi_data_t Structure......................................................................................................................309
12-4 Message Service Functions...................................................................................................................311
12-5 Options Negotiations Structure (driver_ops_t).....................................................................................314
12-6 Enumerated Type..................................................................................................................................314
12-7 DLPI Primitives and Iocltls....................................................................................................................317
12-8 enet_ift Data Structure Fields..............................................................................................................320
12-9 Bound SAP Data Fields........................................................................................................................321
12-10 Fields in mib_ifEntry Data Structure................................................................................................329
13-1 General Purpose and Data Retrieval Services....................................................................................341
13-2 General Display Services.....................................................................................................................345
13-3 Online Configuration Services............................................................................................................347
13-4 Command Validation Services..............................................................................................................348
13-5 Configuration File Services..................................................................................................................348
13-6 Diagnostic Services..............................................................................................................................350
## List of Examples

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>Fixed-Sized Arena with Default Attributes</td>
<td>60</td>
</tr>
<tr>
<td>2-2</td>
<td>Overriding the Default Arena Attributes</td>
<td>60</td>
</tr>
<tr>
<td>2-3</td>
<td>Variable-Sized Arena with Default Attributes</td>
<td>61</td>
</tr>
<tr>
<td>4-1</td>
<td>Register Address Probe Example</td>
<td>97</td>
</tr>
<tr>
<td>4-2</td>
<td>scsi_probe Routine Example</td>
<td>100</td>
</tr>
<tr>
<td>4-3</td>
<td>Sample driver_read Routine Using physio()</td>
<td>109</td>
</tr>
<tr>
<td>4-4</td>
<td>driver_minphys Routine Example</td>
<td>117</td>
</tr>
<tr>
<td>4-5</td>
<td>Sample driver_select Routine</td>
<td>119</td>
</tr>
<tr>
<td>4-6</td>
<td>Sample driver_install Routine</td>
<td>132</td>
</tr>
<tr>
<td>11-1</td>
<td>ISR for the IELAN Sample Driver</td>
<td>263</td>
</tr>
</tbody>
</table>
This manual provides information on how to write a kernel driver module for HP-UX 11i v3. It is applicable for both Itanium® and PA-RISC platforms.

The document printing date and part number indicate the document’s current edition. The printing date will change when a new edition is printed. Minor changes may be made at reprint without changing the printing date. The document part number will change when extensive changes are made.

Document updates may be issued between editions to correct errors or document product changes. To ensure that you receive the updated or new editions, you should subscribe to the appropriate product support service. See your HP sales representative for details.

The latest version of this document can be found on line at:
http://www.hp.com/go/hpux_ddk

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.

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.

Publishing History

Table 1 Publishing History Details

<table>
<thead>
<tr>
<th>Document Manufacturing Part Number</th>
<th>Operating Systems Supported</th>
<th>Supported Product Versions</th>
<th>Publication Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>5990-8504</td>
<td>11i v2</td>
<td></td>
<td>September 2004</td>
</tr>
<tr>
<td>5991-7948</td>
<td>11i v3</td>
<td></td>
<td>February 2007</td>
</tr>
</tbody>
</table>

Document Organization

The HP-UX 11i v3 Driver Development Guide (DDG) is divided into several chapters, and each contains information about installing or configuring driver:

Chapter 1 (page 29) Summarizes what is in this document and provides an overview of the HP-UX driver environment.

Chapter 2 (page 33) Explains software concepts associated with the HP-UX I/O system.

Chapter 3 (page 69) Covers the kernel services that handle synchronization used by drivers on multiprocessor systems.
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page No.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>77</td>
<td>Logical modular sequence and code necessary to write a device driver.</td>
</tr>
<tr>
<td>5</td>
<td>137</td>
<td>Describes managing and developing Dynamically Loadable Kernel Modules.</td>
</tr>
<tr>
<td>6</td>
<td>157</td>
<td>Shows information and models on how to write a kernel module.</td>
</tr>
<tr>
<td>8</td>
<td>183</td>
<td>The integer variable that controls the behavior of the module.</td>
</tr>
<tr>
<td>9</td>
<td>197</td>
<td>Presents routines and concepts specifically for drivers of Peripheral Component Interconnect (PCI) drivers.</td>
</tr>
<tr>
<td>10</td>
<td>213</td>
<td>Explains designing and writing a device driver for Network Interface Cards (NICs).</td>
</tr>
<tr>
<td>11</td>
<td>221</td>
<td>Provides information on designing and developing a LAN driver under HP-DLPI.</td>
</tr>
<tr>
<td>12</td>
<td>293</td>
<td>Provides information on designing and developing a Native STREAMS DLPI network driver.</td>
</tr>
<tr>
<td>13</td>
<td>337</td>
<td>Provides the user with the ability to scan and administer LAN interfaces on an HP-UX system.</td>
</tr>
<tr>
<td>14</td>
<td>353</td>
<td>To aid in troubleshooting network problems.</td>
</tr>
<tr>
<td>15</td>
<td>377</td>
<td>Interaction between transport layer and DLS providers.</td>
</tr>
<tr>
<td>16</td>
<td>387</td>
<td>Provides both GUI and TUI based interface to configure system resources.</td>
</tr>
<tr>
<td>17</td>
<td>397</td>
<td>Includes and overview of the HP-UX Mass Storage I/O Stack and SCSI Addressing Paradigm.</td>
</tr>
<tr>
<td>18</td>
<td>405</td>
<td>Describes how to design and develop a SCSI interface driver.</td>
</tr>
<tr>
<td>19</td>
<td>445</td>
<td>Explains support implementation for the *sf commands.</td>
</tr>
<tr>
<td>20</td>
<td>455</td>
<td>Required feature for high availability servers.</td>
</tr>
<tr>
<td>22</td>
<td>495</td>
<td>Describes a flexible mechanism for managing CPU interrupt assignments.</td>
</tr>
<tr>
<td>23</td>
<td>511</td>
<td>Describes how to distribute software on HP-UX both locally and on remote systems.</td>
</tr>
<tr>
<td>Glossary</td>
<td>575</td>
<td>Provides a comprehensive list of terms commonly used in the HP-UX Device Driver Reference Guide and HP-UX Device Driver Guide.</td>
</tr>
</tbody>
</table>

**Typographic 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)` for more information.

- **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.
HP-UX Release Name and Release Identifier

Each HP-UX 11i release has an associated release name and release identifier. The `uname` command with the `-r` option returns the release identifier. This table shows the releases available for HP-UX 11i.

### Table 2 HP-UX 11i Releases

<table>
<thead>
<tr>
<th>Release Identifier</th>
<th>Release Name</th>
<th>Supported Processor Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.11.23</td>
<td>HP-UX 11i v2 September 2004</td>
<td>PA-RISC, Intel Itanium®</td>
</tr>
<tr>
<td>B.11.31.02</td>
<td>HP-UX 11i v3 February 2007</td>
<td>PA RISC and Intel® Itanium®</td>
</tr>
</tbody>
</table>

Related Documents

Additional information about the DDG can be found at:


Other documents in the DDK collection include:

- **DDK FAQ**
- **HP-UX 11i v3 Driver Development Getting Started Guide**
- **HP-UX 11i v3 Driver Development Reference Guide**

HP Encourages Your Comments

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

Please send comments to:

feedback@fc.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 and Internet Resources

HPHP program and developer resource materials are available at the following location:

- HPHP program E-mail at:
  [hphp.support@hp.com](mailto:hphp.support@hp.com)
- HP hardware provider program at:
Support/Compatibility Disclaimers

Because drivers function at the level of the kernel, 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.
- If 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.
1 Driver Environment Overview

This chapter describes how the HP-UX I/O subsystem is structured, and how the driver fits into this environment. This chapter addresses the following topics:

- “I/O Subsystem Structure”
- “General I/O System”
- “Context Dependent I/O Modules (CDIOs)” (page 30)

I/O Subsystem Structure

The I/O subsystem provides a uniform interface for user processes to use to read information from and write information to devices, and it provides 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 they use 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) are also important.

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

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

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

- 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 the following:
  - 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 is independent of individual driver views.

Figure 1-1 shows the flow of information through the GIO.
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 enables individual component CDIOs to be configured in or out of a system as needed.

There are two categories of CDIOs:

- **Bus-nexus CDIOs**
  
  CDIOs that communicate directly with a bus. Provide bus-dependent services to other CDIOs. These can have bus-nexus drivers that control bus adapters or bus converters. PCI CDIO is an example of a bus-nexus CDIO.

- **Driver Environment CDIOs**
  
  Provide drivers with a defined environment. Drivers within a CDIO environment share a common set of services and entry points. Workstation CDIO (WSIO CDIO) is an example of a Driver Environment CDIO.

Basic Components of a CDIO

A CDIO has the following components:

- **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 enable 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. A service in one CDIO can be called by a driver in another CDIO. For example, some PCI card drivers are part of the WSIO CDIO, but call bus-dependent functions from the PCI 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 PCI cards do not always require the PCI CDIO (if there is no PCI hardware in the system). For example, some drivers can control cards in either a CORE CDIO or PCI CDIO environment. The WSIO CDIO driver environment helps to hide the bus-specific services, so that both PCI and CORE do not need to be used every time the driver is configured.

- Drivers
  In most cases, a CDIO contains drivers. In a bus-nexus CDIO like PCI, the driver is the PCI bus-nexus manager that configures the PCI adapter and that provides services specific to PCI. A Driver Environment CDIO like WSIO can support many drivers. In most cases, the drivers written 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 is designed for the workstation single-processor environment and the server multiprocessor environment. Because the WSIO CDIO is a Driver Environment CDIO, it provides a consistent environment no matter how it is configured 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 PCI CDIO, or with the system I/O bus. To the driver, all configurations look the same. The WSIO CDIO must interpret the service calls and take the appropriate actions for the given configuration.

Figure 1-2 shows the same driver in different configurations.

Figure 1-2 Same Driver in Different Configurations

The WSIO CDIO also acts as a buffer zone that protects its drivers from the peculiarities of the bus-nexus CDIO that it is configured with. The WSIO CDIO masks and hides all interface differences from the driver, handles configuration issues, and monitors resources.

See the HP-UX 11i v3 Driver Development Reference for the reference pages for WSIO CDIO routines, services, and data structures.

Figure 1-3 illustrates the buffer zone concept.
Figure 1-3 WSIO CDIO as a Buffer Zone
This chapter describes software concepts associated with the HP-UX I/O subsystem. It also provides information and models about how to write a driver. The topics described in this chapter form the foundation of most kernel drivers. This chapter addresses the following topics:

- “Driver Types” (page 33)
- “Major and Minor Numbers” (page 34)
- “System Calls” (page 36)
- “Kernel Data Structures” (page 38)
- “Timeout Mechanisms” (page 40)
- “Interrupt Handling” (page 41)
- “Memory Allocation and Mapping I/O Space” (page 43)
- “Cache Coherence” (page 45)
- “DMA Mapping” (page 46)
- “WSIO Interface Registration Services” (page 53)

### Driver Types

WSIO supports three types of drivers:

- **Device**
  - Manages peripheral devices such as disk and 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 (for example, SCSI disk drives).

- **Pseudo**
  - Not associated with a particular device. In many cases, it is a preprocessor for a device driver. Examples include the Logical Volume Manager (LVM), the `pty` driver for logical terminals, and certain types of pipes. Pseudo drivers are accessed from user and system programs.

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

A **monolithic driver** is a combined device and interface driver. Use a monolithic driver in the following instances:

- When a particular type of device is always connected to a particular type of interface card. For example, 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 Devices

Block devices are structured devices, such as a disk or magnetic tape, that have file system support and are random access devices. A block device driver enables buffered, random access through the file system buffer cache to the device. Use block device drivers for disks and any other devices on which a file system can reside.

With file system support, HP-UX transfers data between a user process and a block device in blocks of size BLKDEV_IOSIZE (2048 bytes).

Use the disk drives for data caching because the system benefits from caching file system data. The following system calls provide more information:

read  In data caching, the kernel first checks the file system 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 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. After determining that the data in a buffer must be copied, the kernel calls the appropriate block device driver and enables it to perform asynchronous, buffered I/O.

Character Devices

Character devices (raw devices) have no file system support and are typically sequential-access devices that process raw I/O. A character device driver does not access a device in blocks.

With no file system support, the kernel does not cache data for character devices. After accessing a device as a character device, data is transmitted in units of one or more bytes in one of the following raw I/O methods:

- Unbuffered (direct)
  Use the physio routine for unbuffered data transfer.

- Buffered
  Use the copyin or uiomove routine to transfer data from the character device driver's own buffer. This is useful for small amounts of data or to control the data rate.

Major and Minor Numbers

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 (for example, SCSI disk drivers or optical auto-changers) have both a block major number and a character major number. Devices that support character mode access only have a character major number only.

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 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) that uses the file system buffer cache, and one for character devices (cdevsw) that 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

Drivers must request dynamic major numbers. To have the system dynamically assign a major number to the driver, follow these steps:
1. Specify -1 in both the \texttt{b\_major} and \texttt{c\_major} fields of the \texttt{drv\_info\_t} structure in the driver’s header. See “Step 3: Defining Installation Structures” (page 78) for more information about the \texttt{drv\_info\_t} structure.

2. In the \texttt{drv\_info\_t} structure, set the following bit values in the \texttt{flags} field:
   \begin{itemize}
     \item If a block driver, set the \texttt{DRV\_BLOCK} bit.
     \item If a character driver, set the \texttt{DRV\_CHAR} bit.
     \item If the driver is both a block and a character driver, set both bits.
   \end{itemize}

After you build and boot a kernel containing the driver, find the major number that has been dynamically assigned by using the \texttt{lsdev} command. This command reads the information provided by the driver header and retrieves the major number. Major numbers are displayed in decimal form. For an example of a dynamic way of extracting the major number from a standard HP-UX 11i driver, see \texttt{lsdev}(1M).

Using Minor Numbers

Minor numbers contain 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, see the manpages in the \textit{HP-UX 11i v3 Driver Development Reference}.

Device Special Files

To create a device-special file for the driver, see \texttt{mknod}(1M) in the \textit{HP-UX 11i v3 Driver Development Reference}. A long listing (\texttt{ls -l}) of a typical device special file is similar to the following:

\begin{verbatim}
crw------- 2 bin 193 0x00080 Jul 12 02:19 mux0
\end{verbatim}

The two important fields 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 the \texttt{dev\_t} format. This format consists of the following:

\begin{itemize}
  \item \textbf{Bits 0-7} The major number, which can range from 0 to 255, inclusive. Character and block major numbers are separate ranges.
  \item \textbf{Bits 8-31} The minor number. The conventional notation for the minor number follows the format 0x\texttt{hhhhhh}, where \texttt{h} is a four-bit hexadecimal digit. Bits 8-15 usually encode the instance number of the interface card. It represents the order in which HP-UX encounters the interface card within a class when binding it into the system, and is displayed in decimal notation in the \texttt{Instance (I)} column of \texttt{ioscan -f} output.
  \item \textbf{Bits 16-31} encode device and driver dependent characteristics. These can include special rules for the following:
    \begin{itemize}
      \item \texttt{Tapes} Rewind-on-close, density
      \item \texttt{Printers} All caps
      \item \texttt{Disks} Section number, unit number
    \end{itemize}
\end{itemize}

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 major number as an index into a device switch table.

The kernel maintains two device switch tables:
\begin{itemize}
  \item \texttt{bdevsw} For block devices
  \item \texttt{cdevsw} For character devices
\end{itemize}

Each block device driver has an entry in the \texttt{bdevsw} table. Each character device driver has an entry in the \texttt{cdevsw} table.
The kernel automatically uses the driver installation routine, entries in the driver `drv_ops_t` and `drv_info_t` structures, and information from the driver module metadata file to construct the device switch tables. For more information, see “Step 3: Defining Installation Structures” (page 78) and “Step 5: Writing Configuration Routines” (page 87).

For additional information about the module metadata files, see Chapter 6 (page 157).

System Privileges

The UNIX® operating system has traditionally used a privilege model where superusers (those with effective UID 0, such as the root user) have virtually unlimited power and other users have few or no special privileges.

The HP-UX fine-grained privilege model splits the powers of root users into a set of privileges. Each privilege grants a process that possesses the privilege the right to a certain set of restricted services provided by the kernel.

A process can manage privileges internally with privilege bracketing. Privilege bracketing enables (or raises) a privilege only while the privilege is needed, then disables (or lowers) the privilege when it is not needed. The privileges that a process raises determines the sensitive system call services that the process can invoke.

The HP-UX kernel exports the `priv_scall` and `priv_policy` utility functions. Kernel components can use these functions to determine whether a privilege is enabled in a process or credential. For more information, see `priv_scall(9F)` and `priv_policy(9F)` in the HP-UX 11i v3 Driver Development Reference.

System Calls

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

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

The system calls that perform I/O include:

- `close(2)`
- `ioctl(2)`
- `open(2)`
- `read(2)`
- `select(2)`
- `write(2)`

When a user process issues an I/O system call on a device special file, the corresponding driver routine is called. Before the routine is activated, the kernel takes the following actions:

1. Checks that the user has permission to access the device.
2. Obtains system buffers to use, if necessary.
3. Uses the major number to index into a device switch table.
4. Calls the driver associated with the device file with the appropriate parameters.

Parts of a Driver

The device driver’s entry point routines constitute the upper half of the driver and the user context. A system call from a user program activates the upper half. The lower half of the driver, the interrupt context, processes interrupts from the device. Each half works as follows:

1. The upper half initiates activity on the device, then waits.
2. The device completes the activity and interrupts. The lower half of the driver informs the upper half that it can continue.
Interrupts are handled by an **Interrupt Service Routine** (ISR) and supporting routines in the interface driver. For a description of interrupt services, see “Interrupt Handling” (page 41). For information on writing the driver ISR, see Chapter 4 (page 77).

**I/O Request Flow**

When writing a driver, you must understand how the kernel and the driver interact as they process an I/O request. This section describes how I/O is accomplished.

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

1. A user process makes an I/O system call and invokes the kernel through the system call interface. The kernel manages the process and resources for the request.

2. The kernel gets the 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.

3. The kernel calls the device driver.
   
   The I/O system call obtains the device driver entry point from the device switch table entry, and passes request-generated control and parameter information to the device driver.

4. The device driver initializes data structures and sets 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. This depends on the characteristics of the device and the needs of the driver.
   
   A device driver routine waits by calling `sleep`, which puts the user process to sleep until a corresponding call to `wakeup` is issued by another routine.

5. The interface driver processes the request 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.

6. The device driver completes the request.
   
   If the device driver called `sleep` for the duration of the data transfer, when transfer is complete the device interrupt routine uses `wakeup` to awaken the process. The device driver resumes execution from where it put itself to sleep, completes the call, and returns an integer status value (indicating the success or failure of the request) to the kernel routine that invoked it.

7. 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.

**I/O Read Request Sample**

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, you might need 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 performs the following steps:
   
   a. Preprocesses the request, such as verifying the file is a character device or 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 \textit{driver\_strategy} routine uses the minor number to access the correct device, and decodes the device options.

5. The \textit{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 \textit{driver\_strategy} routine returns and \texttt{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 the read request.

10. The hardware completes the read request and interrupts the system.

11. The interface driver processes the interrupt, waking up the device driver upper half.

12. Control returns to the \textit{driver\_read} routine, which completes any final device-specific processing of the request.

13. Control returns to the \texttt{read} system call, which completes the request and returns control to the user process.

**Kernel Data Structures**

This section describes the following kernel data structures commonly used by the I/O subsystem:

- \texttt{buf}
- \texttt{isc\_table\_type} (interface select code table)
- \texttt{ifsw} and \texttt{gfsw} (I/O switch tables)
- \texttt{uio}
- \texttt{iovec}
- \texttt{io\_events\_t}
- \texttt{io\_post\_event\_t}
- \texttt{wsio\_reg\_node\_t}
- \texttt{wsio\_legacy\_info\_t}

**buf Structure**

The \texttt{buf} structure is a block I/O buffer header 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 and I/O systems. The information in this structure specifies the buffer and the operations that can be performed on it.

When the kernel invokes the \textit{driver\_strategy} routine, it passes a pointer to a \texttt{buf} structure as a parameter. The \textit{driver\_strategy} routine schedules the transfer of data into or out of the buffer allocated to the \texttt{buf} structure.

For \texttt{buf} structure field descriptions, see \texttt{buf(9S)} in the HP-UX 11i v3 Driver Development Reference.

**isc\_table\_type Structure**

Each instance of an interface card has an ISC entry that the system maintains in an internal table. WSIO uses each ISC entry, defined as an \texttt{isc\_table\_type} structure, to maintain interface driver information.

An interface driver obtains information specific to each instance of its interface card by referencing the appropriate ISC entry.

A device driver calls the \texttt{wsio\_get\_isc} service to obtain a pointer to the ISC entry for its corresponding interface driver by giving its \texttt{dev\_t} number. It passes that \texttt{isc} pointer to the interface driver in an \texttt{ifsw} function call.
For field descriptions of the `isc_table_type` structure, see `isc_table_type(9S)` in the HP-UX 11i v3 Driver Development Reference.

I/O Switch Tables

The I/O system supports two I/O interface switch tables through fields in the ISC structure. The `isc->gfsw` path is used by the system. The `isc->ifsw` path is used for communication between one or more device drivers and one or more interface drivers. The fields in both tables 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 used only for system to interface driver communication. It consists of pointers to the following routines:

- `isc->gfsw.init` Points to a driver-defined interface initialization routine that is run by the system during the boot process. For more information, see Chapter 4 (page 77).
- `isc->gfsw.diag` Points to a driver-defined interface diagnostic routine with undefined usage.

Interface Function Switch

The interface function switch, `isc->ifsw`, is used for device driver to interface driver communication. A device driver uses this path to call the functions of an interface driver. It consists of an address pointer that is set to a defined structure of interface functions and other relevant flags and data that make up the interface driver.

A principal use of the interface function switch is for one or more device drivers to work 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 enter the addresses of the structures in their respective `isc->ifsw` fields.

The device driver obtains the ISC structure for the appropriate interface driver and calls the interface driver through the interface switch table to perform the operation. This enables the device driver to trigger interface specific routines without knowing which interface driver is configured with it. Multiple interface drivers are configured interchangeably with this device driver if they share the same `ifsw` type definition.

In a single device driver to single 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. It passes this structure to the driver's `driver_read` or `driver_write` routine.

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

Drivers of character devices seldom access individual fields in the `uio` structure. The `uiomove` and `physio` routines are used instead.

For field definitions in the `uio` structure, see `uio(9S)` in the HP-UX 11i v3 Driver Development Reference.

`iovec` Structure

The data buffer descriptor is passed in as part of the `uio` structure for character I/O and also used by the Context Dependent I/O (CDIO) mapping services. See `iovec(9S)` in the HP-UX 11i v3 Driver Development Reference for field definitions in the `iovec` structure.
io_post_event_req Structure

The io_post_event_req structure is passed in to io_post_event to post I/O infrastructure events. For field definitions, see io_post_event_req_t(9S) in the HP-UX 11i v3 Driver Development Reference.

wsio_reg_node Structure

The wsio_reg_node structure is passed in to wsio_reg_node to register or validate an I/O node. For field definition, see wsio_reg_node_t(9S) in the HP-UX 11i v3 Driver Development Reference.

wsio_legacy_info Structure

The wsio_legacy_info structure is passed in to wsio_reg_legacy to register or validate a legacy (SCSI-2) I/O node. For field definitions, see wsio_legacy_info_t(9S) in the HP-UX 11i v3 Driver Development Reference.

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 are also used by the driver to poll the status of device registers at regular intervals. This section describes how timeouts work and 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 causes a timeout to occur at priority level 5.

HP recommends using 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, the driver can unnecessarily cause interrupt servicing to be delayed. The Ktimeout routine 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 can hang while processing a request. By setting a timeout, the driver has an opportunity to recover if the device hangs.

The untimeout routine cancels a request previously made by timeout or Ktimeout. The func and arg parameters are compared to those in the called timeout. If they match, the timeout request is cancelled and its allocated resources are released.

timeout Routine

The timeout routine has the following syntax:

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

Where:

*func* Address of the function to be called when the timeout occurs.
*arg* Argument passed to *func* when *func* is called.
*t* Number of clock ticks to wait before the timeout occurs. The *t* can be expressed 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.

For more information, see `timeout`(9F) in the *HP-UX 11i v3 Driver Development Reference*. The caller must check for the return value of `timeout`.

**untimeout Routine**

The `untimeout` routine has the following syntax:

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

*func* Address of the function passed to `timeout`.
*arg* Argument passed to `timeout`.

For more information, see `untimeout`(9F) in the *HP-UX 11i v3 Driver Development Reference*.

**Interrupt Handling**

Interrupt services are included that enable drivers to allocate multiple interrupt resources. Drivers can specify one of the following interrupts:

- Line-based interrupts (LBI)
- Transaction-based interrupts (TBI)
- Message-signaled interrupts (MSI or MSI-X)

**Interrupt Services**

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

Table 2-1 shows each interface interrupt routine, its interrupt type, and its description. For parameters, return codes, and examples, see the appropriate manpage in the *HP-UX 11i v3 Driver Development Reference*.

**Table 2-1 Interrupt Service Routines**

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Interrupt Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wsio_intr_activate</code></td>
<td>TBI/LBI</td>
<td>Enables a WSIO interrupt object.</td>
</tr>
<tr>
<td><code>wsio_intr_alloc</code></td>
<td>TBI/LBI</td>
<td>Allocates a WSIO interrupt object.</td>
</tr>
<tr>
<td><code>wsio_intr_deactivate</code></td>
<td>TBI/LBI</td>
<td>Disables a WSIO interrupt object. This call blocks.</td>
</tr>
<tr>
<td><code>wsio_intr_deactivate_nowait</code></td>
<td>TBI/LBI</td>
<td>Disables a WSIO interrupt object with a callback. This call does not block.</td>
</tr>
<tr>
<td><code>wsio_intr_free</code></td>
<td>TBI/LBI</td>
<td>Frees a WSIO interrupt object.</td>
</tr>
<tr>
<td><code>wsio_intr_get_assigned_cpu</code></td>
<td>TBI/LBI</td>
<td>Gets the assigned CPU.</td>
</tr>
</tbody>
</table>
Table 2-1 Interrupt Service Routines (continued)

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Interrupt Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wsio_intr_get_irq_line</td>
<td>LBI</td>
<td>Gets the Interrupt ReQuest (IRQ) line for LBIs.</td>
</tr>
<tr>
<td>wsio_intr_get_txn_info</td>
<td>TBI</td>
<td>Gets the transaction address and data values for TBIs.</td>
</tr>
<tr>
<td>wsio_intr_set_cpu_spec</td>
<td>TBI</td>
<td>Sets up TBIs.</td>
</tr>
<tr>
<td>wsio_intr_set_irq_line</td>
<td>LBI</td>
<td>Sets up LBIs.</td>
</tr>
<tr>
<td>wsio_msi_alloc</td>
<td>MSI/MSI-X</td>
<td>Allocates an interrupt object.</td>
</tr>
<tr>
<td>wsio_msi_assign</td>
<td>MSI/MSI-X</td>
<td>Sets per-vector attributes.</td>
</tr>
<tr>
<td>wsio_msi_capability</td>
<td>MSI/MSI-X</td>
<td>Queries a PCI card for its MSI capabilities.</td>
</tr>
<tr>
<td>wsio_msi_free</td>
<td>MSI/MSI-X</td>
<td>Frees an interrupt object.</td>
</tr>
<tr>
<td>wsio_msi_disable</td>
<td>MSI/MSI-X</td>
<td>Disables interrupts on a per-vector or function basis.</td>
</tr>
<tr>
<td>wsio_msi_enable</td>
<td>MSI/MSI-X</td>
<td>Enables interrupts on a per-vector or function basis.</td>
</tr>
<tr>
<td>wsio_msi_get_cpus</td>
<td>MSI/MSI-X</td>
<td>Returns a list of CPUs available for I/O interrupts.</td>
</tr>
<tr>
<td>wsio_msi_query</td>
<td>MSI/MSI-X</td>
<td>Queries per-vector attributes.</td>
</tr>
<tr>
<td>wsio_msi_resize</td>
<td>MSI/MSI-X</td>
<td>Resizes the number of vectors associated with an MSI object.</td>
</tr>
</tbody>
</table>

To specify a TBI, call `wsio_intr_set_cpu_spec`, passing in the specific interrupt object. To specify an LBI, call `wsio_intr_set_irq_line`. Some platforms might not support both types of interrupts due to the underlying hardware, so the driver can be restricted to using only one type. The services 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 an LBI, call `wsio_intr_get_irq_line`. To get the interrupt address and vector for a TBI, call `wsio_intr_get_txn_info`. Use the `wsio_intr_activate` and `wsio_intr_deactivate` WSIO services to enable and disable interrupts.

MSI and MSI-X are similar to TBIs. They generate interrupts by writing a data value to a CPU’s interrupt address. With MSI interrupts, a PCI function gets a single 32- or 64-bit interrupt address, and can receive up to 32 different data values.

NOTE: The HP-UX MSI implementation permits a single data value only.

With MSI-X interrupts, an interrupt vector table is provided that can support up to 2048 unique interrupt vectors. Each vector consists of a 64-bit interrupt address and a 32-bit data value. Therefore, the interrupts can be distributed across multiple CPUs.

A PCI card must have specific hardware to support either MSI or MSI-X type interrupts. A PCI card can support both MSI and MSI-X interrupts, but only one type can be enabled at a time. To obtain a card’s MSI capabilities, call `wsio_msi_capability`. If the card supports MSI-X interrupts, call `wsio_msi_get_cpus` to obtain a list of CPUs that are available for processing interrupts.

To program the card’s interrupt vectors, call `wsio_msi_assign`. To enable and disable interrupts, call `wsio_msi_enable` and `wsio_msi_disable`, respectively.

Interrupt Migration

The following event types can be passed to a driver event handler:

**WSIO_EVENT_ONLINE_CPU** Indicates that a new CPU is available and can be used by the driver for TBIs or MSIs.
**Memory Allocation and Mapping I/O Space**

The WSIO CDIO has several memory allocation services that take advantage of kernel virtual memory (VM) features. HP recommends that drivers use these routines rather than the older kmalloc and kfree routines.

### Memory Allocation Services

Use the `wsio_alloc_mem_handle` and `wsio_free_mem_handle` services 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.

*Table 2-2* lists the memory allocation service routines and their descriptions. For more specific information on how to call these routines and the parameters passed to them, see the appropriate manpage in the *HP-UX 11i v3 Driver Development Reference*.

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wsio_alloc_mem</code></td>
<td>Allocates host memory using the memory allocation handle created by</td>
</tr>
<tr>
<td></td>
<td><code>wsio_alloc_mem_handle</code></td>
</tr>
<tr>
<td><code>wsio_alloc_mem_handle</code></td>
<td>Creates a memory handle that specifies the type of memory to allocate. It</td>
</tr>
<tr>
<td></td>
<td>returns a memory handle.</td>
</tr>
<tr>
<td><code>wsio_free_mem</code></td>
<td>Frees memory using the memory allocation handle created by</td>
</tr>
<tr>
<td></td>
<td><code>wsio_alloc_mem_handle</code></td>
</tr>
<tr>
<td><code>wsio_free_mem_handle</code></td>
<td>Frees (destroys) a memory handle allocated by <code>wsio_alloc_mem_handle</code>.</td>
</tr>
</tbody>
</table>

Typically, drivers call `wsio_alloc_mem_handle` in their init routines to specify the type of memory they want to allocate. The service returns a handle that the driver then passes into the `wsio_alloc_mem` and `wsio_free_mem` services when allocating and freeing buffers.

Drivers can allocate multiple memory handles for the different types of memory buffers they use. For example, a driver can specify a memory handle that only allocates physical memory below 4 GB. It can then specify another handle for allocating memory that is always physically contiguous.

### I/O Space Services

The WSIO CDIO introduces three new sets of services that drivers can use to discover, map, and access a card’s memory mapped registers (register services), configuration space (configuration space services), and I/O port space (I/O port space services).

### Register Services

Drivers call the `wsio_get_all_registers` service in their attach routine to obtain the memory mapped register sets of an interface card. They then call the `wsio_map_reg` service repeatedly to map each set in. Finally, the driver calls `wsio_read_regXX` to read the register or
wsio_write_regXX to write the register. There are separate services (XX = 8, 16, 32, or 64 bits) for each register size.

Table 2-3 lists the register services. For parameters, return codes, and examples, see the appropriate manpages in the *HP-UX 11i v3 Driver Development Reference*.

**Table 2-3 Register Service Routines**

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

**Configuration Space Services**

Device drivers use these services to discover and access configuration space resources. Use the wsio_map_cfg_handle service to obtain a configuration space handle. The handle is then passed to the wsio_cfg_inXX and wsio_cfg_outXX services. There are separate versions of these services (XX = 8, 16, 32, or 64 bits). The wsio_unmap_cfg_handle service releases the configuration space handle.

Table 2-4 lists the configuration space services. For parameters, return codes, and examples, see the appropriate manpages in the *HP-UX 11i v3 Driver Development Reference*.

**Table 2-4 Configuration Space Service Routines**

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>
<tr>
<td>wsio_map_cfg_handle</td>
<td>Obtains a configuration space access handle.</td>
</tr>
<tr>
<td>wsio_unmap_cfg_handle</td>
<td>Release a configuration space handle.</td>
</tr>
</tbody>
</table>

**I/O Port Space Services**

Drivers that control devices with I/O ports use these services to access those ports. The drivers first call wsio_get_ioports to obtain an array of I/O port addresses for the interface card. This is done in the driver attach or init routine when claiming or configuring the card. Next, the driver calls wsio_map_port to map each set of I/O port addresses in. The service returns a handle the driver can 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-, 32- or 64-bit values.

Table 2-5 lists the I/O port space service routines and their description. For parameters, return codes, and examples, see the appropriate manpages in the *HP-UX 11i v3 Driver Development Reference*.

**Table 2-5 I/O Port Space Service Routines**

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wsio_get_ioports</td>
<td>Obtains the addresses and sizes of I/O ports.</td>
</tr>
<tr>
<td>wsio_map_port</td>
<td>Obtains a port handle.</td>
</tr>
<tr>
<td>wsio_port_inXX</td>
<td>Reads from an I/O port.</td>
</tr>
</tbody>
</table>
Table 2-5 I/O Port Space Service Routines (continued)

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wsio_port_outXX</td>
<td>Writes to an I/O port.</td>
</tr>
<tr>
<td>wsio_unmap_port</td>
<td>Releases a port handle.</td>
</tr>
</tbody>
</table>

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.

The coherent platform implementation of Direct Memory Access (DMA) ensures that accesses to data in host memory by I/O devices are consistent with accesses by processor caches. Hardware in the platform maintains a consistent view of data in host memory as DMA transactions flow through the hardware.

Semicoherent platforms implement DMA in a manner similar to coherent systems. If data is read from an I/O device, software must synchronize that data into host memory after the DMA transaction completes.

The noncoherent platform implementation of DMA ensures that accesses to data in host memory by I/O devices are not made simultaneously with processor caches by hardware. Software must flush the processor caches prior to starting a DMA transaction by an I/O device. If data is read from an I/O device, purge the processor caches after the DMA transaction completes.

Figure 2-1 shows a noncoherent system.

![Figure 2-1 Noncoherent System](image)

Driver Requirements for Coherency

Drivers assume that platforms are noncoherent, and that they must control flushing and synchronization of the processor caches by calling `dma.sync.IO`. Drivers written for noncoherent platforms also work correctly on coherent and semicoherent platforms.

The `dma.sync.IO` routine is sensitive to the 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` synchronizes with the processor cache data that has been read into host memory.
PCI buses have special coherency exceptions that are discussed in “PCI Masters and Coherence” (page 205). For more information, see pci_errata(9), wsio_map_dma_buffer(9F), and wsio_fastmapd_dma_buffer(9F) in the HP-UX 11i v3 Driver Development Reference.

dma_sync_IO Rules

Drivers must call dma_sync_IO in the following cases:

• Before starting a write transaction
  For each buffer that is to write data, the driver must call dma_sync_IO with the IO_WRITE hint set. On noncoherent platforms, this flushes the associated processor caches. For all but the last buffer, the IO_NO_SYNC hint must also be set to reduce the performance penalty of synchronizing the cache flushes on noncoherent platforms.

• Before starting a read transaction
  For each buffer that is to read data, the driver must call dma_sync_IO with the IO_READ_START hint set. On noncoherent platforms, this purges the associated processor caches. For all but the last buffers, the IO_NO_SYNC hint must also be set to reduce the performance penalty of synchronizing the cache purges on noncoherent platforms.

• After completing a read transaction
  For each buffer that has read data, the driver must call dma_sync_IO with the IO_READ hint set. On noncoherent platforms, this purges the associated processor caches of data that might have been prefetched. For all but the last buffer, the IO_NO_SYNC hint must also be set to reduce the performance penalty of synchronizing the cache purges on noncoherent platforms. On semicoherent platforms, the processor caches synchronize with the data read when the IO_NO_SYNC hint is not set.

DMA Mapping

There are three address views of host memory: physical, 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.

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

• Virtual Address View
  When executing in virtual mode, processors 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.

• 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 hardware 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 beyond the I/O bus range. Address translation hardware embedded in the I/O adapter that connects the I/O bus to the processor/memory interconnect must be programmed to translate I/O bus addresses to host memory physical addresses. The special address translation hardware often participates in the cache coherence protocols with processors.

  Figure 2-2 shows the physical and I/O virtual addressing.
I/O Adapters

An I/O adapter provides 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. You can write drivers 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 have 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 do not have I/O adapter hardware and do not participate in the coherency protocol of the processor caches.

DMA Services

DMA services enable drivers to specify multiple DMA objects and configure them for different types of DMA. For example, a driver can specify two DMA objects and configure one for continuous DMA and another for packet DMA. In continuous DMA, a buffer is allocated and mapped long term for such purposes as control structures. Packet DMA maps buffers short term for one DMA, and then unmaps them.

Programming Considerations

WSIO DMA services hide the underlying platform hardware from drivers. Drivers that use these 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.
These services enable drivers to allocate multiple DMA objects and set attributes in each that favor the same types of DMA as those of the driver or interface. DMA attributes can take advantage of certain features of the underlying hardware such as prefetch depth, or they can specify how a buffer must be mapped for 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 \texttt{wsio_dma_set_device_attributes}(9F) and \texttt{wsio_set_dma_attributes}(9F) in the HP-UX 11i v3 Driver Development Reference.

Table 2-6 lists the DMA service interfaces. For parameters, return codes, and examples, see the applicable manpages in the HP-UX 11i v3 Driver Development Reference.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
\textbf{Interface Name} & \textbf{Description} \\
\hline
\texttt{wsio_allocate_dma_handle} & Obtains a DMA handle used to set up DMA. \\
\texttt{wsio_allocate_shared_mem} & Allocates and maps a buffer for DMA. \\
\texttt{wsio_dma_pass_thru} & Calls a pass-through function that might not otherwise be directly accessible. \\
\texttt{wsio_dma_set_device_attributes} & Sets DMA hints for all DMA objects associated with a device. \\
\texttt{wsio_fastmap_dma_buffer} & Maps an existing memory buffer for DMA. \\
\texttt{wsio_flush_shared_mem} & Flushes a buffer allocated by \texttt{wsio_allocate_shared_mem}. \\
\texttt{wsio_free_dma_handle} & Frees a DMA handle. \\
\texttt{wsio_free_shared_mem} & Unmaps and frees a buffer allocated by \texttt{wsio_allocate_shared_mem}. \\
\texttt{wsio_init_map_context} & Initializes a new DMA context. \\
\texttt{wsio_iova_to_phys} & Translates an I/OVA to the physical address of the buffer. \\
\texttt{wsio_map_dma_buffer} & Maps an existing memory buffer for DMA. \\
\texttt{wsio_remap_dma_buffer} & Maps an existing memory buffer using a previously allocated range of IOVAs. \\
\texttt{wsio_set_dma_attributes} & Sets DMA hints for the DMA object passed in only. \\
\texttt{wsio_set_dma_callback} & Sets a callback function for DMA. \\
\texttt{wsio_unmap_dma_buffer} & Unmaps a previously mapped memory buffer. \\
\hline
\end{tabular}
\caption{DMA Service Routines}
\end{table}

To use the WSIO DMA service, a driver must first allocate a DMA object by calling the service \texttt{wsio_allocate_dma_handle}. Then, the driver specifies the desired characteristics for the type of DMA by setting attributes in the object using the \texttt{wsio_set_dma_attributes} service. Drivers usually do this in their init routines when they claim their interface card.

DMA objects are allocated on a per driver instance. Then, the DMA object is saved by the driver and used to initiate all DMAs for that driver instance. Drivers can allocate more than one DMA object 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 only allocate one or two objects. When a driver has multiple DMA objects for a certain driver instance and needs to set an attribute for all of them, it uses the \texttt{wsio_dma_set_device_attributes} service.

To map an existing buffer for packet DMA, the driver calls either \texttt{wsio_map_dma_buffer} or \texttt{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 unmaps the buffer by calling \texttt{wsio_unmap_dma_buffer}. If \texttt{wsio_map_dma_buffer} is able to only partially map the buffer, it returns \texttt{WSIO_MAP_W_PARTIAL} instead of \texttt{WSIO_MAP_OK}. Be sure to check for partially mapped buffers when using \texttt{wsio_map_dma_buffer}. 

If a driver needs to map a new memory buffer into an existing range of IOVAs, it calls the `wsio_remap_dma_buffer` service. This service does not guarantee that the new buffer is mapped into exactly the same range of IOVAs. The functionality is dependent upon the underlying platform hardware supporting an IOPDIR.

If a driver needs to allocate and map a buffer for continuous DMA, it calls the `wsio_allocate_shared_mem` service. Usually these buffers are allocated in the `init` routine of the driver when it claims and configures the interface card. These buffers are usually mapped for long-term usage.

Any buffers allocated by `wsio_allocate_shared_mem` must be freed by calling `wsio_free_shared_mem`. Because these tend to be long-term buffers, this is only done when the driver is being unloaded by a Dynamically Loadable Kernel Module (DLKM) action.

Use the `wsio_flush_shared_mem` service to flush buffers for continuous DMA. Its effect is dependent on the underlying hardware; on some platforms it has no effect.

Use the `wsio_iowa_to_phys` service to translate an IOVA to the physical address of the buffer it maps.

You can set the `WSIO_DMA_ATTR_INTERLEAVE` attribute. Setting it tunes the mapping allocating scheme to favor certain types of DMA objects. The default value for this attribute is zero. This favors drivers that must 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 DMA objects involving small buffers that are processed sequentially. If your driver controls mass storage devices (for example, SCSI), use the default value. If your driver controls networking devices, set the attribute to one. When setting the attribute to one, the DMA buffers being mapped must reside on the same 4K page.

**WSIO Services**

The WSIO services can be grouped into the following sets:

- I/O Space Services
  - Register Services
  - Configuration Space Services
  - I/O Port Space Services
  - Endian Services
- DMA Services
- Interrupt Services
- Memory Allocation Services
- Driver Event Handling Services
- System Services
  - Description Services
  - Ordered Interrupt Services
  - System Administration Services
  - I/O Synchronization Services

The following sections provide an overview of each set of services.

**I/O Space Services**

This group of services enables a driver to discover, map, and access memory mapped registers, configuration spaces, and I/O Port spaces. A driver can also determine whether a bus is big endian or little endian.
Register Services

Device drivers use these services to discover and map device registers. Registers can contain information about the hardware device, or can configure and fine tune the device. 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 usually does this in its attach or init routine. These services replace the legacy WSIO services.

Table 2-3 (page 44) lists the register services interface names and services. For parameters, return codes, and examples, see the applicable manpages in the HP-UX 11i v3 Driver Development Reference.

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 contains hardware device information and some locations used to configure and fine tune the device. These services replace services provided by other CDIOs, such as PCI CDIO.

Table 2-4 (page 44) lists the configuration space services interfaces. For parameters, return codes, and examples, see the applicable manpages in the HP-UX 11i v3 Driver Development Reference.

I/O Port Space Services

These services enable a driver to discover, map, and access a card’s I/O port space, which is an area sometimes used when communicating with devices. These are new services. I/O space accesses do not do any endian checking. If necessary, the driver must perform the endian translation. Table 2-5 shows the I/O port space service routines and their description. For parameters, return codes, and examples, see the appropriate manpages in the HP-UX 11i v3 Driver Development Reference.

Endian Services

Devices drivers can use these services to determine whether the local bus is big endian or little endian. This is necessary in deciding whether to perform endian translation for shared memory accesses. Such translation is automatically performed for any access to or from configuration space. For access to registers, I/O ports, or shared memory, the device driver might need to perform its own translation.

Table 2-7 lists the endian services interface routines and their description. For parameters, return codes, and examples, see the appropriate manpages in the HP-UX 11i v3 Driver Development Reference.

<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>

DMA Services

DMA is a process that enables memory to be shared between an I/O device and the host processor main memory. Device drivers use the DMA Services to configure and use DMA resources. These new services replace the older legacy WSIO DMA services (for example, wsio_map, wsio_fastmap, wsio_remap, and wsio_unmap). The new services enable 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 the device will use.
Typically a driver allocates one or more DMA handles in its `init` routine. It then uses `wsio_dma_set_device_attributes` or `wsio_set_dma_attributes` to specify attributes of the type of DMA it will use the object for. A driver can configure one handle for large packet DMA and another for continuous DMA with small buffers. When setting up a DMA, the driver passes the handle to one of the DMA mapping routines.

Table 2-6 (page 48) lists the DMA services interfaces. For parameters, return codes, and examples, see the applicable manpages in the HP-UX 11i v3 Driver Development Reference.

### Interrupt Services

These services replace the older WSIO DMA services, which include `isrlink`. Device drivers use these services to obtain and set up interrupts. They enable a device driver to allocate multiple interrupts and associate separate interrupt service routines with each. Table 2-1 (page 41) lists the interrupt service interfaces. For parameters, return codes, and examples, see the appropriate manpages in the HP-UX 11i v3 Driver Development Reference.

### Memory Allocation Services

Memory allocation services enable drivers to allocate host memory. Table 2-2 (page 43) lists the memory allocation services interfaces. For parameters, return codes, and examples, see the appropriate manpage in the HP-UX 11i v3 Driver Development Reference.

### Driver Event Handling Services

Event handling services enable drivers to register an event handler and event mask with the WSIO. The driver event handler is called to handle events, and the event mask indicates the events the driver handles. Events include PCI OLAR actions such as `suspend` or `resume`.

Event handlers are registered on a per-driver basis. This is usually done in the driver’s `install` routine. The mask 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.

Table 2-8 lists the driver event handling services interfaces. For parameters, return codes, and examples, see the appropriate manpages in the HP-UX 11i v3 Driver Development Reference.

#### Table 2-8 Driver Event Handling Service Routines

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wsio_install_drv_event_handler</code></td>
<td>Registers a driver's event handler.</td>
</tr>
<tr>
<td><code>wsio_reg_drv_capability_mask</code></td>
<td>Registers an event capabilities mask.</td>
</tr>
</tbody>
</table>

When an event affects an I/O device, the WSIO first checks to see if there is an event handler associated with the device. If there is, it checks the event mask to see if the handler will respond to that type of event. If it does, it calls the handler.

All driver handlers must have the following caller syntax:

```c
void my_drv_handler(  
    wsio_generic_event_t *event_info_ptr  
);
```

### System Services

This service group provides device and platform information, interrupt status, and memory synchronization state for the device and CPU.
Description Service

This service is used by device drivers to set their I/O tree description. It is a necessary service because a meaningful description cannot be established automatically. A driver calls this in its init routine. The description appears in the ioscan output.

Table 2-9 lists the description service interface. For parameters, return codes, and examples, see the applicable manpage in the HP-UX 11i v3 Driver Development Reference.

Table 2-9 Description Service Routine

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wsio_set_description</td>
<td>Sets the I/O tree node description for this driver.</td>
</tr>
</tbody>
</table>

Ordered Interrupts Service

Device drivers use this service to determine whether interrupts are ordered with respect to DMA transactions. This tells device drivers whether it must explicitly perform a sync to ensure DMA transactions have completed.

Table 2-10 lists the ordered interrupts service interface. For parameters, return codes, and examples, see the applicable manpage in the HP-UX 11i v3 Driver Development Reference.

Table 2-10 Ordered Interrupts Service Routine

<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>

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.

Table 2-11 lists the system attribute services interfaces. For parameters, return codes, and examples, see the applicable manpage in the HP-UX 11i v3 Driver Development Reference.

Table 2-11 System Attribute Service Routines

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wsio_get_active_processor_count</td>
<td>Gets the active number of processors on the system.</td>
</tr>
<tr>
<td>wsio_get_processor_count</td>
<td>Gets the number of processors on the system.</td>
</tr>
<tr>
<td>wsio_get_system_params</td>
<td>Obtains information about the system.</td>
</tr>
</tbody>
</table>

I/O Synchronization Service

Device drivers use this service to explicitly synchronize the CPU and I/O device’s views of memory. If such a synchronization is not necessary, the function does not sync.

Table 2-12 lists the I/O synchronization service interface. For parameters, return codes, and examples, see the applicable manpage in the HP-UX 11i v3 Driver Development Reference.

Table 2-12 I/O Synchronization Service Routine

<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>
The WSIO CDIO provides a number of new driver services to register interface cards with the WSIO and General I/O (GIO) system, assign properties to them, and query hardware paths. An interface can be of type interface or transparent. All interfaces created 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 because no two interfaces can share the same hardware path. Services can convert an interface to its hardware path and vice versa.

Drivers can assign properties to an interface. A property has an ASCII string name and a value. The combination of the property name and the interface uniquely identify a property. No two properties assigned to the same interface can have the same name, but two properties assigned to different interfaces can have the same name. When a property is created, the caller can indicate if it is to be exported to any children underneath.

The following HP-UX 11i v2 services are deprecated in HP-UX 11i v3:

- `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`

The following services are new in HP-UX 11i v3:

- `wsio_reg_node`
- `wsio_claim_node`
- `wsio_reg_legacy`
- `wsio_destroy_legacy`
- `wsio_get_legacy`
- `wsio_put_drpv_priv`
- `wsio_get_drpv_priv`
- `wsio_node_get_isc`
- `wsio_async_scan`
- `io_hw_path_to_node`
- `io_node_to_hw_path`
- `io_hw_path_to_str`
- `io_str_to_hw_path`
- `io_get_instance`
- `io_post_event`
- `prop_create`
- `prop_modify`
- `prop_get`
- `prop_size`
- `prop_destroy`
- `prop_destroy_all`

The following sections provide a brief description of the new services. For parameters, return codes, and examples, see the applicable manpages in the `HP-UX 11i v3 Driver Development Reference`. 
wsio_reg_node Service

Call this service to register new I/O interfaces. It can be called in a driver's install, probe, or scan routine.

The following types of interfaces can be created:

- **T_INTERFACE** An interface
- **TDEVICE** A device
- **T_CDIO_PRIVATE** A transparent interface
- **T_VIRTBUS** A virtual bus interface
- **T_TGT_PATH** A target path I/O node
- **T_LUN_PATH** A LUN path I/O node

The **T_CDIO_PRIVATE** is a specialized type of interface. It has no associated hardware and is used to create hardware path elements. The **T_CDIO_PRIVATE** and **T_INTERFACE** interfaces have an isc handle associated with them. This handle can then be passed into the other WSIO routines. Both types of interface must be created in the driver's scan or probe routine. The **hw_path** and **parent** parameters are used together to build the hardware path for the new interface. If the **parent** parameter is not NULL, **hw_path** is assumed to be relative to the parent. Otherwise it is assumed to be absolute.

The service first checks to see if the interface already exists at the specified hardware path. If it does not, it creates it. Otherwise it compares the node_info structure information of the existing interface with those passed in as parameters. If they are different, it updates the node_info structure information with the new values and reports the difference to the I/O subsystem.

The service returns a node handle for the newly created entry.

wsio_claim_node Service

Call this service to claim an I/O interface.

wsio_reg_legacy Service

Call this service to register an I/O interface with a legacy hardware path (SCSI-2). The types of interfaces it can create are as follows:

- **T_INTERFACE**
- **TDEVICE**
- **T_VIRTBUS**

wsio_destroy_legacy Service

Call this service to destroy a legacy I/O node. When a SCSI-3 I/O node is destroyed, the corresponding legacy I/O nodes are also destroyed automatically.

wsio_get_legacy Service

Call this service to obtain information about the legacy I/O node associated with an I/O node.

wsio_put_drv_priv Service

Call this service to save a reference to driver private information with an I/O node.

wsio_get_drv_priv

Call this service to obtain a reference to driver private information with an I/O node.
wsio_node_get_isc Service
Call this service to obtain the WSIO isc structure associated with an interface I/O node.

wsio_async_scan Service
Call this service to initiate an asynchronous scan from an I/O node.

io_hw_path_to_node Service
Call this service to map a hardware path to an I/O node.

io_node_to_hw_path Service
Call this service to retrieve the hardware path for a specified I/O node.

io_hw_path_to_str Service
Call this service to convert a hardware path into an ASCII string, accounting for all aliases. This service is available for legacy and SCSI-3 nodes.

io_str_to_hw_path Service
Call this service to convert an ASCII string into a hardware path, accounting for all aliases. This service is available for legacy and SCSI-3 nodes.

io_get_instance Service
Call this service to obtain the class instance number of an I/O node.

io_post_event Service
Call this service to post events for I/O nodes.

prop_create Service
Call this service to create a new property for an interface. The node parameter identifies the interface. The name parameter is the name for the new property. The val, length, and flags parameters identify the initial data for the property. The flags parameter identifies characteristics of the property and the data referenced by val.

NOTE: This service is not safe to call on the Interrupt Control Stack (ICS) unless the P_NO_SLEEP flag is specified in the flags parameter.

By default, this property is visible to any child processes.
Table 2-13 lists the valid flags for properties that a driver creates.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_NO_COPY</td>
<td>If set, val is assumed to contain the immediate data that is saved. Otherwise, val is assumed to contain an address that references a data buffer. Copies the contents of the buffer.</td>
</tr>
<tr>
<td>P_NO_EXPORT</td>
<td>Does not export the property to any child processes.</td>
</tr>
<tr>
<td>P_MODIFY</td>
<td>Modifies the property, if it already exists.</td>
</tr>
<tr>
<td>P_NO_SLEEP</td>
<td>The call does not block until resources are available.</td>
</tr>
</tbody>
</table>

When creating a property, a reference to a kernel memory data structure can be saved by passing in the address and size of the structure as the val and length parameters. The kernel memory data
structure must then be persistent in memory as long as the property exists. By default, the service
copies the contents of the data to an internal buffer.

**prop_modify Service**

Call this service to modify the value of any property associated with an interface. The node
handle of the interface is passed in as the first parameter. The val and length parameters define
the new data for the property. The length parameter indicates the size of the new data. If the size
is greater than the original data, the service can fail or block unless P_NO_SLEEP is specified in
the flags.

Table 2-14 lists the valid flags for properties that a driver modifies.

**Table 2-14 Interface Property Flags (Modify)**

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_NO_COPY</td>
<td>If set, val is assumed to contain the immediate data that is saved. Otherwise, val is assumed to contain an address that references a data buffer. Copies the contents of the buffer.</td>
</tr>
<tr>
<td>P_NO_SLEEP</td>
<td>The call does not block until resources are available.</td>
</tr>
</tbody>
</table>

**prop_get Service**

Use this service to retrieve the current value of a property associated with the interface identified
by the node parameter. The value returned depends on how the flags are specified. If P_NO_COPY
is set, prop_get copies a pointer to the property data into the location pointed to by ptr. The
ptr parameter must reference a buffer large enough to hold the data. If the P_NO_COPY flag is
not set, the property data is copied to ptr.

**prop_size Service**

This service returns the size of an attribute identified by the name and node parameters.

**prop_destroy Service**

Call this service to destroy a property.

**prop_destroy_all Service**

Call this service to destroy all properties associated with the node parameter.

**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.

Use object caching 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
enables 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
(for example, memory pools). Incorrect use of deallocated memory through stale pointer references
can lead to memory corruption of other memory objects in the kernel. 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, making it easier to isolate defects.
In addition, special attributes can be applied to each arena. For example, certain types of memory objects might require stricter alignment or contiguous physical memory. You can specify these and other attributes 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 can have its own arena. Arenas associated with fixed-size objects are called fixed arenas. Arenas associated with variable-sized 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, remove the arena 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 to 16K bytes, are considered large objects. For example, an object with a requested size of 1200 bytes that requires 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 following interfaces dynamically allocate kernel memory:

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

- `kmem_arena_create` creates either a fixed-size or variable-sized arena. It is called once before any allocations from an arena are made, preferably when the kernel module is loaded or installed. Default and nondefault 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` 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` returns variable-sized allocations. The first parameter is the handle returned from the call to `kmem_arena_create` that create the variable-sized arena.

- `kmem_arena_free` deallocates the memory allocated by either `kmem_arena_alloc` or `kmem_arena_varalloc`. The kernel returns the memory to the same arena from which the memory originated.

- `kmem_arena_destroy` 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 is deallocated before removing the arena.
kmem_arena_attr_t  Specifies attributes for creating an arena.
kmem_handle_t  Identifies arenas. This is an opaque type.

Arena allocator interfaces are declared in the `<sys/vm_arena_iface.h>` header file. See the manpages for these interfaces in the HP-UX 11i v3 Driver Development Reference.

**Fixed Size and Variable Size Arenas**

The following are the available arena types:

- **Fixed-Sized Arena**  Objects allocated from this arena are of the same size.
- **Variable-Sized Arena**  Objects allocated from this arena may vary in size.

Fixed-size arenas have the following benefits:

- Allocations are faster than with variable-sized arenas.
- Use object caching to improve performance.
- Efficiently use memory because 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. Because there is higher memory consumption in using arenas with variable-sized objects, HP recommends that each kernel module create one arena for all its variable-sized objects.

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

The default alignment for allocated objects is 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 deallocations. Create arenas 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`. The following attributes can be modified by a kernel module:

- **Constructor**  Every 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. Optionally, specify the constructor function in `kat_ctor` to implement object caching. The default is no object caching.
- **Destructor**  Dismantles the object every time an object is reclaimed from an arena. Optionally, specify the destructor function in `kat_dtor` to implement object caching.
- **Alignment**  By default, allocated objects are at least double-word (64-bit) aligned. Change the alignment to other than the default alignment (only for fixed-sized arenas) by setting the alignment value in `kat_align`.
- **Number of objects per refill**  For sizes less than a page, by default the kernel splits a 4K page into multiple objects during refill. For object sizes 4K and greater only one object is refilled by default. Set the
refill value in `kat_refillcnt` to refill greater quantities heavily used arenas to improve performance.

**Maximum Allocation**  
Set the maximum count in `kat_maxcnt` to limit the number of objects allocated to this arena. Use this attribute to detect memory leaks and prevent bringing down the system.

**Minimum number of objects in free list**  
Set the value in `kat_minfcnt` to advise the kernel to keep a minimum number of objects in the free list per SPU during garbage collection. This attribute is an advisory for small objects. The garbage collector skips 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 attempts to free as many pages as possible to reduce memory fragmentation. After all complete pages are collected, the length of the free list can be less than this number. For large objects, this value is enforced.

**Multicache Line Size**  
By default, the kernel shares the same cache line as the object to store the object header. This can cause problems on noncoherent I/O machines when the object is used for DMA. In such instances, set `KAT_MULTICACHE_SIZE` in `kat_flags`.

**No Large Page**  
Set `KAT_NO_LGPG` in `kat_flags` to instruct the kernel to use 4K pages rather than a superpage (large pages). This attribute is required by certain drivers to remap a page.

**Cache Xlarge Objects**  
By default, Xlarge objects are not cached. Specify `KAT_CACHE_XLARGE_OBJECTS` in `kat_flags` to override this default.
Example 2-1 Fixed-Sized 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_abc(void)
{
    struct bar *my_obj;
    my_obj = kmem_arena_alloc(bar_handle, M_WAITOK);
    ...
    kmem_arena_free(my_obj, M_WAITOK);
}
```

Example 2-2 Overriding the Default Arena 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_abc(void)
{
    struct bar *my_obj;
    my_obj = kmem_arena_alloc(bar_handle, M_WAITOK);
    ...
    kmem_arena_free(my_obj, M_WAITOK);
}
Example 2-3 Variable-Sized 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_abc(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

Most of the allocations in the kernel are for fixed-sized objects. This enables each module the option to cache objects and avoid costly initialization. Object caching has the following benefits:

- Avoids costly construction of objects after each allocation.
- Avoids having each kernel module write code to manage the cache.
- Enables the kernel to efficiently reclaim memory when there is memory pressure.
- Enables 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_abc(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 because it must allocate memory for the lock, then initialize the lock. The call to `alloc_spinlock` can be moved to a constructor function, avoiding the call to `alloc_spinlock` after each allocation. The constructor function is called by the kernel only when the cache (free list) is empty and new objects are created. The code with the new allocator can be written 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);
    ...
    kmem_arena_free(my_obj, M_WAITOK);
}
```
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_abc(void){
    struct bar *my_obj;
    my_obj = kmem_arena_alloc(bar_handle, M_WAITOK);
    do_useful_work();
    kmem_arena_free(my_obj, M_WAITOK);
}

It is generally a bad programming practice to allocate memory within critical sections of your
code. The kernel memory allocator might be required to do a lot of work to satisfy an allocation
request, and will sleep to wait for memory to become available, if necessary. This can have a
large negative impact on the performance of your code.

There are two critical sections in which sleeping is prohibited by the kernel:
• Interrupt routines (code running in interrupt context)
• Code executed while holding a spinlock.

The M_NOWAIT flag enables you to allocate memory in these contexts. If M_NOWAIT is specified,
the kernel memory allocator only tries the fast paths of memory allocation, returning NULL if
unsuccessful. In this case, your code must be prepared to deal with a memory allocation failure.
Typically, this requires retrying the allocation outside the critical section with the M_WAITOK
flag specified.

HP strongly recommends that whenever possible you structure your code so all memory
allocations are done outside of critical sections.

Object Constructors and Destructors

To use object caching, the kernel module must 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 is 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. The destructor function is optional. The prototypes for
the constructor function and destruction function are the following:

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 corresponds to the *flags* specified for the initial allocation. The constructor and destructor functions must honor these flags. If the client specifies *M_NOWAIT*, the constructor function cannot block (sleep). The constructor function must return a nonzero 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 refills 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. Use this functionality to link together a stack of objects. The kernel module must ensure that these recursions are safe. Because there is a hard limit on the kernel stack size, keep the number of recursions small.

There is no mechanism to return an error returned from the constructor function to the caller of *kmem_arena_alloc*. If a complicated error recovery is necessary, do this after allocation. For example, if a constructor function fails and this error must be communicated to the caller, initialize after allocation.

The destructor function must deallocate all objects allocated by the constructor, if any. The destructor function must honor all flags passed to the destructor function. No flags are defined at present. The destructor function cannot fail because there is no mechanism to convey this error, because the destructor function is executed asynchronously to the call to *kmem_arena_free*.

**CAUTION:** DLKM drivers that can be unloaded must not specify a destructor function. The kernel can 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 crashes when it tries to call the destructor function.

### Legacy Kernel Memory Allocation Interfaces

The *kmalloc* and *kfree* legacy interfaces are still supported, although HP discourages their use. The kernel creates a variable size compatibility arena for each of the M-types (for example, *M_DMA*, *M_IOSYS*, and *M_IHV*) of the old interface.

The *kmalloc* routine allocates an object from one of the compatibility arenas. Its prototype is as follows:

```c
void *kmalloc(
    size_t size,
    int type,
    arena_flags_t flags
);
```

The *kfree* routine deallocates object allocated by *kmalloc*. Its prototype is as follows:

```c
void kfree(
    caddr_t va,
    int type
);
```

Over time, the number of compatibility arenas is expected to decrease as kernel modules migrate over to using the Arena Allocator interfaces.

Binary compatibility with future releases of HP-UX is guaranteed for the *type* value of 0 only. For example:

```c
p = kmalloc(size, 0, M_WAITOK);
kfree(p, 0);
```
TIP: HP recommends that kernel modules use the Arena Allocator interfaces and stop using the kmalloc and kfree legacy interfaces.

**HP-UX Kernel I/O Mapping Interfaces**

The HP-UX kernel provides a set of interfaces that drivers can use to map a range of device I/O memory to a kernel virtual address or user virtual address. These interfaces enable driver to do the following:

- Map I/O memory to a user process private address space, disable inheritance of memory mapped I/O across fork, and specify read-only access to I/O memory.
- Accelerate store instructions to I/O memory by enabling write coalescing on Integrity platforms that support the write coalescing memory attribute.
- Access I/O memory mapped to both the kernel and a user process without calling iomap_enter_shared_acc and iomap_exit_shared_acc.

Table 2-15 shows the I/O mapping interfaces and their description. For parameters, return codes, and examples, see the appropriate manpages in the HP-UX 11i v3 Driver Development Reference.

**Table 2-15 Kernel I/O Mapping Interfaces**

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kernel_iomap</td>
<td>Maps I/O memory to a kernel virtual address.</td>
</tr>
<tr>
<td>kernel_iounmap</td>
<td>Unmaps I/O memory for the assigned kernel virtual address.</td>
</tr>
<tr>
<td>user_iomap</td>
<td>Maps I/O memory to a user virtual address.</td>
</tr>
<tr>
<td>user_iomap_private</td>
<td>Maps I/O memory to a private user virtual address.</td>
</tr>
<tr>
<td>user_iounmap</td>
<td>Unmaps I/O memory for the assigned user address.</td>
</tr>
<tr>
<td>iomap_enable_wc</td>
<td>Enables write coalescing for memory-mapped I/O.</td>
</tr>
<tr>
<td>iomap_pagesize</td>
<td>Returns the base page size (in bytes) for memory-mapped I/O.</td>
</tr>
</tbody>
</table>

The following I/O mapping interfaces were introduced in HP-UX 11i v2, but are deprecated in HP-UX 11i v3:

- iomap_enter_shared_acc Enters code that accesses shared I/O memory.
- iomap_exit_shared_acc Exits code that accesses shared I/O memory.

If your driver does not need user processes to access their device I/O memory, use the wsio_map_reg WSIO interface, which maps device registers (PCI BAR memory ranges) to kernel memory that is accessible by the driver only.

The following sections provide information for drivers that need to map their device I/O memory to user processes, which enables user processes to access the I/O memory directly.

**User Process Mapping Guidelines**

To map device I/O memory to a user process, a driver can call either user_iomap or user_iomap_private. HP recommends that drivers call user_iomap_private to map I/O memory to private user virtual address space because of the following:
Drivers do not take resources away from the address space of shared objects. Shared user virtual address space is scarce for 32-bit user processes. Allocating shared user virtual address for I/O mappings removes address space that can be allocated to other shared objects (for example, memory-mapped files and shared memory).

Drivers can disable inheritance of memory mapped I/O by child processes. When a parent calls fork and creates a child process, the child process usually inherits access to all shared and private objects that the parent process can access. A call to user_iomap_private disables the normal inheritance semantics of fork.

Drivers can restrict access to I/O memory to read-only access for user processes. Drivers are granted read-write access by kernel_iomap. A call to user_iomap_private restricts user processes to read-only access to the same I/O memory.

If many user processes are expected to share access to the same I/O memory (for example, graphics frame buffers), call user_iomap instead of user_iomap_private. Both user_iomap and user_iomap_private behave as follows:

- Drivers can share access to I/O memory concurrently with user processes.
  You can call either routine for the same I/O memory that the driver has mapped by calling kernel_iomap with the PROT_URW attribute set.
- Multiple user processes can map to the same I/O memory.
  Calling user_iomap allocates a unique global virtual address to 32-bit processes and different global virtual address to 64-bit processes. Calling user_iomap_private allocates an aliased private virtual address to each user process.

**NOTE:** On HP-UX 11i v3 for Integrity platforms, I/O mapped shared 64-bit user virtual addresses are in a separate name space from kernel virtual addresses. Previous versions of HP-UX allocated virtual addresses from the kernel region; HP-UX 11i v3 allocates from shared user space.

Calls to user_iomap and user_iomap_private can fail for the following reasons:

- 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, the protection is set to PROT_KRW.
- The pages overlap an I/O memory range that was already mapped by a previous call to user_iomap, user_iomap_private, or kernel_iomap.
  To map an I/O memory range multiple times, the mapping must cover the same range (paddr and count parameters must be the same).
- The driver previously called wsio_map_reg 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, WSIO has mapped for kernel-only access (up to 8 KB) of the first PCI BAR in the device that specifies the memory range.

**Finding the Physical I/O Address**

Drivers must find the processor’s view of the physical I/O address obtained from a device’s PCI BAR. The following code example shows how a driver can obtain the physical I/O address using available kernel interfaces, independent of the processor and platform implementation.
NOTE: For simplicity, error handling is not shown.

```c
uint32_t barvalue;      /* Value read from a PCI BAR */
caddr_t  baraddr;       /* PCI BAR view of I/O address */
void    *physaddr;     /* Processor 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
 * must 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 view of its I/O address to a kernel
 * virtual address. The map_mem_to_host() function
 * translates the PCI BAR address to the processor view
 * of the I/O address (which may be different), and then map
 * the processor 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.
 *
 * Note 1: Physical I/O addresses are not equivalent to
 *         kernel virtual addresses on Integrity implementations.
 *
 * Note 2: 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 the driver has the processor view of the physical
 * I/O address, unmap the kernel virtual address.
 * If we do not unmap the kernel virtual address, then a
 * call to kernel_iomap() passing PROT_URW fails, and on
 * Integrity 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);

/*
 * Let the kernel share access with the user.
 */
kvaddr = kernel_iomap(NULL, physaddr, MY_PAGES_OF_IOMEM,
                      PROT_URW);
```

Write Coalescing Guidelines (Integrity platform only)

Write coalescing accelerates the performance of store instructions to memory mapped I/O addresses (PIO writes). It combines multiple data store instructions to contiguous I/O addresses
and issues burst writes to the target device. Write coalescing increases PIO write bandwidth
while reducing individual store instruction latencies.
The target device must be able to receive different sized coalesced write transactions. A target
device can also receive data out of order from the store instructions issued by the processor.
Write coalescing is typically enabled for payload data; writes to control and status registers are
not candidates for write coalescing.
Drivers typically use the volatile qualifier for data written to and read from I/O memory. Do not
use the volatile qualifier for data to be write coalesced. After the last coalesced write, add the
volatile qualifier. The volatile qualifier causes the C language compiler to issue store instructions
with release semantics, which forces the processor to flush its coalescing buffer. The following
example illustrates the correct use of the volatile qualifier:
long payload[BUFFLEN+1];
long *iop = xyz; //Non-volatile pointer to I/O memory
for (i = 0; i < BUFFLEN; i++) {
/*
* The following non-volatile store writes data to the payload
* on the target device. If write coalescing is enabled, the
* processor inserts the written data into its coalescing
* buffer. When the coalescing buffer is filled, the processor
* flushes (writes) its contents to the target device.
*/
*iop++ = payload[i]; //Non-volatile store to coalescing buffer
}
/* The last write is a volatile store to flush the processor's
* coalescing buffer. Data from the coalescing buffer are written
* to the target device before the volatile store completes.
*/
(volatile long *)iop = payload[i];

HP-UX Kernel I/O Mapping Interfaces

67


3 Multiprocessing

HP-UX servers and workstations are either uniprocessor or multiprocessor systems. Current and new drivers for servers and workstations must be multiprocessing safe because they may eventually run on multiprocessor systems. This chapter describes the kernel services that drivers on multiprocessor systems use to handle synchronization. This chapter addresses the following topics:

- “Uniprocessing and Multiprocessing Comparison”
- “Synchronization Mechanisms” (page 69)

Uniprocessing and Multiprocessing Comparison

A uniprocessor (UP) system has one processor. A driver in a UP system executes one thread at a time. This can be either a kernel thread (processed by the upper half of a driver), or on the Interrupt Control Stack (ICS) in a processor interrupt context (processed by 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 has two or more processors. A driver in this system can execute multiple threads concurrently. For each processor, the driver can execute in a kernel thread or in the interrupt context of that processor. The MP driver synchronization model coordinates execution among multiple kernel threads, and between the driver’s upper and lower halves. HP-UX is a multiprocessing operating system, and its drivers must use synchronization mechanisms designed for an MP system. Drivers that use these mechanisms work correctly on both MP and UP systems.

Synchronization Mechanisms

Figure 3-1 shows the synchronization areas.

Figure 3-1 Synchronization Areas

The HP-UX 11i v3 kernel supports the following types of synchronization:

- Spinlocks
- Mutexes
- Reader-writer locks and reader-writer spinlocks
- Condition variables
- Counting semaphores

Mutexes and reader-writer locks are adaptive locks.

Figure 3-2 shows the synchronization mechanism.
To use a synchronization mechanism, a driver performs the following steps:

1. **Allocate the mechanism.**
   The driver allocates and initializes a spinlock resource. The driver maintains pointers to the lock and data area.

2. **Acquire the mechanism.**
   The driver takes ownership of the spinlock and locks it.

3. **Release the mechanism.**
   The driver releases ownership of the spinlock and unlocks it.

4. **Deallocate the mechanism.**
   The driver terminates usage of the spinlock resource.

### Adaptive Locking

Under contention, an adaptive lock either blocks or spins, depending on which is more efficient. The advantage of spinning is that the thread does not incur the overhead of a context switch. The advantage of blocking is that it frees the CPU for useful work. In choosing between spinning and blocking, adaptive locks get the benefits of both mechanisms. From a programming perspective, adaptive locks have the following advantages:

- Adaptive locks are as efficient as spinlocks for locks with short hold times or no contention.
- Adaptive blocking locks are as efficient as HP-UX 11i v2 blocking locks for contended locks that are held across blocking operations.
- Adaptive blocking locks free you from having to determine the maximum lock hold time or if any called function blocks.
- Adaptive locks (compared to spinlocks) can improve system responsiveness.

### Spinlocks

Spinlocks are the most heavily used synchronization mechanism in the HP-UX kernel, and are the basic locking primitive the kernel uses for short-term locks. A spinlock has the following characteristics:

- It protects data accessed from either a kernel thread or an interrupt context.
- A spinlock is owned by only one processor at any given time.
- 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 spin while waiting; they do not block.

The other processors burn CPU cycles without doing useful work when this occurs. Design your drivers to hold spinlocks for only short periods of time. As a rule, if a spinlock is held for longer than a few milliseconds, it is being held too long.
• While a processor owns a spinlock, interrupts to the processor from I/O devices are disabled. This avoids a potential interruption deadlock. For example, driver code is executing on a processor and owns (has locked) a spinlock. If I/O device interrupts are not disabled, one can cause the driver’s Interrupt Service Routine (ISR) to be entered on the same processor. If the ISR attempts to lock the same spinlock, a deadlock occurs, the spinlock is already owned by the processor, and the ISR spins forever.

• Spinlocks must not be held across calls to system services that may block (put the thread to sleep).

Beta semaphores are only held in a kernel thread context, and usually for longer durations than spinlocks.

Spinlock Routines

HP-UX provides the following spinlock routines:

- **spin_attr_init**: Initializes the spinlock attribute structure.
- **spin_alloc**: Allocates and initializes a spinlock.
- **spin_dealloc**: Deallocates (terminates) a spinlock.
- **spin_owned**: Checks if the processor owns a spinlock.
- **spin_locks_held**: Determines whether the processor owns any type of spinlock.
- **spin_lock**: Acquires (locks) a spinlock.
- **spin_trylock**: Conditionally acquires (locks) a spinlock.
- **spin_unlock**: Releases (unlocks) a spinlock.

For more information, see the *HP-UX 11i v3 Driver Development Reference*.

Mutexes

Mutexes are adaptive, mutually-exclusive, blocking semaphores. They replace the HP-UX 11i beta semaphores, which are supported on HP-UX 11i v3, but their use is deprecated. For descriptions of the deprecated interfaces, see the *HP-UX 11i v3 Driver Development Reference*.

When a thread acquires a mutex, it is the owning thread until the mutex is released. The owning thread can subsequently block (or sleep) and still keep ownership. Threads waiting to acquire an owned mutex spin or block, depending on which is more efficient.

Mutexes protect data accessed by a driver’s upper half (executing in a kernel thread that can block). They do not protect data that are accessed by a driver’s lower half (executing in an interrupt context that cannot block).

A mutex can be held across context switches and does not mask interrupts.

Mutex Routines

HP-UX provides the following mutex routines. For more information, see the *HP-UX 11i v3 Driver Development Reference*.

- **mutex_attr_init**: Initializes the mutex attribute structure to its default value.
- **mutex_attr_setflag**: Sets mutex attribute structure values.
- **mutex_alloc**: Dynamically allocates and initializes a mutex.
- **mutex_dealloc**: Destroys and deallocates a mutex.
- **mutex_lock**: Acquires (locks) a mutex.
- **mutex_unlock**: Releases (unlocks) a mutex.
- **mutex_trylock**: Conditionally acquires (locks) a mutex.
- **mutex_owned**: Tests whether a thread has the mutex locked.
Reader-Writer Locks

Reader-writer locks are blocking semaphores that support multiple concurrent readers or a single writer. HP recommends that you use them for data structures whose predominant form of access is reading. In supporting multiple readers, a reader-writer lock promotes parallelism. However, when there is significant contention on a lock, a mutex or reader-writer spinlock can be more efficient because of the relatively low overhead.

A reader-writer lock is an adaptive lock.

Reader-Writer Lock Routines

HP-UX provides the following reader-writer lock routines. For more information, see the HP-UX 11i v3 Driver Development Reference.

- `rwlock_attr_init` initializes the reader-writer lock attribute structure to its default value.
- `rwlock_attr_setflag` sets reader-writer lock attribute structure values.
- `rwlock_alloc` dynamically allocates and initializes a reader-writer lock.
- `rwlock_dealloc` destroys and deallocates a reader-writer lock.
- `rwlock_rdlock` acquires (locks) a read lock on a reader-writer lock.
- `rwlock_wrlock` acquires (locks) a write lock on a reader-writer lock.
- `rwlock_unlock` releases (unlocks) a reader-writer lock.
- `rwlock_tryrdlock` conditionally acquires (locks) a read lock on a reader-writer lock.
- `rwlock_trywrlock` conditionally acquires (locks) a writer lock on a reader-writer lock.
- `rwlock_tryupgrade` upgrades a read lock to a write lock if there are no other readers and no waiting writers.
- `rwlock_upgrade` upgrades a read lock held to a write lock.
- `rwlock_downgrade` downgrades a write lock to a read lock.
- `rwlock_owned` returns the current lock mode for the calling thread.
- `rwlock_wrowned` determines if the caller has a write lock on the reader-writer lock.

Reader-Writer Spinlocks

Reader-writer spinlocks support multiple concurrent readers or a single writer. HP recommends that you use them when the predominant form of access is reading. As with standard spinlocks, all but the highest priority interrupts are masked while spinning for the lock and while holding the lock. A thread cannot block while holding a reader-writer spinlock.

A reader-writer spinlock has the following characteristics:
- Can be used in interrupt or thread context, assuming that the lock hold time is short.
- Supports parallelism by allowing multiple CPUs to read data concurrently. Readers and writers are services in first-in/first-out (FIFO) order.

Reader-Writer Spinlock Routines

HP-UX provides the following reader-writer spinlock routines. For more information, see the HP-UX 11i v3 Driver Development Reference.

- `rwspin_attr_init` initializes the reader-writer spinlock attribute structure to its default value.
- `rwspin_attr_setflag` sets reader-writer spinlock attribute structure values.
- `rwspin_alloc` dynamically allocates and initializes a reader-writer spinlock.
rwspin_dealloc | Destroys and deallocates a reader-writer spinlock.
rwspin_rdlock | Acquires (locks) a read lock on a reader-writer spinlock.
rwspin_wrlock | Acquires (locks) a write lock on a reader-writer spinlock.
rwspin_rdunlock | Releases (unlocks) read access on a reader-writer spinlock.
rwspin_wrunlock | Releases (unlocks) write access on a reader-writer spinlock.
rwspin_owned | Returns the current lock mode for the calling thread.

Deadlocks

If a driver acquires a mutex, spinlock, reader-writer lock, or reader-writer spinlock in an incorrect order, a deadlock can occur.

For example, processes 1 and 2 both need resource C, then resource D to complete an activity. If process 1 locks resource C, then tries to acquire resource D and process 2 locks resource D, then tries to acquire resource C, each process is blocked forever waiting for the other.

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

Lock Acquisition Rules

All locks (and the resources they protect) are assigned a lock order when initialized, which is used as follows:

- When an executing thread acquires a spinlock or reader-writer spinlock unconditionally, the order of the requested spinlock or reader-writer spinlock must be greater than the order of any spinlock or reader-writer spinlock the processor already holds.
- When a kernel thread acquires a mutex or reader-writer lock unconditionally, the order of the requested mutex or reader-writer lock must be greater than the order of any mutex or reader-writer lock the kernel thread already holds.
- If the orders of the acquired and held mutexes or reader-writer locks are equal, both mutexes or reader-writer locks must have the deadlock safe option set. This option is set by ORing the order with the SEMA_DEADLOCK_SAFE bit when the mutex or reader-writer lock is initialized.
- Lock orders for spinlocks occupy a separate and higher order set than lock orders for mutex and reader-writer locks. A thread can acquire a spinlock while holding a mutex, but it cannot acquire a mutex unconditionally while holding a spinlock.

Lock Orders

The <sys/semglobal.h> header file contains the lock orders used by kernel services supplied by HP. Drivers typically choose a low value lock order so that the driver can hold its own spinlock, mutex, reader-writer lock, or reader-writer spinlock while calling a kernel service.

Synchronization with Condition Variables

Condition variables enable a thread to wait for specific conditions to occur. They replace the HP-UX 11i sleep and wakeup routines, which are supported HP-UX 11i v3, but their use is deprecated. For descriptions of the deprecated interfaces, see the HP-UX 11i v3 Driver Development Reference.

Typically, a thread uses a condition variable to check for a condition while holding a lock that prevents the condition from changing. If the thread cannot proceed because of the condition, the thread calls a condition variable wait routine cv_wait with the associated condition variable and lock. Later, another thread can obtain the lock, update the condition, release the lock, and unblock the waiting thread using cv_signal or cv_broadcast.
**NOTE:** The `cv_wait` routine atomically blocks the calling thread and unlocks the lock. This guarantees that the thread does not miss a condition change signal. The `cv_wait` routine returns to the thread with the lock locked.

Condition variables have the following characteristics:
- Intended for use in thread context, although condition variable events can be signaled or broadcast from an interrupt context.
- Can be used with spinlocks, mutexes, reader-writer locks, and reader-writer spinlocks.
- Offer better performance than the `sleep` and `wakeup` routines.

### Condition Variable Routines

HP-UX provides the following condition variable routines. For more information, see the HP-UX 11i v3 Driver Development Reference.

- **cv_attr_init**: Initializes the condition variable attribute structure to its default value.
- **cv_attr_setdata**: Sets condition variable attribute structure values.
- **cv_alloc**: Dynamically allocates and initializes a condition variable.
- **cv_dealloc**: Destroys and deallocates a condition variable.
- **cv_wait**: Waits for the occurrence of the condition associated with the condition variable.
- **cv_timedwait**: Waits for the occurrence of the condition associated with the condition variable or the expiration of a timer.
- **cv_timedwait_sig**: Waits for the occurrence of the condition associated with the condition variable, an HP-UX signal, or the expiration of a timer.
- **cv_wait_sig**: Waits for the occurrence of the condition associated with the condition variable or an HP-UX signal.
- **cv_signal**: Wakes up one of the threads waiting on the condition associated with the condition variable.
- **cv_broadcast**: Wakes up all threads waiting on the condition associated with the condition variable.

### Counting Semaphores

Counting semaphores are blocking primitives used in a producer/consumer environment. They are typically used in conjunction with a queue that is protected by a spinlock or mutex. Counting semaphores can be incremented (credits) or decremented (debits). If the decrementing of a counting semaphore results in a negative value, the thread is blocked until the semaphore is incremented.

Counting semaphores have the following characteristics:
- Can be initialized, incremented, and decremented by a specified value.
- Must not be used as mutexes because they do not detect deadlocks.
- Can be used early in the boot sequence if the semaphore is available.

### Counting Semaphore Routines

HP-UX provides the following counting semaphore routines. For more information, see the HP-UX 11i v3 Driver Development Reference.

- **csema_attr_init**: Initializes the counting semaphore attribute structure to its default value.
- **csema_attr_setdata**: Sets counting semaphore attribute structure values.
- **csema_alloc**: Dynamically allocates and initializes a counting semaphore.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>csema_dealloc</td>
<td>Destroys and deallocates a counting semaphore.</td>
</tr>
<tr>
<td>csema_increment</td>
<td>Increments the counting semaphore by a specified value.</td>
</tr>
<tr>
<td>csema_decrement</td>
<td>Decrements the counting semaphore by a specified value.</td>
</tr>
<tr>
<td>csema_trydecrement</td>
<td>Conditionally decrements the semaphore by a specified value.</td>
</tr>
<tr>
<td>csema_decrement_sig</td>
<td>Decrements the counting semaphore by a specified value, returning an error if an HP-UX signal is received.</td>
</tr>
<tr>
<td>csema_timeddecrement</td>
<td>Decrements the counting semaphore by a specified value, returning an error if a timer expires.</td>
</tr>
<tr>
<td>csema_timeddecrement_sig</td>
<td>Decrements the counting semaphore by a specified value, returning an error if a an HP-UX signal is received or a timer expires.</td>
</tr>
<tr>
<td>csema_value</td>
<td>Returns the current value of a specified counting semaphore.</td>
</tr>
</tbody>
</table>
This chapter describes the steps and code necessary to write a device driver. Read this chapter, then read other relevant chapters in this manual that provide details about the specific driver. For example, if you write PCI Drivers, see Chapter 9 (page 197). If you want to write a loadable driver module, see Chapter 5 (page 137).

When writing a device driver, follow these steps:

• “Step 1: Choosing a Driver Name”
• “Step 2: Choosing System Header Files”
• “Step 3: Defining Installation Structures” (page 78)
• “Step 4: Identifying Routines for the Driver” (page 85)
• “Step 5: Writing Configuration Routines” (page 87)
• “Step 6: Writing Entry Point Routines” (page 102)
• “Step 7: Writing Other Driver Routines” (page 123)

Step 1: Choosing a Driver Name

The driver needs a name. It must be unique to avoid conflict with kernel routines and global variables.

TIP: HP recommends using the company’s name and something that indicates the driver’s purpose.

The name is required in the following places:

• Module metadata file (Chapter 6 (page 157))
• In the name field of the drv_info_t structure ("drv_info_t Structure" (page 80))
• As the prefix of the name of the installation routine, driver_install ("Writing a driver_install Routine" (page 89))

Follow the convention in which all installation, entry point, and other external routines are prefixed with the driver name or a distinctive abbreviation. The format is driver_routine, where driver is the driver name and routine is the standard part of the routine name, as in driver_open.

For example, if your company name is Wonderful Software, and you are writing a MUX driver, name the driver wondermux. When it installs the driver, the kernel calls a routine named wondermux_install.

In this chapter, mydriver and skel are used as sample driver names.

Step 2: Choosing System Header Files

A driver header file contains definitions of data structures, macros, and constants that are used to compile the driver module. You can create most of these definitions by including other header files in the driver header file.

This section lists header files the driver might need. To find out which header files the driver requires, see the reference pages in the HP-UX 11i v3 Driver Development Reference and HP-UX Reference for each kernel call and data structure that the driver uses.
NOTE: Because it redefines some entries from other header files, you must specify the <sys/wsio.h> file in the last #include statement in the driver header file.

Header Files for All Drivers

All drivers require the following header files:

- `/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/errorno.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` Needed for acquiring and releasing memory.
- `/usr/include/sys/vm_arena_iface.h` Arena allocator interfaces.
- `/usr/include/sys/kern_svcs.h` Kernel services, such as `bcopy`.
- `/usr/include/sys/spinlock.h` Spinlock services.
- `/usr/include/sys/sysmacros.h` Commonly used fields in some driver minor numbers.
- `/usr/include/sys/uio.h` The `uio` structure and its elements.
- `/usr/include/sys/wsio.h` Data and macros used in the WSIO context, including the `wsio_drv_info` and `wsio_drv_data` structures. Each driver and each WSIO-dependent pseudo driver header must include this header file.

Header Files for Disk Drivers

All disk drivers require the following header files:

- `/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

All tape drivers require the following header files:

- `/usr/include/sys/mtio.h` Flags for the `ioctl` system call for use with magnetic tapes.

Step 3: Defining Installation Structures

Include some configuration data structures in the driver. The data structures the driver requires depend on what the device or interface card driver does. They can appear either in the `.c` file of the driver or in a header file included in the driver. The following structures are available:

- `drv_ops_t` Device driver entry points for device switch tables.
- `drv_info_t` Driver-specific fields 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 driver header files.

**drv_ops_t Structure**

The `drv_ops_t` structure type, defined in `<sys/conf.h>`, contains pointers to a driver's entry points. A `drv_ops_t` structure must be statically allocated. The structure is defined as follows:

```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 */
    d_psize1_t (*d_psize1); /* block */
    int (*d_drv_cb)(drv_cb_opcode_t, uintptr_t); /* block and character */
    int (*reserved3)();   /* NULL */
    int d_flags;          /* block and character */
} drv_ops_t;
```

The relevant fields are described in Table 4-1. All other fields in `drv_ops_t` must be NULL. Except as noted, entry points that do not apply to the driver or that the driver does not provide must be NULL. For example, `d_psize` has no meaning for a printer.

**Table 4-1 driv_ops_t Structure Field Descriptions**

<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 <code>driver_open</code> routine, which enables a device for subsequent operations. If the device is offline, does not exist, or cannot be accessed, the routine must return an error. When appropriate, the open routine can do nothing.</td>
</tr>
<tr>
<td>d_close</td>
<td>Both</td>
<td>Pointer to the <code>driver_close</code> routine, which disables interrupts, resets a device, frees resources, and performs other tasks required when a device closes.</td>
</tr>
<tr>
<td>d_strategy</td>
<td>Block</td>
<td>Pointer to the <code>driver_strategy</code> routine, which queues I/O requests for either reading or writing. Drivers of character devices often call <code>physio</code> from their read and write routines; <code>physio</code> 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 <code>driver_psize</code> routine. For a swapping device, the routine must return the size of the swap partition. Consider writing this routine only if the device is used for swapping.</td>
</tr>
<tr>
<td>d_read</td>
<td>Character</td>
<td>Pointer to the <code>driver_read</code> routine, which returns the requested data transferred from the device.</td>
</tr>
<tr>
<td>d_write</td>
<td>Character</td>
<td>Pointer to the <code>driver_write</code> routine, which writes the requested data to the device.</td>
</tr>
<tr>
<td>d_ioctl</td>
<td>Character</td>
<td>Pointer to the <code>driver_ioctl</code> routine, which sends control information to, or gets it from, a device. 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 <code>driver_select</code> routine, which tests for I/O completion and driver-dependent exception conditions. If the device is always ready for reading or writing, put <code>seltrue</code> in the <code>d_select</code> field. If set, calls to <code>select</code> always return true without invoking the driver.</td>
</tr>
</tbody>
</table>
### Table 4-1 drv_ops_t Structure Field Descriptions (continued)

<table>
<thead>
<tr>
<th>Field</th>
<th>Device Type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<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 these disks. For other conditions, set this field to NULL; it is ignored.</td>
</tr>
<tr>
<td>d_psize1</td>
<td>Block;</td>
<td>For a given minor number, it returns the 64-bit value of the partition size. The current d_psize_t is maintained for backward compatibility, but only returns a 32-bit value. This entry point is defined as follows in conf.h: int (*d_psize1_t) (dev_t dev, int64_t *size); The difference between this and the current d_psize_t entry point is that the former returns the size in the size parameter and the latter returns size as the function return value.</td>
</tr>
</tbody>
</table>
| d_drv_cb  | Both        | The driver callback routine is a new entry point into the driver. It is called with an opcode. Currently, the following opcodes are defined:  
• CB_DEV_2_NODE — Return the associated I/O tree node handle for a specified dev_t.  
• CB_GET_OPTS — Return the associated device-specific options for a specified dev_t.  
For more information, see “Writing a driver_callback Routine” (page 115). |
| d_flags  | Both        | 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 the following:  
C_ALLCLOSes — Forces a call to driver_close every time the device closes. The default action is to call the driver close routine only on the last close of the device.  
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.  
C_MGR_IS_MP — Identifies the driver as safe to use in a multiprocessing environment.  
C_MAP_BUFFER_TO_KERNEL — Indicates the device driver needs physio to remap a user buffer to kernel space prior to calling the driver_strategy routine.  
C_OPAQ_DEV — Indicates that the device driver’s minor number is opaque. The minor number does not encode any device information such as controller instance, target, LUN or device options. Drivers defining this flag should register for the CB_DEV_2_NODE and CB_GET_OPTS callbacks. This flag is mandatory for all mass storage interface drivers. |

### drv_info_t Structure

All CDIOs use the driver-specific fields in the drv_info_t structure type defined in `<sys/conf.h>`. A drv_info_t structure must be statically allocated. The structure is defined as follows:

```c
typedef struct drv_info {
    char    *name;
    char    *class;
    ubit32  flags;
    int     b_major;
    int     c_major;
    cdio_t  *cdio;
} drv_info_t;
```
Table 4-2 describes the relevant fields. All other fields in `drv_info_t` must be NULL.

<table>
<thead>
<tr>
<th>Field</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Pointer to a string containing the name of the driver. This is the name defined in “Step 1: Choosing a Driver Name” (page 77).</td>
</tr>
<tr>
<td>class</td>
<td>Pointer to a string containing the name of the driver device class (for example, disk). For interface drivers, instances of a card are enumerated within each class as they are identified by the kernel at boot time. Instance numbers are unique in a device class.</td>
</tr>
<tr>
<td>flags</td>
<td>The bitwise OR of flag values that describe the driver.</td>
</tr>
<tr>
<td></td>
<td>DRV_CHAR</td>
</tr>
<tr>
<td></td>
<td>DRV_BLOCK</td>
</tr>
<tr>
<td></td>
<td>DRV_PSEUDO</td>
</tr>
<tr>
<td></td>
<td>DRV_SCAN</td>
</tr>
<tr>
<td></td>
<td>DRV_MP_SAFE</td>
</tr>
<tr>
<td></td>
<td>DRV_SAVE_CONF</td>
</tr>
<tr>
<td>b_major</td>
<td>The major number if this is a block device. Set it to -1 for dynamic assignment or if this is not a block device. Drivers must ask for a dynamic major number. If DRV_BLOCK is not set in flags, no block major number is assigned to the driver.</td>
</tr>
<tr>
<td>c_major</td>
<td>The major number if this is a character device. Set it to -1 for dynamic assignment or if this is not a character device. Drivers must ask for a dynamic major number. If DRV_CHAR is not set in flags, no character major number is assigned to the driver.</td>
</tr>
</tbody>
</table>

wsio_drv_data_t Structure

The `wsio_drv_data_t` structure type, defined in `<sys/wsio.h>`, contains driver-specific fields for WSIO drivers. The structure is defined as follows:

```c
typedef struct wsio_drv_data {
    char    *drv_path;
    sbit8   drv_type;
    ubit32  drv_flags;
    int     (*drv_minor_build)();
    int     (*drv_minor_decode)();
    int     (*drv_get_minors)();
    int     (*io_path_mgr)();
} wsio_drv_data_t;
```

Table 4-3 `wsio_drv_data_t` Structure Field Descriptions

<table>
<thead>
<tr>
<th>Field</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>drv_path</code></td>
<td>For device drivers, <code>drv_path</code> is typically a string that contains the interface card type and the device class. For example, <code>scsi_disk</code>. For interface drivers, <code>drv_path</code> must match the card type. For example, <code>scsi</code> instead of <code>ext_bus</code>. For pseudo drivers, <code>drv_path</code> must match the card class. For example, <code>graphics</code>.</td>
</tr>
<tr>
<td><code>drv_type</code></td>
<td>One of the following values:</td>
</tr>
<tr>
<td></td>
<td>TDEVICE</td>
</tr>
<tr>
<td></td>
<td>TINTERFACE</td>
</tr>
</tbody>
</table>
### Table 4-3 wsio_drv_data_t Structure Field Descriptions (continued)

<table>
<thead>
<tr>
<th>Field</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>drv_flags</code></td>
<td>One of the following values:</td>
</tr>
<tr>
<td></td>
<td><code>DRV_CONVERGED</code></td>
</tr>
<tr>
<td></td>
<td><code>NOT_CONVERGED</code></td>
</tr>
<tr>
<td><code>drv_minor_build</code></td>
<td>Pointer to the minor number formatter.</td>
</tr>
<tr>
<td><code>drv_minor_decode</code></td>
<td>Pointer to the minor number interpreter.</td>
</tr>
<tr>
<td><code>drv_get_minors</code></td>
<td>Set this field to <code>NULL</code>.</td>
</tr>
<tr>
<td><code>io_path_mgr</code></td>
<td>Set this field to <code>NULL</code>.</td>
</tr>
</tbody>
</table>

### The wsio_drv_info_t Structure Type

The `wsio_drv_info_t` structure type, defined in `<sys/wsio.h>`, contains pointers to the three preceding data structures. The last field is a driver version. Use it to indicate different versions of your driver. Drivers can use the `WSIO_DRV_CURRENT_VERSION` define to set the value of this field. The structure is defined as follows:

```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

Following 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,
        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 other than configuration and interrupt. However, you can write 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. A driver can be both an interface driver and a device driver (monolithic driver). In that case, it has both standard entry points and an ISC structure, and so forth.

The following is an sample 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",
    DRV_SAVE_CONF | DRV_MP_SAFE | DRV_SCAN,
    NULL,
    NULL,
};
### Sample Header for a Pseudo Driver

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

If not installed in the WSIO CDIO, the pseudo driver must define the following header and structures and use the `install_driver` installation function in `driver_install`. See “Writing a driver_install Routine” (page 89).

#### NOTE:  This header uses 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,  
  NULL,  
  NULL,  
  NULL,  
};
```
In the WSIO case, the driver adds the following header and structure to the preceding header and uses the `wsio_install_driver` installation function in `driver_install`. See “Writing a driver_install Routine” (page 89).

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

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

static wsio_drv_info_t my_wsio_info =
{
    &my_info,
    &my_ops,
    &my_data,
    WSIO_DRV_CURRENT_VERSION
};
```

Step 4: Identifying Routines for the Driver

Table 4-4 lists the configuration, entry point, and other routines for different driver types.

**Table 4-4 Configuration and Entry Point Routines**

<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><code>driver_install</code></td>
<td><code>driver_open</code></td>
<td>None</td>
</tr>
<tr>
<td></td>
<td><code>driver_dev_init</code></td>
<td><code>driver_close</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>driver_strategy</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>driver_psize</code></td>
<td></td>
</tr>
<tr>
<td>Block Pseudo</td>
<td><code>driver_install</code></td>
<td><code>driver_open</code></td>
<td>None</td>
</tr>
<tr>
<td></td>
<td><code>driver_dev_init</code></td>
<td><code>driver_close</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>driver_strategy</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>driver_psize</code></td>
<td></td>
</tr>
<tr>
<td>Character Device</td>
<td><code>driver_install</code></td>
<td><code>driver_open</code></td>
<td><code>driver_strategy</code></td>
</tr>
<tr>
<td></td>
<td><code>driver_dev_init</code></td>
<td><code>driver_close</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>driver_read</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>driver_write</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>driver_ioctl</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>driver_select</code></td>
<td></td>
</tr>
<tr>
<td>Character Monolithic</td>
<td><code>driver_install</code></td>
<td><code>driver_open</code></td>
<td><code>driver_isr</code></td>
</tr>
<tr>
<td></td>
<td><code>driver_attach</code></td>
<td><code>driver_close</code></td>
<td><code>driver_strategy</code></td>
</tr>
<tr>
<td></td>
<td><code>driver_dev_init</code></td>
<td><code>driver_read</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>driver_if_init</code></td>
<td><code>driver_write</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>driver_probe</code></td>
<td><code>driver_ioctl</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>driver_select</code></td>
<td></td>
</tr>
</tbody>
</table>
Table 4-4 Configuration and Entry Point Routines (continued)

<table>
<thead>
<tr>
<th>Driver Type</th>
<th>Configuration Routines</th>
<th>Entry Points</th>
<th>Other Routines</th>
</tr>
</thead>
</table>
| Character        | driver_install driver_dev_init       | driver_open driver_open
| Pseudo           |                                       | driver_close driver_close
|                  |                                       | driver_read driver_read
|                  |                                       | driver_write driver_write
|                  |                                       | driver_ioctl driver_ioctl
|                  |                                       | driver_select driver_select |
|                  |                                       | driver_strategy      |                   |
| Interface        | driver_install driver_attach         | None ^3              | driver_isr        |
|                  | driver_if_init                      |                      |                   |
|                  | driver_probe                        |                      |                   |

1 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.
2 In a combined block and character driver, the two driver_strategy routines can be combined.
3 An interface driver does not have system-called entry points. It can provide a driver-dependent interface through isc->ifsw, as described in “Writing a driver_attach Routine” (page 91).

The driver might not need all the routines shown in Table 4-4. Choose the routines you need from the table.

Except for driver_install, select any name for each of these routines. For convenience in maintenance and debugging, HP recommends that you use the names shown, substituting your driver name for driver. The driver_install routine must be named as shown, with the your driver name substituted for driver.

You can combine block and character types into the same driver. In this case, the driver can specify the entry points for both types. See “Entry Points for a Block Driver” (page 87).

Configuration Routines

Configuration routines execute when the system boots. Use the following guidelines when choosing configuration routines:

- 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.
- The driver_probe routine 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

Entry point routines are the interfaces 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 is not asked to, it does not need the corresponding routine. For example, a printer often has no need for a read routine.

Table 4-5 lists the system calls and corresponding entry points for a block driver.
Table 4-5 Entry Points for a Block Driver

<table>
<thead>
<tr>
<th>System Call</th>
<th>Routine Executed</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>close</td>
<td>driver_close</td>
<td>Closes the device.</td>
</tr>
<tr>
<td>kernel invokes the d_psize entry point.</td>
<td>driver_psize</td>
<td>Specifies swap partition size.</td>
</tr>
<tr>
<td>open</td>
<td>driver_open</td>
<td>Opens the device.</td>
</tr>
<tr>
<td>read</td>
<td>driver_strategy</td>
<td>Performs a block read.</td>
</tr>
<tr>
<td>write</td>
<td>driver_strategy</td>
<td>Performs a block write.</td>
</tr>
</tbody>
</table>

Table 4-6 lists the system calls and corresponding entry points for a character driver.

Table 4-6 Entry Points for a Character Driver

<table>
<thead>
<tr>
<th>System Call</th>
<th>Routine Executed</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>close</td>
<td>driver_close</td>
<td>Closes the device.</td>
</tr>
<tr>
<td>ioctl</td>
<td>driver_ioctl</td>
<td>Performs a special command.</td>
</tr>
<tr>
<td>open</td>
<td>driver_open</td>
<td>Opens the device.</td>
</tr>
<tr>
<td>read</td>
<td>driver_read</td>
<td>Performs a character read.</td>
</tr>
<tr>
<td>select</td>
<td>driver_select</td>
<td>Tests I/O completion.</td>
</tr>
<tr>
<td>write</td>
<td>driver_write</td>
<td>Performs a character write.</td>
</tr>
</tbody>
</table>

Interface drivers have no entry points because they are called only by device drivers, not from user programs.

Other Routines

The other routines in Table 4-4 are not defined in the header structures. Instead, they are defined within the driver and passed as parameters to other routines. The following is a list of these routines:

- The `driver_isr` is the interrupt service routine for a device or interface, established by the interface driver.
- The `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 be combined.
- The `driver_minphys` is used for a character driver that calls `physio` and requires that subsequent `driver_strategy` calls be made with different size buffers than that provided by the standard `minphys` routine.

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_dev_probe`
The following steps describe the order of events that the kernel follows to install and initialize all drivers at boot time (system start or power-on time). The steps illustrate configuration routines for four types of drivers: device, interface, pseudo, and monolithic, 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 must 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. The `ifd_install` routine puts `ifd_attach` on the applicable chains of attach routines for supported interfaces. For example, the `pci_attach` routines for a driver that supports the PCI bus interface cards. If the install routine has an `ifd_probe` routine, it registers the routine with the system.
   e. The `monod_install` routine puts `monod_attach` on the applicable chain of `attach` routines. If the routine has a `monod_dev_init` routine, it puts the routine on a chain of `init` routines. If the routine has a `monod_probe` routine, it registers the routine with the system.

2. The kernel installs all the drivers.

   The driver now has a dynamically assigned major number.

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 calls the next entry on the chain.

   When called, `ifd_attach` or `monod_attach` claims 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 with a valid function call in its `isc->gfsw->init` structure (set up in step 3), it calls that function.

5. The `init` chain runs. For each device, pseudo, or monolithic driver that specified one, the `devd_dev_init`, `pseudod_dev_init`, or `monod_dev_init` routine is called once. Each entry on the chain calls the next entry on the chain.

   Because 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.

The following sections describe how to write the various configuration routines and include return values, diagnostics, and examples.
Writing a driver_install Routine

You provide the driver_install function. The name must be in the format shown, with driver replaced by the name of the driver as it is specified in the module metadata file.

The driver_install routine does the following tasks:

• It calls a driver installation service to register the driver with the system and to fill out system structures and tables.
  
  Use wsio_install_driver for WSIO installation. (See wsio_install_driver(9F) in the HP-UX Device Driver Reference.) Use install_driver for installation outside of WSIO, typically with certain pseudo drivers (see install_driver(9F)).

• For an interface or monolithic driver, driver_install puts the driver_attach routine at the head of a global attach chain. After 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 put 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(9F) and wsio_register_addr_probe(9F).

• 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 puts its driver_dev_init routine at the head of the chain. The chain is processed after all the drivers have been configured. The head of the global init chain is pointed to by dev_init.

Return Values

Beginning with HP-UX 11i v3, the driver_install routine has a void return type. Therefore, it cannot return any values.

If it fails, it can print error messages on the system console before returning to the caller.

Device and WSIO Pseudo Driver Example

The following example shows the 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_saved_dev_init)();
/* to save previous init head */
int (*skel_saved_del_init)();

void skel_install(void)
{
    /* Head pointer for init chain */
    extern int (*dev_init)();

    /* Save head pointer*/
    skel_saved_dev_init = dev_init;
    /* Make my dev_init the head */
    dev_init = skel_dev_init;

    /* Register driver with WSIO */
    wsio_install_driver(&skel_wsio_info);
    return;
}
```
Non-WSIO Pseudo Driver Example

The following example shows the 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
/* save previous init head */
int (*pseu_saved_dev_init)();

void pseu_install(void)
{
    /* Head pointer for init chain */
    extern int (*dev_init)();

    /* Save head pointer */
    pseu_saved_dev_init = dev_init;
    /* Make my dev_init the head */
    dev_init = pseu_dev_init;

    /* Register driver */
    install_driver(&pseu_drv_info, &pseu_drv_ops);
    return;
}
```

Interface Driver Example

The following example shows the 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_saved_attach)(); /* save previous attach head */

void skel_install(void)
{
    /* Head of pci attach chain */
    extern int (*pci_attach)();

    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 */
    wsio_install_driver(&skel_wsio_info);
    return;
}
```

Monolithic Driver Example

The following example shows the 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 */

void skel_install(void)
{
    /* Pointer to pci attach chain */
    extern int (*pci_attach)();
```
Writing a driver_attach Routine

You provide the driver_attach function. 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. Typically, driver is replaced by the 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 (for example, pci_attach) 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. Optionally, it 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.

Return Value

Each driver_attach routine must return the value returned by the next driver_attach routine in the chain. The end-of-chain function returns a unique completion code.

Diagnostics

The driver_attach routine can signal an error as follows:

- The card is faulty. If the INIT_ERROR flag is set in isc->if_info->flags, the kernel displays the following message:
  
  init of hardware not successful

  The driver_attach routine has access to the card registers through the isc pointer passed into it. The routine can verify that data.

- Bad driver. A driver_attach routine returned without calling the next driver_attach routine in the chain. The system panics and displays the following message:

  bad driver in kernel

Examples

static int
skel_attach(int product_id, struct isc_table_type *isc_ptr)
{
    /* the four-byte value for this driver */
int MY_PRODUCT_ID = 0x12345678;

/* 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;
*/
return (*skel_saved_attach)(product_id, isc_ptr);

Writing a driver_dev_init Routine

You provide the driver_dev_init routine. It can have any unique name. Pass the name to
WSIO Services by linking it into an init chain from the driver_install routine.

Writing a driver_if_init Routine

You provide the driver_if_init routine. It can have any unique name. Pass the name to
WSIO Services by specifying it in the gfsw structure during driver_attach.
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 card's line based interrupt */
    status = wsio_intr_alloc(isc, mydriver_isr,
                            (uintptr_t)isc, 0, &my_intr_obj);
    if(status != WSIO_OK)
        return(ERROR);
    status = wsio_intr_set_irq_line(isc, my_intr_obj,
                                    WSIO_IRQ_LINE_AUTO, 0);
    if(status != WSIO_OK)
        return(ERROR);
    status = wsio_intr_activate(isc, my_intr_obj);
    if(status != WSIO_OK)
        return(ERROR);
    ...
    /* successfully initialized the interface */
    return(0);
}

Writing Driver Probe Routines

Drivers can register probe functions that are used by the WSIO to scan for devices attached to 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 is found.
PROBE_ADDRESS Look for a device at the specified hardware address.

You can use the following WSIO services to register driver probe functions:

wsio_register_dev_probe Most drivers only use the wsio_register_dev_probe service to register a probe function based on either the class of devices the driver controls or the driver name as specified in the name field of the dev_init structure.

wsio_register_addr_probe The wsio_register_addr_probe service associates an additional probe function based on the name of the driver, as specified in the driver's drv_info_t structure.

The following sections describe each WSIO service.

wsio_register_dev_probe Routine

The calling semantics for wsio_register_dev_probe are as follows:

int wsio_register_dev_probe(
    u_int type,
    int (*func()),
    char *str

Where:

type Either IF_CLASS or DRV_NAME.

func A pointer to the driver probe function.

str A pointer to an ASCII string.

The wsio_register_dev_probe service is used to register a probe function for a driver based on either the driver's class or name. If the driver specifies a 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 a 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. In the following example, the driver specifies a class type probe function:

```c
void sctl_install(void)
{
    /*
     * Register the SCSI probe function
     * with the WSIO CDIO.
     */
    wsio_register_dev_probe(IF_CLASS, scsi_probe, "scsi");

    /*
     * Register the driver with WSIO.
     */
    wsio_install_driver_(&sctl_wsio_info);
    return;
}
```

The WSIO saves a pointer to the probe function and later retrieves it when probing for devices underneath an interface card. When the WSIO associates a probe function with an interface card, it first tries to find a probe function based on the driver's name. If one is not found, it looks 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 *handle,
    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
);
```

Table 4-7 shows the device probe parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>IN/OUT</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>desc</td>
<td>OUT</td>
<td>A pointer to a string with the device description. This is driver-dependent.</td>
</tr>
<tr>
<td>drv_info</td>
<td>IN</td>
<td>A pointer to the drv_info_t structure.</td>
</tr>
<tr>
<td>hw_path</td>
<td>IN</td>
<td>A pointer to the hardware path of the last device found.</td>
</tr>
<tr>
<td>isc</td>
<td>IN</td>
<td>A pointer to the isc_table_type structure.</td>
</tr>
</tbody>
</table>
Table 4-7 Device Probe Parameters (continued)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>IN/OUT</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>OUT</td>
<td>A pointer to a string initialized with the device's name, such as <code>scsi_disk</code>. This information is used to match the device to a driver based on the information in the <code>drv_path</code>.</td>
</tr>
<tr>
<td>handle</td>
<td>IN</td>
<td>A pointer to an internal GIO structure.</td>
</tr>
<tr>
<td>probe_id</td>
<td>OUT</td>
<td>A pointer to a unique identifier for the device found.</td>
</tr>
<tr>
<td>probe_type</td>
<td>IN</td>
<td>The type of hardware probe: FIRST, NEXT, or ADDRESS.</td>
</tr>
</tbody>
</table>

`wsio_register_addr_probe` Routine

Drivers can register an additional address probe function by calling the `wsio_register_addr_probe` WSIO Service. Use this service 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. For example:

```c
void c720_install(void)
{
    /**
     * Register the driver with WSIO.
     * If it succeeds, add our attach function
     * to the PCI attach list and register the
     * `parallel_scsi_probe` function for the address probe.
     */
    if (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;
}
```

Driver address probe functions registered by `wsio_register_addr_probe` must have the following prototype:

```c
int drv_addr_probe(
    void *handle,
    void (*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
);
```

Table 4-8 shows the address probe parameters.

Table 4-8 Address Probe Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>IN/OUT</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>desc</td>
<td>OUT</td>
<td>A pointer to a string with the device description. This is driver-dependent.</td>
</tr>
<tr>
<td>dev_probe</td>
<td>IN</td>
<td>A pointer to a class probe function, if one exists. Else, NULL.</td>
</tr>
</tbody>
</table>
Table 4-8 Address Probe Parameters (continued)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>IN/OUT</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>drv_info</td>
<td>IN</td>
<td>A pointer to the drv_info_t structure.</td>
</tr>
<tr>
<td>hw_path</td>
<td>OUT</td>
<td>A pointer to the hardware path of the next device found.</td>
</tr>
<tr>
<td>isc</td>
<td>IN</td>
<td>A pointer to the isc_table_type structure.</td>
</tr>
<tr>
<td>name</td>
<td>OUT</td>
<td>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.</td>
</tr>
<tr>
<td>handle</td>
<td>IN</td>
<td>A pointer to a GIO structure.</td>
</tr>
<tr>
<td>probe_id</td>
<td>OUT</td>
<td>A pointer to a unique identifier for the device found.</td>
</tr>
<tr>
<td>probe_type</td>
<td>IN</td>
<td>The type of hardware probe. Three types are defined.</td>
</tr>
</tbody>
</table>

The only difference between the calling interface of a probe function registered by `wsio_register_dev_probe` and one registered by `wsio_register_addr_probe` is that the latter has an additional parameter, `(*dev_probe)()`. This enables the two types of probe function to be used together by some driver stacks.

Typically, a driver registers a single probe function by calling `wsio_register_dev_probe` to register a probe function based on either the driver’s name or class. This is the case for monolithic drivers. However, there are some driver stacks that are modular; a device driver can be paired with several different interface drivers with each interface driver supporting a different range of addresses. In this case, use the two WSIO services to register separate probe functions that work together.

For example, the driver install routine of the SCSI interface driver `scsi_c720` registered an address probe function by calling `wsio_register_addr_probe`. The `scsiCtl.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 uses both probe functions by calling the address probe function, passing in a pointer to the class probe function. The driver address probe function sets up the address for the next device to be probed, then calls the class probe function to talk to the devices underneath.

When a driver class probe function is used with an address probe function and not called directly by the WSIO CDIO probe code, it does not need to adhere to the WSIO calling semantics described for class probe functions. The driver stack can define its own parameter list for the class probe function. In this case, never use the class probe function as a standalone probe function 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`. The `scsiCtl.c` device driver registers a class probe function called `scsi_probe`. The two probe functions cooperate, and the interface to the class probe function `scsi_probe` does not adhere to the WSIO CDIO specification.
Example 4-1 Register Address Probe Example

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

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

/*@ 
* 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, then calls the scsi_ctl registered probe 
* function, probe_func = scsi_probe(), 
* which 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 must contain two 
     * elements at the most, a target and a 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);

    ... 
}
```

Step 5: Writing Configuration Routines 97
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. The next thing to look for
     * is the first lun of that target. 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 did not 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++;
            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];
        }

    break;

    } while(false);
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 have not run out of valid targets. Either
* we are looking for the next target and we have not
* 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 4-2 scsi_probe Routine Example

/*
 * 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,
};

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

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 stating 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 are dealing with
     * a 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 the 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.
* Make sure that it is also a valid LUN.
*/
if (iqr_data.inq2.periph_qualifier == 0)
    found = VALID_LUN;
} /* dev_type != NO_SCSI_DEV */
} /* sctl_ioctl */
sctl_close(dev);
Writing a driver_minor_build Routine

The driver_minor_build routine is used when the driver has a special method for building minor numbers. 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 and 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;
}
```

The driver_minor_build routine is called through the I/O command shared library, if one provided by the driver. For more information on writing an I/O command shared library, see Chapter 19 (page 445). Drivers that do not provide an I/O command shared library do not need to provide a driver_minor_build routine.

Step 6: Writing Entry Point Routines

Most drivers have defined entry point routines, which are driver routines 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 described in the “Step 7: Writing Other Driver Routines” (page 123). To determine applicable routines for the driver, see “Step 4: Identifying Routines for the Driver” (page 85).

Writing a driver_open Routine

You must provide the driver_open routine to prepare a device for I/O. It can have any unique name. Pass the name to WSIO services by specifying it in the d_open field of the drv_ops structure. Replace driver with 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. Next it calls the corresponding driver_open routine and passes control to the driver routine defined in the drv_ops_t structure.
The \texttt{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 \texttt{dev_t} value passed to it.)
- Sets a flag indicating that 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 succeeds.

The device driver must implement one of the following types of \texttt{open} required by the device:

\textbf{Exclusive Open} \hspace{1cm} Opening a device exclusively allows only one process at a time to access the device. Magnetic tapes and printers are exclusive-open 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 can open a device, the \texttt{driver_open} routine must return an error whenever it executes and finds the device is already opened.

\textbf{Shared Open} \hspace{1cm} Devices that allow more than one process at a time to access them can be shared opened. Terminals are typically shared open devices, so users can communicate with each other using the \texttt{write} command. 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.

\textbf{Multiple Open} \hspace{1cm} 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 enables each process to modify the data structures' values independently.

Decide which type of \texttt{open} routine you need and add appropriate code to the routine. See the following skeleton routines. Also see the “Device Driver Example” (page 104).

\begin{verbatim}
int driver_open(
    dev_t dev,
    int flag
);\end{verbatim}

Where:

\textbf{dev} \hspace{1cm} The \texttt{dev_t} device number of the file to be opened. The \texttt{driver_open} routine can extract the major and minor numbers from the device number. See \texttt{major(9F)} and \texttt{minor(9F)} in the HP-UX 11i v3 Driver Development Reference.

\textbf{flag} \hspace{1cm} A value corresponding to the \texttt{oflag} parameter of the \texttt{open} system call. The kernel executes the \texttt{oflag} functions (described in \texttt{fcntl(5)} and \texttt{open(2)}) before it calls the driver. Therefore, the driver can usually ignore these flags. The kernel translates the O\_xxxx values into corresponding Fxxxx values, which it passes to the \texttt{driver_open} routine. The flags of possible interest to the driver include \texttt{FREAD}, \texttt{FWRITE}, \texttt{FNDELAY}, and \texttt{FEXCL}. 

---

Step 6: Writing Entry Point Routines 103
For example, the `driver_open` routine for a magnetic tape 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.

For example, if the kernel calls a terminal driver's `driver_open` routine and `FNDELAY` is set, the routine does not wait for the hardware to respond before returning control to the `open` system call. For more information, see `fcntl(5)`, `open(2)`, and `termio(7)`.

**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 must discover this error.

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

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`. See `open(2)` for the following expected error names:

- The device is offline.
- 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.

**Device Driver Example**

The following 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;

    /* Initialize the device (call the interface
     * driver). If initialization 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

You must provide a `driver_close` routine. 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. Replace `driver` with the driver's name.

The `driver_close` routine 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
);
```

Where:

- **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(9F)` and `minor(9F)` in the `HP-UX 11i v3 Driver Development Reference`.

- **flag** A value corresponding to the `flag` field in the `driver_open` call. For values that can appear in the `flag` parameter, see the parameter description in “Writing a driver_open Routine” (page 102).

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), force the `driver_close` routine to return an integer value (such as 0).

### Exclusive-Open Devices

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:

- Completes all I/O in progress.
- Releases data structures.
- Clears the driver's `open` flag set by `driver_open`, indicating that the device is closed.
- 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 this driver releases data structures, those structures are the ones allocated by the driver when the device was opened.

### Shared-Open or Multiple-Open Devices

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 can 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:

- Completes all I/O in progress.
- Releases data structures.
• Clears the open flag, indicating that the device is closed.
• 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 must 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. For instance, it will not call the driver_close routine when a forked child process calls close for an open file descriptor it has inherited.

### Using the C_ALLCLOSES Flag

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

The kernel maintains a count of opens and closes 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 enables tracking of complex device structures, such as a device with two device files that have the same major number.

**NOTE:** When file descriptors created by fork or dup are closed, the driver_close routine is not called, regardless of the C_ALLCLOSES flag.

### Sample driver_close Routine

The following example is for a character driver that controls a Centronics interface. For the corresponding driver_open routine, see “Device Driver Example” (page 104).

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

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

    /* Invoke the interface driver shutdown routine */
    wsio_get_isc(dev, &isc, &CharDrv_wsio_info);
```

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

/* Release the device */
DevIsOpen--;
return 0;

Writing driver_read and driver_write Routines

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. A write transaction moves data from processor memory to a device. You must provide driver_read and driver_write routines. 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. Replace driver with 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 lot of code, combine common code 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 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, then call the physio kernel routine, which calls the driver_strategy routine.

The driver_read and driver_write routines can also process requests using uiomove.

You can implement a driver_read routine in either of the following ways:

- Call physio with the appropriate parameters, enabling the driver_strategy routine to complete the request. If you use physio, you must write a driver_strategy routine. The driver_strategy routine is passed as a parameter to physio. For more information about using physio in driver_read, see “Using the physio Routine” (page 108).

- Use uiomove to buffer the data, then to complete the request. If you use uiomove, the driver_read routine must do the following:
  - Initialize data structures.
  - Set a flag indicating that I/O is in progress.
  - Request an I/O operation from the device.
  - Wait or sleep while the device completes the I/O operation.
  - Call uiomove to transfer the data from the kernel's buffer to the user's buffer.
  - Return a value to the read or readv call.

For more information about using uiomove in driver_read, see “Using the uiomove Routine” (page 109).
You can implement a `driver_write` routine in either of the following ways:

- Use `physio` and a `driver_strategy` routine. For more information about using `physio` in `driver_write`, see “Using the physio Routine” (page 108).
- Use `uiomove`, in which case the `driver_write` routine must do the following:
  - Initialize data structures.
  - Call `uiomove` to copy the data into kernel space.
  - Set a flag indicating that I/O is in progress.
  - Request that the device start the I/O operation.
  - Wait or sleep while the device completes the I/O operation.
  - Return a value.

For more information about using `uiomove` in `driver_write`, see “Using the `uiomove` Routine” (page 109).

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

The function prototypes for `driver_read` and `driver_write` are as follows:

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

Where:

- `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 must verify that the minor number is valid before accessing the kernel’s data structures. See the Parameter section in “Writing a driver_open Routine” (page 102).
- `uiop` - A pointer to a `uio` structure. The `uio` structure contains information about the data being read or written. See “Step 2: Choosing System Header Files” (page 77).

The `driver_read` and `driver_write` routines are expected to return the following values:

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

Using the physio Routine

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 the following declaration:

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

Where:
The `physio` routine that sets up an I/O request.

**bp** Pointer to a `buf` structure. If the pointer is `NULL`, `physio` allocates a `buf` structure from the file system buffer cache.

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

**dev** Device number.

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

**mincnt** 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 is no larger than the system’s maximum size. Use the kernel’s `minphys` routine, which most drivers use, or write one. The `physio` routine sends the requests to the routine specified in `strat`.

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

The `physio` routine also handles the I/O transfer for the `driver_read` and `driver_write` routines.

The `physio` routine locks the user’s data area so that it cannot be swapped out during the transfer. 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. When the transfer is complete, `physio` awakens when the driver sets the `b_done` flag in the buffer header and calls `biodone(b)`.

For more information, see `physio(9F)` in the *HP-UX 11i v3 Driver Development Reference*.

Example 4-3 shows a `skel_read` routine using `physio()`.

**Example 4-3 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 the `uiomove` Routine

The `uiomove` routine moves data from one address space to another. Use `uiomove` if the 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
);

Where:

cp Pointer to kernel space to hold data.

n Number of bytes to transfer.

flag Transfer direction, UIO_READ or UIO_WRITE.

uiop Pointer to the uio structure for the transfer.

To write a routine using uiomove, follow these guidelines:

• 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.

• For a driver to buffer data between the device and the user buffer, use geteblk to get an empty buffer and a buffer header from the kernel’s buffer cache.

• The geteblk routine allocates a buffer from the file system 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 must release the buf structure and the buffer it obtained from geteblk using the kernel’s brelse routine.

• When a device driver gets a buffer using geteblk, it is borrowing a buffer that is otherwise used by the file system to cache data. This means a device driver that allocates buffers indiscriminately using geteblk can affect system performance.

For more information, see uiomove(9F) in the HP-UX 11i v3 Driver Development Reference.

Writing a driver_ioctl Routine

The driver_ioctl entry point routine executes driver-specific control functions. You must provide the driver_ioctl routine. 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. Replace driver with the driver’s name. The function prototype is as follows:

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

Where:

dev Device number of the associated device.

cmd Command word. See “Command Words” (page 111).

data Pointer to the command arguments, if any.

flag File access flags. Most drivers ignore this parameter.

ioct1 System Call

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

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

The *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
);```

Where:

- `fildes` - File descriptor obtained from an `open` or `dup` call made earlier.
- `request` - 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. See “Command Words.”
- `arg` - The type and value of `arg` are driver-dependent.

You can specify the following requests in the `request` field:

- Requests to be processed by one driver.
- Requests to be processed by multiple drivers.

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 similar tasks. For example, `FIOASYNC`. Upon choosing to implement any of these requests, the driver must 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. For a list of these requests and the standard processing the driver should perform, see `ioctl(5)` in the *HP-UX 11i v3 Driver Development Reference*.

Examine the header files in `/usr/include/sys` for examples of `ioctl` command definitions. The command `grep 'define.*_IO' /usr/include/sys/* | more` displays a large list of the commands used by many device drivers.

**Command Words**

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 the driver and must include this file.

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

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 copies the data structure to which `arg` points between the user address space and the kernel address space. The driver sees `arg` as a pointer to a kernel buffer.

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

Where:

- `_IO` Indicates that the `ioctl` command does not pass in an argument.
- `_IOR` Reads 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-specifed buffer.
_IOW  Writes 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.

The parameters are as follows:

command  Identifier assigned to the command.

t  Arbitrary character of choice, used to associate the ioctl call with the driver. Use the command grep 'define.*_IO' /usr/include/sys/* | more to ensure that this value (after macro expansion) does not conflict with another driver.

n  Number (0 to 127) that identifies a driver specific command for the driver.

object  Type of object to which arg points. The object has a 16 KB size limit.

The driver_ioctl entry point routine is expected to return the following values:

0  Successful completion.

<>0  Error. The ioctl function returns the error value to the user process in errno. For a list of standard error values, see ioctl(2), ioctl(5), and errno(2).

LP64 Considerations

The ioctl commands generated by the _IO macro have the same encoded values for both ILP32 and LP64 data models. However, the _IOR, _IOW, and _IOWR macros 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. However, if the size is scalable, HP recommends that you write the driver to accommodate two versions (one for each data model) of the ioctl.

In the following example, the cmd ioctl 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 a 32-bit application */
#endif __LP64__
/* Driver ioctl code snippet */
switch (cmd) {
  case SOME_IOCTL:
    <do SOME_IOCTL processing>
  #ifdef __LP64__
  case _SOME_IOCTL_32:
    <do _SOME_IOCTL_32 processing>
  #endif /* __LP64__ */
}
```

In the following example, a cmd ioctl specifies a structure with a pointer data type as a data member:

```c
/* Public Header File */
#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;
}
```

112 Writing a Driver
To copy the 32-bit application buffer, the driver calls `copyin`. For example:

```c
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 `_IO` needs 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. For example:

```c
/* Public Header File */
#else __LP64__
#define LONG_IOCTL _IO('A',1)
#else
#define LONG_IOCTL _IO('a',1)
#endif /* __LP64__ */

/* Private Header File */
#else __LP64__
#define _LONG_IOCTL_32('a',1)
/* long data from 32 bit app */
#endif /* __LP64__ */
```

Sample driver_ioctl Routine

The `mydevice_ioctl` routine implements the `ioctl` commands defined for the `mydevice` driver.

The public header file (`mydevice.h`) used to define `ioctl` commands is as follows:

```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);
```
The header file defines four commands that the driver executes:

CARD_RESET    Resets the device to its default state.
CARD_STATUS   Returns the contents of the device status register to the user in the reg_value field of the mydevice_ioctl_arg structure.
CARD_CONTROL  Sets or clears the bits in the device control register. In the mydevice_ioctl_arg structure, the reg_value field specifies the bits affected. The location field contains SET or CLEAR to indicate the action the driver takes.
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 must 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.

The user program code segment, which sets a bit in the device control register, is as follows:

```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);
```

The private header file to handle 32-bit applications running on a 64-bit OS is as follows:

```c
#ifndef __LP64__
struct mydevice_ioctl_arg32 {
    char reg_value;
    ptr32_t location;
};
#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
```

The driver code to implement the ioctl commands is as follows.

```c
/* 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,
```
```c
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);
#endif
    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 */
        return(0);
    case CARD_BUFADR_32:
        arg32->location = set_buf_addr(arg32->location);
        return(0);
    #endif
    default;
    return(EINVAL);
} /* switch */

Writing a driver_callback Routine

For the driver callback entry point, the following opcodes are defined:
• CB_DEV_2_NODE
• CB_GET_OPTS

The enumeration that defines these opcodes is as follows:

typedef enum drv_cb_opcode {
    CB_GET_OPTS = 1,
    CB_DEV_2_NODE
} drv_cb_opcode_t;

The driver callback has the following prototype:

int (*d_drv_cb) (drv_cb_opcode_t, uintptr_t *);

The drv_cb_info_t data structure, which is passed as a pointer in the uintptr_t *, is defined as follows in the <sys/conf.h> header file:

struct drv_cb_info {
    dev_t dev;
    int dev_type;
    intptr_t data;
    intptr_t reserved1;
    intptr_t reserved2;
} drv_cb_info_t;

Where:

dev The dev_t to be converted to a node.
dev_type The device type, either character or block.
data On return, the driver callback returns the node corresponding to the given dev_t in the data field.
reserved1 Reserved. Set to NULL.
reserved2 Reserved. Set to NULL.

Following is a code snippet of this driver entry point taken from the Qlisp sample driver.

static int qlisp_drv_cb (drv_cb_opcode_t opcode, uintptr_t *info)
{
    drv_cb_info_t *drv_cbinfo = (drv_cb_info_t *) info;
    qlisp_isc_t *lisc = NULL;

    switch(opcode) {
    case CB_GET_OPTS:
        ...
        return GIO_SUCCESS;

    case CB_DEV_2_NODE:
        ...
        drv_cbinfo->data = (intptr_t) lisc->isc->card_node;
        return GIO_SUCCESS;

    default:
        return GIO_E_NOT_SUPPORTED;
    }
}

Writing a driver_minphys Routine

The driver_minphys routine adjusts a physio transfer count into the size 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 `bp->b_bcount` with whatever transfer size the device requires. If `bp->b_bcount` is larger, then `bp->b_bcount` is set to the device's transfer size. Otherwise, `bp->b_bcount` is unchanged. The `driver_minphys` function prototype is as follows:

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

void driver_minphys(
    struct buf *bp
);
```

Where:

- `bp` Pointer to a `buf` structure.

The `driver_minphys` is passed as the `mincnt` parameter to `physio`. In that case, `physio` calls `driver_minphys` and `physio` tracks partial transfers that occur due to the request size limit imposed by `driver_minphys`.

**Example 4-4 driver_minphys Routine Example**

```c
#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, mydriver_minphys, uiop);
}
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 tests I/O completion on a device. You must provide this routine. 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. Replace `driver` with the 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 must return true (a nonzero value) if its device is always ready for I/O. A character driver that does not have a `driver_select` routine must always return true. To do this, specify the `seltrue` kernel function in the `d_select` field of the `drv_ops` structure. For more information, see the *HP-UX 11i v3 Driver Development Reference*.

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)`.) 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. 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
If multiple threads are waiting on a select condition, the driver must set the `collision` argument to `selwakeup` when the select condition becomes true. For more information, see the *HP-UX 11i v3 Driver Development Reference*.

**NOTE:** In prior releases, when calling `waiting_in_select`, 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. 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.

The `driver_select` routine has the following function prototype:

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

Where:

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

The `driver_select` function returns 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 bitmask 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. The `select` system call 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 `selwakeup`. For the call’s second parameter, see the *HP-UX 11i v3 Driver Development Reference*. A true return (nonzero) indicates the select succeeded. When false is returned, the `select` system call sleeps and waits.
Example 4-5 Sample driver_select Routine

```c
#include <sys/types.h>
#include <sys/param.h>
#include <sys/kthread.h>
#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 = kthread_self();
        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 = kthread_self();
        break;
    }
    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 sample skel_output_ready routine is called when the driver finds that the device is ready to output more characters. The skel_output_ready skeleton routine is as follows:

```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 sample skel_input_ready routine when the device driver has input available. The skel_input_ready skeleton routine, which awakens all processes sleeping for the read condition, is as follows:

/*
 * Wake up any readers when the 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 executes block read or write for character or block devices. You must provide this routine. 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. Replace driver with the driver's name.

The 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, because most of the code is the same for the two methods.

The driver_strategy routine 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 from 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 differences for a character device, such as mapping of the user buffer and allocation of the buf structure. For additional information about writing a driver_strategy routine for a character device, see physio(9F).

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

- Initializes data structures, such as DMA buf headers.
- Adds the I/O request to a queue, if necessary.
- Sets a flag that indicates I/O is in progress.
- Returns to the calling process.
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 might 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 must do the following tasks:

1. If an error occurs, set `B_ERROR` in `b_flags` and put a value in `b_error` in the `buf` structure.
2. Set `b_resid` to indicate the amount of remaining data to be transferred.
3. Awaken the driver's top half by calling `biodone`. See the HP-UX 11i v3 Driver Development Reference.

The `driver_strategy` function prototype is as follows:

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

Where:

- `bp` A pointer to a `buf` structure, which contains all the information 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.

If the `C_MAP_BUFFER_TO_KERNEL` flag was set in the driver `drv_opts_t` structure, the kernel maps the data from the user data area into the kernel 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 and passes 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 must 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 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. See the HP-UX 11i v3 Driver Development Reference.

When the I/O is completed, the lower half of the driver sets `b_resid` to the remaining amount of data 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 buffer to the user 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 must 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 must set bp->b_resid to the remaining amount of data 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 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.

**Sample driver_strategy Routine**

The following code shows a driver routine named skel_strategy, derived from an actual device driver. 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 int
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 */
        if (-cnt)
        {
            bp->b_flags |= B_DONE;
            spinunlock( my_lock );
            return;
        }
    }
}
```

122 Writing a Driver
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 the drv_ops_t structure. Interface drivers typically have driver_attach and driver_isr routines. They can also have driver_if_init and driver_probe routines.

When an interface is shared by multiple device drivers, a linkage method is required between the two types of drivers. Typically this is done using an I/O switch structure, as described in I/O Switch Tables (page 39).

Device drivers can also have driver_dev_init routines. For more information, see “Writing a driver_dev_init Routine” (page 92). This section addresses the driver_isr and driver_psize routines.

This section also addresses interface management and device queue management.

Writing a driver_isr Routine

The driver_isr function handles device interrupts in interrupt context. You must provide this routine. It can have any unique name. Pass the name to WSIO Services by specifying it as a parameter of the wsio_intr_alloc or wsio_msi_assign function executed in the driver_attach or driver_if_init routine. Replace driver with the driver's name. See the HP-UX 11i v3 Driver Development Reference.
In an interface or monolithic driver, the ISR processes interrupts from an interface card and performs the following tasks:

- Stops the interface card from interrupting.
- Determines the reason for the interrupt (if appropriate).
- Takes 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 can be called for interrupts not originating from its device. The ISR must handle this and return a zero to the caller. Otherwise, the ISR returns a one, indicating 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 can block. An ISR has the following interface:

```c
int driver_isr(
    long arg1
);
```

Where:

- `arg1` Driver-defined parameter passed in the call to `wsio_intr_alloc`.

The `driver_isr` returns the following values:

- 0 The card does not belong to this driver.
- 1 This routine handled the interrupt.

### Line-Based and Transaction-Based Interrupts

For a line-based interrupt (LBI) or transaction-based interrupt (TBI), specify a pointer to the ISR using the `wsio_intr_alloc` WSIO service. The service has the following prototype:

```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
);
```

Where:

- `isc` Pointer to the `isc_table` entry that represents the device.
- `driver_isr` The driver’s ISR routine.
- `arg` The argument that is passed into the `driver_isr` when it is called. Typically the driver specifies the `isc` pointer as the argument.
- `flags` The flags to indicate if the driver wants an exclusive or shared interrupt resource. With shared interrupt, a driver's ISR can be called when its device did not interrupt. If this is not acceptable, the driver must set the flag to `WSIO_INTR_EXCLUSIVE`.
- `wsio_intr` Pointer to an interrupt handle that `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.

Additional WSIO interrupt services enable the driver to specify whether it wants to use a LBI or TBI. For example, if the driver uses an LBI it calls the `wsio_intr_set_irq_line` service to specify an LBI. Then it calls `wsio_intr_get_irq_line` to get the IRQ line. If the driver wants to use a TBI, it calls `wsio_intr_set_cpu_spec` to set up a TBI. Then it calls `wsio_intr_get_txn_info` to get the transaction-based address and vector. The underlying platform hardware can limit what type of interrupt a driver can use. Some I/O controllers do not allow interface cards underneath them to use TBIs.
For a message-signaled interrupt (MSI), `driver_if_init` uses the following steps to set up interrupt handling:

1. Call `wsio_msi_capability`, possibly multiple times, to determine the hardware’s ability to support MSI or MSI-X and the maximum number of vectors supported for each. Based on the results, decide between using MSI or MSI-X, and how many vectors to use.

2. Call `wsio_msi_alloc` to allocate an interrupt object of the desired type (MSI or MSI-X) with the desired number of vectors. The `wsio_msi_alloc` service has the following prototype:

   ```c
   wsio_ret_code_t wsio_msi_alloc(
       struct isc_table_type *isc,
       wsio_msi_type_t type,
       int vector_cnt,
       wsio_msi_hints_t hints,
       wsio_msi_hndl_t *hndl
   );
   ```

   Where:
   - `isc` A pointer to the `isc_table` entry that represents the device.
   - `type` The type of MSI object requested, either `WSIO_MSI_TYPE` or `WSIO_MSI_X_TYPE`.
   - `vector_cnt` The number of requested vectors. For an MSI object, this must be equal to one. It must not exceed the value returned by `wsio_msi_capability`.
   - `hints` A placeholder for future hints. This must be zero.
   - `hndl` A pointer to an interrupt handle that is returned to the driver. The driver passes this handle into other WSIO MSI services.

3. Call `wsio_msi_get_cpus` to get a list of CPUs that are available for I/O interrupts. The driver can specify whether all CPUs must be returned, or limit the list to those local to the card, as described in the `isc` structure.

   **NOTE:** For MSI interrupts that are limited to a single CPU (as opposed to MSI-X), this function is useful only if the driver uses the `WSIO_MSI_NEXT_CPU` hint to round-robin CPU assignments for multiple MSI-capable cards.

4. Allocate an array of `wsio_msi_setinfo_t` structures that describe how each MSI/MSI-X vector is to be initialized. Each structure, defined in `<sio/wsio.h>`, has the following fields:

   ```c
typedef struct wsio_msi_setinfo {
   int     mask;
   int     (*isr)();
   void    *isr_arg;
   wsio_cpu_id_t cpu_id;
} wsio_msi_setinfo_t;
```

   Where:
   - `mask` Specifies the valid fields: `WSIO_MSI_SET_CPU` and `WSIO_MSI_SET_ISR`.
   - `isr` Specifies the driver ISR routine, typically `driver_isr`.
   - `isr_arg` Specifies the argument to be passed to the ISR.
   - `cpu_id` Specifies the CPU identifier returned from `wsio_msi_get_cpus`.

5. Call `wsio_msi_assign` to program the interrupt vectors. The `wsio_msi_assign` service has the following prototype:

   ```c
   wsio_ret_code_t wsio_msi_assign(
   ```
wsio_msi_hndl_t hndl,
wsio_msi_assign_type_t assign_type,
wsio_msi_set_info_t *vec_info,
int qualifier,
size_t num;
}

Where:

hndl             The driver’s interrupt handle.
assign_type      A set of flags indicating how to initialize the vectors. The
                 WSIO_MSI_ASSIGN_PER_VEC flag indicates that the array of
                 wsio_msi_set_info_t structures contains a separate entry for each
                 vector. The WSIO_MSI_ASSIGN_ALL flag indicates that all the specified
                 vectors are to be initialized from the same wsio_msi_set_info_t
                 structure.
vec_info          Pointer to the array of wsio_msi_set_info_t structures allocated in
                 the previous step.
qualifier         The interrupt vectors to be initialized.
num               The number of vectors to initialize, or the number of elements in the
                 vec_info array.

6. Call wsio_msi_enable to enable the MSI or MSI-X interrupts.

Additional WSIO MSI services enable the driver to disable interrupts, resize the interrupt object,
and query the currently defined interrupt object. For information on these functions and interfaces
to all WSIO MSI services, see the HP-UX 11i v3 Driver Development Reference.

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. The driver passes in the
WSIO_IRQ_LINE_AUTO flag to wsio_intr_set_irq_line, which informs the services that
they must determine the IRQ line value for the particular device.

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);

    /* Activate the interrupt. */
    status = wsio_intr_activate(isc, intr_obj);
    if(status != WSIO_OK)
        return(ERROR);

    else
return(wsio.ok)
}

The following is a sample ISR routine for the centif driver:

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;
        RealIntrHndlr(arg1, arg2);
        return 1; /* interrupt has been serviced */
    } else {
        return 0; /* interrupt not from my device */
    }
}

The following example uses MSI interrupts. The my_driver_init function checks the platform’s MSI capabilities, then calls wsio_msi_alloc to allocate an MSI handle with a single vector. It then sets up the ISR and its argument, and enables interrupts.

static int
my_driver_init (struct isc_table_type * isc)
{
    int vector_cnt;
    int cpu_cnt;
    wsio_cpu_id_t cpu_id;
    wsio_msi_hndl_t my_hndl;
    wsio_msc_setinfo_t *vector_args;

    /* Do any additional driver initialization.
    */
    /* Determine if the card can do MSI interrupts, and how many vectors
     * it supports.
     */
    if((ret = wsio_msi_capability(isc, WSIO_MSI_CAPABILITY, &vector_cnt)) != WSIO_OK)
        return(ret);
    /* Only 1 vector supported in the current implementation. */
    if (vector_cnt > 1)
        vector_cnt = 1;

    /* Get the next available CPU id. Since this is MSI, it only uses 1 CPU */
    cpu_cnt = 1;
    cpu_id = -1;
    if((ret = wsio_msi_get_cpus(isc, &cpu_id, &cpu_cnt, WSIO_MSI_NEXT_CPU)) != WSIO_OK)
        return(ret);

    /* Allocate an interrupt object */
    if((ret = wsio_msi_alloc(isc, WSIO_MSI_TYPE, vector_cnt, 0, &my_hndl) != WSIO_OK)
        return(ret);

    /* Allocate and initialize the vector */
    vector_args = kmalloc(sizeof(wsio_msi_setinfo_t) * vector_cnt, M_IHV, M_NOWAIT);
    bzero(vector_args, sizeof(wsio_msi_setinfo_t) * vector_cnt);
    vector_args[0].mask = WSIO_MSI_SET_CPU | WSIO_MSI_SET_ISR;
    vector_args[0].isr = my_isr;
    vector_args[0].isr_arg = isc;
    vector_args[0].cpu_id = cpu_id;

    /* Assign the vector to the hardware and enable interrupts */
    type = WSIO_MSI_ASSIGN_RANGE | WSIO_MSI_ASSIGN_PER_VEC;
    if((ret = wsio_msi_assign(my_hndl, type, vector_args, 0, vector_cnt)) == WSIO_OK)
        ret = wsio_msi_enable(my_hndl, WSIO_MSI_VECTOR_RANGE, 0, vector_cnt);
    kfree(vector_args);
    return(ret);
The following example uses MSI-X interrupts. The `msix_driver_init` function calls `wsio_msi_alloc` to allocate enough vectors for the maximum number of CPUs that the platform can support, but it will only program enough for the currently active ones. It then sets up the ISRs and their arguments. The maximum number of CPUs that a platform can support is assumed to be 128.

```c
#define MAX_CPU_CNT 128
static int
msix_driver_init (struct isc_table_type * isc)
{
    int vector_cnt;
    int cpu_cnt;
    wsio_cpu_id_t cpu_array[MAX_CPU_CNT];
    wsio_msi_hndl_t my_hndl;
    wsio_msc_setinfo_t *vector_args;
    /* Do any additional driver initialization */
    ...
    /* Find out which CPUs are available for I/O interrupts */
    cpu_cnt = MAX_CPU_CNT;
    if((ret = wsio_msi_get_cpus(isc, cpu_array, &cpu_cnt, WSIO_MSI_GET_ALL_CPUS)) != WSIO_OK)
        return(ret);
    /* Determine if the card can do MSI-X interrupts, and how many
     * vectors it supports. */
    if((ret = wsio_msi_capability(isc, WSIO_MSI_X_CAPABILITY, &vector_cnt)) != WSIO_OK)
        return(ret);
    /* Limit the number of vectors to the maximum number of CPUs
     * supported by the platform. */
    if (vector_cnt > MAX_CPU_CNT)
        vector_cnt = MAX_CPU_CNT;
    /* Allocate an interrupt object */
    if((ret = wsio_msi_alloc(isc, WSIO_MSI_X_TYPE, vector_cnt, 0, &my_hndl) != WSIO_OK)
        return(ret);
    /* Only initialize vectors for each CPU returned by
     * wsio_msi_get_cpus(). The remaining vectors are disabled until
     * more CPUs become available. */
    vector_args = kmalloc(sizeof(wsio_msi_setinfo_t) * cpu_cnt, M_IHV, M_NOWAIT);
    for (i = 0; i < cpu_cnt; i++)
    {
        vector_args[i].mask = WSIO_MSI_SET_CPU | WSIO_MSI_SET_ISR;
        vector_args[i].isr = msix_isr;
        vector_args[i].isr_arg = isc;
        vector_args[i].cpu_id = cpu_array[i];
    }
    /* Assign the vectors to the hardware and enable interrupts */
    type = WSIO_MSI_ASSIGN_RANGE | WSIO_MSI_ASSIGN_PER_VEC;
    if((ret = wsio_msi_assign(my_hndl, type, vector_args, 0, cpu_cnt)) == WSIO_OK)
        ret = wsio_msi_enable(my_hndl, WSIO_MSI_VECTOR_RANGE, 0, cpu_cnt);
    kfree(vector_args);
    return(ret);
}
```

### Writing a `driver_psize` Routine

The `driver_psize` function gets the swap partition size of a device. You must provide this routine. It can have any unique name. Pass the name to WSIO Services by specifying it in the `d_psize` field of the `drv_ops` structure. Replace `driver` with the driver's name.

The `driver_psize` function prototype is as follows:

```c
int  driver_psize (    dev_t  dev ) ;
```

Where:

- `dev` Contains encoded major and minor numbers.

The `driver_psize` WSIO function must return the size of the swap partition on a block swapping device. It is called by the kernel.
The \textit{driver\_psize} function returns the following values:

\begin{itemize}
  \item $>0$ Successful completion. The value is the swap partition size.
  \item $-1$ Error.
\end{itemize}

The following SCSI example assumes that \textit{driver\_psize} is never called when the device is closed; there is no need for an explicit open and close in the routine. The SCSI Services \textit{m\_scsi\_lun} function is used.

\begin{verbatim}
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);
}
\end{verbatim}

\section*{Interrupt-Driven Routines for Device Drivers}

This section discusses the routines used to manage device queues for device drivers that are interrupt-driven rather than context-driven.

If you write an interrupt-driven driver that also requires management of device queues, the driver must also handle problems related to the environment, for example, registers and variable states.

\section*{Management of Device Queues}

If more than one instance of the device driver can run simultaneously, provide the driver with a device queue. This prevents the driver from sending requests to the device faster than the device can complete them. A device queue enables 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 \textit{buf} data structure. Only the fields in the structure that are explicitly used in managing device queues are addressed.

\subsection*{buf Structure}

The \textit{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. All I/O requests end up as a \textit{buf} structure. The structure is defined in \texttt{<sys\_buf\_h>}. See \texttt{buf(9S)} in the HP-UX 11i v3 Driver Development Reference.

The following \textit{buf} field is important to device queue management:

\texttt{av\_forw} Points to the next \textit{buf} structure in the queue. Its value is \texttt{NULL} if the current \textit{buf} structure is last in the queue.

\section*{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 management, but a built-in parallel interface does not. Because all the devices on an interface card cannot perform 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 be only 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. Each interface has one queue.

Data Structures for Managing Interface Cards

Routines that manage interface cards use two data structures, buf and isc_table_type.

The isc_table_type Structure

The isc_table_type structure contains all information pertinent to the interface driver space, and is defined in <sys/io.h>. The structure is initialized by the WSIO before calling the driver attach routine. When claiming an interface card, the driver initializes some additional fields, such as the isc->gfsw_init field. See isc_table_type(9S) in the HP-UX 11i v3 Driver Development Reference.

Support for PCI OLAR

HP-UX supports PCI Online Addition and Replacement (OLAR) for those platforms that have hardware support. A driver instance can be suspended, the card replaced, and the driver instance resumed. Drivers that use 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 from the registered event mask for the affected card.

The type of events that a driver can register an interest in are as follows:

- 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 a WSIO_EVENT_SUSPEND event type, it suspends 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 a WSIO_EVENT_RESUME event type, it resumes the activities of the specified driver instance.

Registering a Driver Event Handler

A driver registers its event handler with the WSIO CDIO in its install routine by calling the wsio_install_drv_event_handler WSIO service. The calling interface for this service is as follows:

```c
int wsio_install_drv_event_handler__
    ((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. The second parameter is a pointer to the driver handler. The driver handler has the following calling interface:

```c
void (*drv_handler) (wsio_generic_event_t *);
```

It takes a single parameter, which is a pointer to a wsio_generic_event_t structure. This structure is defined in the header file wsio.h as follows:

```c
typedef struct wsio_generic_event {
    wsio_event_t event;
    wsio_event_id_t event_id;
    struct isc_table_type *isc;
    generic_complete_callback_t wsio_completion_cb;
    void *arg;
} wsio_generic_event_t;
```
Where:

- **event** specifies the type of event that the driver handler is being called for. For PCI OLAR the following types of `wsio_event_t` are defined:
  - `WSIO_EVENT_SUSPEND`
  - `WSIO_EVENT_RESUME`

- **event_id** Identifies an instance of an event. The driver passes it in the callback.

- **isc** Specifies the *isc* structure of the card.

- **wsio_completion_cb** Specifies the 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 using this callback mechanism.

The calling interface for the callback function is as follows:

```c
int (* wsio_completion_cb)( struct isc_table_type * isc,
                           wsio_event_id_t event_id,
                           void *status );
```

The first parameter is the *isc* of the driver instance, the second is the value of the `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 must set the status to `WSIO_OK`. Otherwise, it must set it to either `WSIO_UNSUPPORTED_EVENT` or `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 is an example of a driver install routine:
Example 4-6 Sample driver_install Routine

```c
#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,
}

void 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;
    }

    /***************************************************************************/
    * Link the driver's attach
    * routine into the global PCI attach list
    ***************************************************************************/
    my_drv_saved_attach = pci_attach;
    pci_attach = my_driver_attach;
    return;
}
```

Registering a Driver Event Mask

Drivers that register handlers must register an event 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 `wsio_reg_drv_capabilities_mask`. The calling interface for the service is as follows:

```c
int wsio_reg_drv_capability_mask __attribute__((__weak__))
    ((struct isc_table_type * isc,
      wsio_event_mask_t mask));
```

The first parameter is the driver's `isc` structure. The second parameter is the mask that is formed by ORing one or more `wsio_event_t` values. For PCI OLAR, drivers must pass in a mask as follows:

```c
(WSIO_EVENT_SUSPEND | WSIO_EVENT_RESUME)
```

Setting Event Timeout Values

The WSIO CDIO sets a timeout when it calls the driver handler. A driver suspend operation must always succeed. If the handler does not reply using the callback function before the timeout expires, the WSIO assumes an error has occurred, and the interface node is kept in TIMEDOUT state. Drivers can change the value of the timeout by calling the `wsio_set_parm` WSIO service in their attach routine. They can also query the default value for the timeout by calling the `wsio_get_parm` WSIO service. 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 as follows:

```c
int wsio_get_parm({
    struct isc_table_type *I_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
);

Where:

I_isc        A pointer to an isc structure of the driver instance.
I_parm       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.
I_value      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.

For more information, see wsio_get_parm(9F) and wsio_set_parm(9F) in the HP-UX 11i v3 Driver Development Reference.

The following example shows a driver attach routine setting the capabilities mask and timeout values:

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 and instance processing. */
        
        /**********************************************************************************
         * Claim the card and register an event's capabilities.
         **********************************************************************************/
        isc_claim(isc, &my_drv_info);

        /**********************************************************************************
         * The return value must 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 is neither, an unknown condition has occurred. Exit.
         **********************************************************************************/
        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. */
            
            /**********************************************************************************
             * Modify the timeout values for
             **********************************************************************************/
            
            isc_unclaim(isc, &my_drv_info);
        } else {
            /* */
        }
    } else {
        /* */
    }
}
Sample 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 following example are left out for simplicity. A driver returns 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 logs messages to dmesg buffer.

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

/* Following is a sample of a global data structure to
 * temporarily save the WSIO's isc, hint, event_id, and
 * callback functions. The algorithm for a suspend event is
 * as follows: 1) The event handler is called. 2) The
 * handler stores the isc, hint, even_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 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 schedules
 * 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 must return a
     * define, such as WSIO_OK, WSIO_UNSUPPORTED_EVENT, or
     * WSIO_ERROR. It can be typecasted as (void *) to avoid
```
my_suspend_cp->complete_cb(isc, my_suspend_cp->event_id,
   (void *)WSIO_OK)
}

/* The following event handler first stores 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() to run. The function
 * returns right away. A driver can decide on a timeout value
 * depending on whether it wants to accommodate any in-progress
 * activities. A driver can design
 * the best approach to fit its requirement. The event ID is
 * a tag that WSIO generates to prepare for future use
 * (match up request with a reply); driver must return it in
 * the completion callback. */

void my_handler(wsio_generic_event * handler_arg) {
    switch (handler_arg->event) {
    case WSIO_EVENT_SUSPEND:
        /* Store 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 Writing a DLKM Driver

This chapter addresses the following topics:

- DLKM Concepts
- DLKM Module Metadata
- DLKM Data Structures
- Dynamic Loading and Unloading Function
- DLKM-Specific to Different Module Types
- Auto-Load Stubs
- Examples of Load and Unload Functions

Dynamically Loadable Kernel Modules

There are two basic ways to use a kernel module. Traditionally, kernel modules have been statically bound into the main kernel executable, `/stand/vmunix`. This method is simple, and still supported, but it has some significant disadvantages. It requires you to rebuild the kernel executable and reboot the system to add, remove, or patch a kernel module. Similarly, it requires you to rebuild and reboot after each change to the kernel module.

Kernel modules can also be developed to support dynamic loading. A **Dynamically Loadable Kernel Module** (DLKM) can be loaded into the running kernel without rebuilding the kernel or rebooting the system. Most dynamically loadable modules are also unloadable, so they can be removed from the running kernel without needing a rebuild or reboot. When you use kernel modules that support dynamic loading, you can reconfigure your kernel quickly and without costly downtime. Module developers can test changes to dynamic modules without having to wait for systems to reboot.

Dynamically loadable modules require a few extra data structures and functions that are not needed for static modules, so there is a slight development cost associated with making a DLKM. This cost is usually overshadowed by the productivity benefit of dynamic loading during development.

There is a slight performance penalty when a module is dynamically loaded. It is not usually noticeable, but can be an issue in some performance-critical environments. For this reason, you can dynamically load and unload modules when determining the desired system configuration, and then build a static kernel containing those modules for production use.

**TIP:** HP recommends that all modules be designed to support static binding, dynamic loading, and dynamic unloading.

Dynamic Loading Concepts

This section addresses some basic concepts that you must understand to write dynamically loadable modules.

Module Load Times

Using the `loadtimes` directive in the module metadata file, you can specify which load times the module supports and which load time is preferred. For example, device drivers typically use `loadtimes driver_install run`. The driver prefers to be loaded during the kernel boot process at the time drivers are being initialized, but they can also be loaded after boot is complete. Possible load times and module metadata information are listed in Chapter 6 (page 157).

When a module must be loaded during boot (the module is in the `loaded` state), it is loaded at one of the load times (dispatch points) specified in its metadata. Usually this is the first load time.
listed in the metadata. However, one of the other load times can be used if necessary to satisfy
dependencies between modules.

All dynamically loadable modules must support the run load time. A module that does not
support the run load time cannot be loaded except during boot. There is little purpose in having
it be dynamically loadable at all.

**IMPORTANT:** PA-RISC systems do not support loading during boot. Specification of load times
other than run are ignored. The run load time is required on PA-RISC systems.

### Dependencies Between Modules

The dependency directive in the module metadata file enables any module to state a dependency
on any other module or on interfaces exported by other modules. Dependencies interact with
dynamic loading as follows:

- A module in static state cannot be dependent on a module in loaded or auto state. The
  kernel configuration tools resolve the dependency by putting both modules in static state,
  if they both support it. If not, the configuration is rejected.
- A module in loaded or auto state can be dependent on a module in static state.
- A module in loaded or auto state can be dependent on another module in loaded or
  auto state only if the depending module loads at the same load time or later. The kernel
  configuration tools resolve the dependency by loading both modules at the earlier of the
  two modules’ preferred load times, if they both support loading at that load time. If not, the
  configuration is rejected.
- When a module is loaded, any modules on which it depends are also loaded (if they are not
  already loaded).
- If a module is automatically loaded to resolve dependencies, the system automatically
  attempts to unload it when the last of its dependent modules is unloaded.

### Module Load Sequence

When the kernel receives a request to load a module, it performs the following steps:

1. The module code is loaded into kernel memory. (Possible failures include file not found and
   insufficient memory.)
2. All the modules on which the target module depends are loaded, if they are not already
   loaded or statically bound into the kernel.
3. All the externalsymbol references of the target module are resolved. (Possible failures include
   reference to unknown symbol and duplicate definition of symbol.)
4. The system calls any of the module initialization functions registered for dispatch points
   that have already passed. For more information, see Chapter 6 (page 157).
5. The system calls the module load function, if any. See “Dynamic Loading Functions”
   (page 140). (Possible failures include load function returns nonzero value.)

A failure anywhere in this process causes the whole process to abort and reverse itself. For
information about a load failure, see the /var/adm/syslog/syslog.log file.

### Synchronization with Other Operations

Dynamic loading and unloading of drivers (wsio_class, wsio_intfc, and streams_drv
module types) are automatically synchronized with the following operations:

- Interrupt Migration
- ioscan
- rmsf

138 Writing a DLKM Driver
Only one such operation is allowed at a time. For example, while loading a driver module, all other operations (for example, `ioscan`) are blocked. Blocked operations resume after the load operation finishes.

**Automatic Module Loads**

The system administrator can select either of two different module states that result in dynamic loading, `loaded` or `auto`. For more information, see Chapter 6 (page 157). The difference between the two is when the dynamic loading occurs. When the administrator puts a module in the `loaded` state, the module loads immediately, and loads during each successive boot of the system.

When the administrator puts a module in the `auto` state, the module does not load immediately. Instead, the system waits until something attempts to use the services supplied by the module; it then loads. For example, if the module is a driver, the system loads the module when some process attempts to open the associated device special file. After each successive boot, the system waits until the module is needed before loading it.

A module that supports the `loaded` state does not need to support the `auto` state. HP recommends that it does. A module cannot support the `auto` state unless it also supports the `loaded` state.

Most module types support autoloading for the primary entry point to modules of that type. For example, `wsio_class` and `wsio_intfc` modules have built-in support for autoloading when a process opens the corresponding device special file (for the `open` call to the class driver). The `streams_drv` and `streams_mod` modules can also be autoloaded on first open. The only thing you must do to enable autoloading is to list the `auto` state on the `states` line in the module's metadata.

If you want to support autoloading on calls to other functions within the module, you must provide autoload stubs for those functions. You must also provide autoload stubs for miscellaneous modules because none are provided. For instructions on providing autoload stubs, see “Autoload Stubs” (page 149).

**Module Metadata for DLKMs**

The details of the module metadata files are described in Chapter 6 (page 157). When converting a module to support dynamic loading, address these parts of the metadata:

- **Supported States (the `states` directive)**
  Adds `loaded` and perhaps `auto` to the list, putting the preferred state at the top of the list.

- **Supported Load Times (the `loadtimes` directive)**
  Adds this directive, listing all supported load times for the module and listing the preferred load time first.

- **Unloadable Flag (the `unloadable` directive)**
  Adds this directive if the module supports dynamic unloading.

- **Driver Details (the `driver{}` block and its contents)**
  Drivers that support dynamic loading must have a `driver{}` block containing at least a `type` directive; the `class`, `flags`, and `struct drv_info` directives are recommended.

**Dynamic Loading Data Structures**

To support dynamic loading, a module must have a set of dynamic loading data structures. The dynamic loading data structures are defined in `<sys/moddefs.h>`. The primary data structure is `struct modwrapper`:

```c
struct modwrapper {
    int     mw_rev;
    int     (*mw_load)(void *);
    int     (*mw_unload)(void *);
};
```
Each dynamically loadable module must define a structure of this type. The structure must be named `modulename_wrapper`, and must be externally visible. The structure fields are as follows:

- **mw_rev**: Indicates the version of `moddefs.h` with which the module is compiled. Always set this field to `MODREV`.
- **mw_load**: Pointer to the module load function.
- **mw_unload**: Pointer to the module unload function.
- **mw_halt**: Reserved. Set to `NULL`.
- **mw_conf_data**: Pointer to the module configuration data. Always set this field to `modulename_conf_data`. The data structure is automatically generated as a part of the module metadata. However, the source file that defines the `modwrapper` structure needs an `extern`. For example:
  ```c
  extern struct mod_conf_data modulename_conf_data
  ```
- **mw_modlink**: Pointer to an array of `struct modlink`.

The array of `struct modlink` gives information about the operations supported by the module. Its exact contents are dependent on the type of the module, and are described in “Module Type-Specific DLKM Requirements” (page 141). The structure fields are as follows:

```c
struct modlink {
    struct  mod_operations  *ml_ops;
    void    *ml_type_data;
};
```

Where:

- **ml_ops**: A pointer to the module operations switch table for the module type. The core kernel provides these switch tables. This field must point to the correct table for the correct module type. See “Module Type-Specific DLKM Requirements” (page 141).
- **ml_type_data**: A pointer to module type-specific data.

The array of `struct modlink` must be terminated by a `[NULL, NULL]` entry.

The following example shows a set of data structures for a device driver:

```c
extern struct mod_conf_data mydriver_conf_data;
struct modlink mydriver_modlink[] = {
    { &gio_mod_ops, &mydriver_drv_link }, /* see driver section */
    { NULL,         NULL               }
};
struct modwrapper mydriver_wrapper = {
    MODREV,
    mydriver_load,
    mydriver_unload,
    (void (*)(void))NULL,
    &mydriver_conf_data,
    mydriver_modlink
};
```

### Dynamic Loading Functions

Dynamically loadable modules must have a load function and an unload function. Pointers to these functions are specified in the module’s `modwrapper` structure; see “Dynamic Loading Data Structures” (page 139). If a module does not have a load or unload function, the corresponding function pointers in the `modwrapper` structure must be set to `NULL`. If the load function pointer is set to `NULL`, the module loads, but does not initialize. If the unload function pointer is set to
NULL, the module unloads without doing any cleanup, which may have undesirable consequences. For details on the module load and unload sequence, showing when these functions are called, see the module load function for drivers in “Module Type-Specific DLKM Requirements” (page 141).

The module load function is called during the process of dynamically loading the module into the kernel. The parameters to the function vary depending on the module type. See “Module Type-Specific DLKM Requirements” (page 141). The load function must prepare the module for use. If it is successful, it must return zero. Any other return causes the module load to be aborted. Nonzero return values must be one of the error constants defined in `<sys/errno.h>`. If a module does not have a load function (that is, the load function pointer in its `modwrapper` structure is NULL), the module load proceeds without any initialization.

The module unload function is called during the process of dynamically unloading the module from the kernel. The parameters to the function vary depending on the module type. See “Module Type-Specific DLKM Requirements” (page 141). The unload function must prepare the module for unload. If it is successful, it must return zero. Any other return causes the module unload to be aborted. Nonzero return values must be one of the error constants defined in `<sys/errno.h>`. If a module does not have an unload function (that is, the unload function pointer in its `modwrapper` structure is NULL), the module unload proceeds without any cleanup.

If a module supports unloading, it must specify the `unloadable` directive in its module metadata. Otherwise, it cannot unload.

TIP: HP recommends that modules support unloading whenever possible; much of the benefit of dynamic loading capabilities is lost without unload capability.

Module Type-Specific DLKM Requirements

This section addresses DLKM requirements for the following module types:

- “WSIO Class and Interface Drivers”
- “STREAMS Drivers” (page 145)
- “Native DLPI Drivers” (page 147)
- “STREAMS Modules” (page 147)
- “Miscellaneous Modules” (page 148)

WSIO Class and Interface Drivers

WSIO drivers control the hardware attached to the system. Class drivers are the upper layer. They have device special files in `/dev` and are used by applications. Interface drivers are the bottom layer. The two types of drivers cooperate, and in some cases are combined into monolithic drivers. See Chapter 4 (page 77) for more information on class and interface drivers.

Dynamic Loading Data Structures

WSIO driver modules must have the dynamic loading data structures defined in “Dynamic Loading Data Structures” (page 139). For WSIO driver modules (both class and interface), the array of `struct modlink` must look like the following:

```c
struct modlink mydriver_modlink[] = {
    { &gio_mod_ops, &mydriver_drv_link },
    { NULL, NULL               }
};
```

`mydriver_drv_link` is a `struct mod_type_data`:

```c
struct mod_type_data {
    char *mtd_info;
    void *mtd_pdata;
};
```
Where:

- **mtd_info**: A string describing the driver. Typically, this is the same string as the `desc` field in the module metadata.

- **mtd_pdata**: This field must be set to NULL for WSIO driver modules.

The following is a complete example of the dynamic loading data structures for a WSIO driver:

```c
extern struct mod_conf_data mydriver_conf_data;
struct mod_type_data mydriver_drv_link = {
   "My Sample Driver",
   NULL
};
struct modlink mydriver_modlink[] = {
   { &gio_mod_ops, &mydriver_drv_link },
   { NULL, NULL               }
};
struct modwrapper mydriver_wrapper = {
   MODREV,
   mydriver_load,
   mydriver_unload,
   (void (*)(()))NULL,
   &mydriver_conf_data,
   mydriver_modlink
};
```

### Module Preparation Scripts for Drivers

Dynamically loadable drivers can provide a module preparation script that runs before and after loading and unloading the driver. For more information about module preparation script, see “Step 8: Creating Module Preparation Scripts” (page 165). This section describes the actions that a driver module might take in its module preparation script. See “Examples” (page 154) for examples of driver module preparation scripts.

#### Preload Operation

The `modprep` script is called with an operation of `preload` when a driver is about to be loaded in run phase. This script is not run in any other phase.

Currently, no preload functionality is foreseen for drivers. However, the facility is available for any driver-specific needs that arise.

#### Postload Operation

The `modprep` script is called with an operation of `postload` after an attempt to load a driver during boot, runtime, or after the driver is statically bound into the kernel.

If the load attempt was successful (indicated by the status argument passed into the script), the script can do the following:

- Run `ioscan` to let the newly loaded driver claim its hardware and start a discovery of the attached devices.
- Create device special files for the driver.
- Restore the driver-specific configuration from persistent storage if any.

If the load attempt fails, the script actions depend on the error status passed in. It can undo any steps taken by the preload operation.

#### Preunload Operation (No PCI Online Deletion Support)

The `modprep` script is called with an operation of `preunload` when a driver is about to be unloaded in the run phase. This script is not run in any other phase.

The script must perform a **Critical Resource Analysis (CRA)**, analyzing the impact of unloading the driver to the subsystems and applications. The script fails if it determines that the driver...
cannot be safely unloaded. It must also provide clear information on why the unload fails so that a system administrator can perform necessary actions to make sure unload will succeed when it is rerun.

The script does the following:

- Checks the list of currently active applications that have opened the driver module. For example, it can run `fuser` on the driver device special files to display the process IDs of active applications on that driver.
- Parses the output of `ioscan` to see whether the driver claimed any device or card.

The script returns an error code of 1, if the check shows that the driver cannot be unloaded successfully. The driver unload does not proceed any further.

### Postunload Operation (No PCI Online Deletion Support)

The `modprep` script is called with the `postunload` operation after an attempt to unload a driver in the run phase. This script is not run in any other phase. If the unload attempt succeeded (indicated by the status argument passed into the script), the script does the following:

- Removes the driver device special file.
- Saves the driver-specific configuration if any so that future loads could restore it.

If the unload attempt fails, the script actions depend on the error status passed in. It can undo any steps taken by the previous `preunload` script.

### Driver Initialization and Dynamic Loading Function

This section assumes that the driver initialization code for static binding is already written, as described in “Writing a driver_dev_init Routine” (page 92) and “Writing a driver_if_init Routine” (page 92). This section addresses only those changes needed to add support for dynamic loading.

The driver install function, traditionally named `modulename_install`, is called only when the module is statically bound. When the module is dynamically loaded, similar (but not identical) operations are performed by the module's load function, and the module load fails if both the driver's load and install functions are called. For that reason, add the `static` keyword to the end of the `initfunc driver_install` directive. The module metadata must be marked so that the function is called only when the module is in the `static` state:

```plaintext
initfunc driver_install modulename_install static
```

To support dynamic loading of the driver module, it must have a load function. See “Dynamic Loading Functions” (page 140). The load function receives one parameter, which is a pointer to a `struct drv_info` for the module:

```plaintext
int modulename_load(const struct drv_info *);
```

The load function must do the following:

1. Ensure that the driver can be properly loaded. If not, it must return a nonzero value chosen from among the error codes in `<sys/errno.h>`. This aborts the module load.
2. Call `wsio_install_driver`, passing it a pointer to the driver's `wsio_drv_info` structure. The `drv_info` field of this structure must be set to the pointer passed into the load function.
3. Perform any actions that the driver's install function handles when the module is statically linked. However, it must not modify or use the `dev_init` chain.
4. Add its attach function, if any, using the `mod_wsio_attach_list_add` function.
5. Register the driver' probe function, if any, using `wsio_register_dev_probe` or `wsio_register_probe_func`. Then, activate it by calling `wsio_activate_probe`.
6. Return zero if all the previous steps are successful.

Driver modules must also support dynamic unloading. To do this, modules must supply an unload function.
Dynamic Unload Function (PCI Online Deletion Support)

The unload function receives one parameter, which is a pointer to the `struct drv_info` for the module:

```
int modulename_unload(const struct drv_info *);
```

The unload function must do the following:

1. Free all allocated memory and other resources such as spinlocks.
2. Perform any other necessary driver-specific cleanup operations.
3. Unregister its probe function, if any, using `wsio_unregister_dev_probe` or `wsio_unregister_probe`.
4. Remove its attach function, if any, using the `mod_wsio_attach_list_remove` function.
5. Unregister any tunable handlers that the module registered. See Chapter 8 (page 183).
6. Call `wsio_uninstall_driver`, passing it a pointer to the driver's `wsio_drv_info` structure. The `drv_info` field of this structure must be set to the pointer passed into the unload function.
7. Return zero if all actions succeed to enable the unload to proceed.

**NOTE:** This list is not exhaustive; driver stack specific unregistration and other uncommonly used driver resources are not listed here. Dynamically unloadable modules must release/free/unregister resources in the reverse order of acquire/allocate/register.

Sample WSIO interface driver load and unload functions are shown in “Examples” (page 154).

Dynamic Unload Function (No PCI Online Deletion Support)

Driver modules must also support dynamic unloading with an unload function. The unload function receives one parameter, which is a pointer to the `struct drv_info` for the module:

```
int modulename_unload(const struct drv_info *);
```

The unload function must do the following:

1. Check that the module can be safely unloaded, and return a nonzero value if it cannot. An unload of a class driver is never attempted if any processes have the driver open, and opens are blocked while the unload is in progress. The unload function does not need to check for these cases. If the module cannot be safely unloaded, the unload function must return a nonzero value chosen from among the error codes in `<sys/errno.h>`. This aborts the module unload.
2. Untimeout any outstanding timeouts.
3. Quiesce the hardware it controls.
4. Unregister the driver ISR. WSIO handles the synchronization between unregistration of the driver ISR and completion of any outstanding interrupts. The `wsio_intr_deactivate` function does not return while the associated driver ISR is running.
5. Free all DMA resources used by the driver.
6. Free all allocated memory and other resources such as spinlocks.
7. Perform any other necessary driver-specific cleanup operations.
8. Unregister its probe function, if any, using `wsio_unregister_dev_probe` or `wsio_unregister_probe`.
9. Remove its attach function, if any, using the `mod_wsio_attach_list_remove` function.
10. Unregister any tunable handlers that the module registered. See Chapter 8 (page 183).
11. Call `wsio_uninstall_driver`, passing it a pointer to the driver's `wsio_drv_info` structure. The `drv_info` field of this structure must be set to the pointer passed into the unload function.
12. Return zero if all tasks succeed to enable the unload to proceed.
NOTE: This list is not exhaustive; driver stack specific unregistration and other uncommonly used driver resources are not listed here. Dynamaically unloadable modules must release/free/unregister resources in the reverse order of acquire/allocate/register.

Sample WSIO interface driver load and unload functions are shown in “Examples” (page 154).

STREAMS Drivers

STREAMS drivers are drivers for the de facto standard STREAMS protocol. For detailed information about STREAMS, see the STREAMS/UX for the HP 9000 Reference Manual.

IMPORTANT: Disregard the instructions in the STREAMS manual about how to build STREAMS drivers and modules. Instead, follow the instructions in this document.

Dynamic Loading Data Structures

STREAMS driver modules must have the dynamic loading data structures defined in “Dynamic Loading Data Structures” (page 141). For STREAMS driver modules, the array of struct modlink is as follows:

```c
struct modlink mydriver_modlink[] = {
    { &str_drv_ops, &mydriver_drv_link },
    { NULL, NULL }
};
```

`mydriver_drv_link` is a struct mod_type_data:

```c
struct mod_type_data {
    char *mtd_info;
    void *mtd_pdata;
};
```

Where:

- `mtd_info` A string describing the driver. Typically, this is the same string as the `desc` field in the module metadata.

- `mtd_pdata` Points to the `streams_info_t` structure for the driver.

The following is a complete example of the dynamic loading data structures for a STREAMS driver:

```c
extern struct mod_conf_data mydriver_conf_data;
extern streams_info_t mydriver_str_info;
struct mod_type_data mydriver_drv_link = { 
    "My Sample Driver", 
    &mydriver_str_info
};
struct modlink mydriver_modlink[] = {
    { &str_drv_ops, &mydriver_drv_link },
    { NULL, NULL }
};
struct modwrapper mydriver_wrapper = {
    MODREV,
    mydriver_load,
    mydriver_unload,
    (void (*)(()))NULL,
    &mydriver_conf_data,
    mydriver_modlink
};
```
Driver Initialization

This section assumes that the driver initialization code for static binding is already written, as described in the STREAMS/UX for HP 9000 Reference Manual. This section addresses only those changes needed to add support for dynamic loading.

The driver install function, traditionally named `modulename_install`, can be called only when the module is statically bound. When the module is dynamically loaded, similar (but not identical) operations are performed by the module's load function, and the module load fails if both the driver's load and install functions are called. For that reason, add the `static` keyword to the end of the `initfunc driver_install` directive in the module metadata so that the function is called only when the module is in the static state:

```
initfunc driver_install modulename_install static
```

To support dynamic loading of the STREAMS driver, it must have a load function. A pointer to the load function is placed in the module's `modwrapper` structure. The load function receives one parameter, which is a pointer to a `struct drv_info` for the module:

```
int modulename_load(const struct drv_info *);
```

The load function must do the following:

1. Call `install_driver`, passing it the `struct drv_info` pointer that was given to the load function.
2. Call `str_install`, passing it a pointer to the driver's `streams_info_t` structure. However, before doing so it must update the `inst_major` field of that structure to match the `c_major` field of the `struct drv_info` that was given to the load function.
3. Return zero if all the previous steps succeed.
4. Return a nonzero value chosen from the error codes in `<sys/errno.h>`, if anything fails or if it is unsafe to load the driver. This aborts the driver load.

Driver modules must also support dynamic unloading. To support this, the driver must supply an unload function. The unload function receives one parameter, which is a pointer to the `struct drv_info` for the module:

```
int modulename_unload(const struct drv_info *);
```

The unload function must do the following:

1. Check that the module can be safely unloaded, and return a nonzero value if it cannot.
   An unload of a STREAMS driver is never attempted if any processes have the driver open. The unload function does not need to check for this. If the module cannot be safely unloaded, the unload function must return a nonzero value chosen from among the error codes in `<sys/errno.h>`. This aborts the module unload.
2. Untimeout any outstanding timeouts.
3. Free all allocated memory and other resources such as spinlocks.
4. Unregister any tunable handlers that the module registered. See Chapter 8 (page 183).
5. Call `str_uninstall`, passing it a pointer to the driver's `streams_info_t` structure.
6. Return zero if all tasks succeed to enable the unload to proceed.

**NOTE:** The list is not exhaustive. Dynamically unloadable modules must release/free/unregister resources in the reverse order of acquire/allocate/register.

An example of STREAMS driver load and unload functions are shown in “STREAMS Driver Load and Unload Functions” (page 156).
Native DLPI Drivers

Native Data Link Provider Interface (DLPI) drivers are hybrids of WSIO and STREAMS drivers. Their module metadata identifies them as type streams_drv. They use the dynamic loading data structures described for STREAMS drivers.

The load and unload functions for native DLPI drivers perform both the tasks described for WSIO drivers and for STREAMS drivers.

For more information, see Chapter 12 (page 293).

STREAMS Modules

STREAMS modules are modules that can be inserted in an I/O stream in the de facto standard STREAMS protocol. For more information about STREAMS, see the STREAMS/UX for the HP 9000 Reference Manual.

**NOTE:** Disregard the instructions in the STREAMS manual about how to build STREAMS modules and drivers. Instead, use the instructions in this document.

Dynamic Loading Data Structures

STREAMS modules must have the dynamic loading data structures defined in “Dynamic Loading Data Structures” (page 139). For STREAMS modules, the array of struct modlink is as follows:

```c
struct modlink mymodule_modlink[] = {
    { &str_mod_ops, &mymodule_mod_link },
    { NULL, NULL }
};
```

mymodule_mod_link is a struct mod_type_data:

```c
struct mod_type_data {
    char *mtd_info;
    void *mtd_pdata;
};
```

Where:

- **mtd_info** A string describing the module. Typically this is the same string as the desc field in the module metadata.
- **mtd_pdata** Points to the streams_info_t structure for the module.

The following is a complete example of the dynamic loading data structures for a STREAMS module:

```c
extern struct mod_conf_data mymodule_conf_data;
extern streams_info_t mymodule_str_info;
struct mod_type_data mymodule_mod_link = {
    "My Sample Module",
    &mymodule_str_info
};
struct modlink mymodule_modlink[] = {
    { &str_mod_ops, &mymodule_mod_link },
    { NULL, NULL }
};
struct modwrapper mymodule_wrapper = {
    MODREV,
    mymodule_load,
    mymodule_unload,
    (void (*)())NULL,
    &mymodule_conf_data,
    mymodule_modlink
};
```
Module Initialization

This section assumes that the module initialization code for static binding is already written, as described in the STREAMS/UX for HP 9000 Reference Manual. This section addresses only those changes needed to add support for dynamic loading.

STREAMS modules have the same initialization code regardless of whether they are statically linked or dynamically loaded. However, that code is invoked in different ways in the two cases. For static linking, the initialization code is in an install function, traditionally called modulename_install, invoked because of an initfunc directive in the module's metadata:

\[
\text{initfunc driver_install modulename_install}
\]

To support dynamic loading, make the following changes:

1. Add the static keyword to the end of the initfunc directive.
2. Put a pointer to the modulename_install function in the mw_load field of the module's modwrapper structure, so that it also acts as the module's load function. For more information on module load functions, see “Dynamic Loading Functions” (page 140).
3. Ensure that the function returns zero for success, and a nonzero error code from <sys/errno.h> if the module load is rejected.

STREAMS modules must also support dynamic unloading. To support this, the module must supply an unload function. The unload function must do the following:

1. Check that the module can be safely unloaded, and return a nonzero value if it cannot.
   
   An unload of a STREAMS module is never attempted if it is pushed onto any streams. The unload function does not need to check for this. If the module cannot be safely unloaded, the unload function must return a nonzero value from the error codes in <sys/errno.h>. This aborts the module unload.

2. Call str_uninstall, passing it a pointer to the module's streams_info_t structure.
3. Unregister any tunable handlers that the module registered. See Chapter 8 (page 183).
4. Return zero if all tasks succeed to enable the unload to proceed.

Miscellaneous Modules

Miscellaneous modules can contain any type of kernel code. They can be used in circumstances where the other module types are not appropriate.

Dynamic Loading Data Structures

Miscellaneous modules must have the dynamic loading data structures defined in “Dynamic Loading Data Structures” (page 139). For miscellaneous modules, the array of struct modlink is as follows:

\[
\text{struct modlink mymodule_modlink[]} = \\
\{ \\
\{ &mod_misc_ops, &mymodule_mod_link \\
\{ NULL, NULL \}
\}
\]

mymodule_mod_link is a struct mod_type_data:

\[
\text{struct mod_type_data} \\
\{ \\
\text{char *mtd_info;} \\
\text{void *mtd_pdata; }
\}
\]

Where:

\[\text{mtd_info}\]

A string describing the module. Typically this is the same string as the desc field in the module metadata.

\[\text{mtd_pdata}\]

Set this field to NULL for miscellaneous modules.
Following is a complete example of the dynamic loading data structures for a miscellaneous module:

```c
extern struct mod_conf_data mymodule_conf_data;
struct mod_type_data mymodule_mod_link = {
    "My Sample Module",
    NULL
};
struct modlink mymodule_modlink[] = {
    { &mod_misc_ops, &mymodule_mod_link },
    { NULL, NULL }
};
struct modwrapper mymodule_wrapper = {
    MODREV,
    mymodule_load,
    mymodule_unload,
    (void (*)(void))NULL,
    &mymodule_conf_data,
    mymodule_modlink
};
```

**Module Initialization**

A miscellaneous module can have a load function. See “Dynamic Loading Functions” (page 140). If successful, the load function must return zero to enable the load to complete. If unsuccessful, the load function must return a nonzero value from the error codes in `<sys/errno.h>`. This aborts the module load.

Miscellaneous modules must also support dynamic unloading. The module can supply an unload function. The unload function must do the following:

1. Check that the module can be safely unloaded, and return a nonzero value if it cannot.
   
   The system does not perform any safety checks before unloading a miscellaneous module; the unload function must do all necessary checks. If the module cannot be safely unloaded, the unload function must return a nonzero value from the error codes in `<sys/errno.h>`. This aborts the module unload.

2. Perform any necessary cleanup and termination of the module services.
3. Unregister any tunable handlers that the module registered. See Chapter 8 (page 183).
4. Return zero if all tasks succeed to enable the unload to proceed.

**Autoload Stubs**

Autoload stubs enable a kernel module to be loaded on first use. Consider the case of a dynamically loadable device driver. The core kernel maintains a switch table associating device special files to device drivers through their major numbers. When a process opens the device special file, the kernel calls the device driver’s `open` function through the function pointer in the switch table.

If the device driver module has not yet been loaded, the `open` function pointer in the switch table points to an autoload stub. This small function loads the module, then replaces itself with the driver’s real open function.

Autoload stubs are supplied automatically for the `open` functions for `wsio_class`, `wsio_intfc`, `streams_drv`, and `streams_mod` modules types. No autoload stubs are automatically supplied for `misc` modules.

If a module needs autoload stubs beyond those that are automatically supplied, you must create and provide them. Autoload stubs for a module are supplied as a separate module.

Create a stub module in the same way you create other modules. See the following sections for details.
Module Metadata for Stub Modules

Write the module metadata file for a stub module like any other module metadata file. See Chapter 6 (page 157). The following are specifics for the various metadata fields:

Module Name
The same as the base module name with a _stub suffix.

Module Version
The stub module version is independent of the base module version. Typically the base module's version changes more often.

Module Type
The stub module type must be misc.

Description
A typical description is "Autoload Stubs for the Module".

Supported States
Stub modules must support the static and loaded states. They cannot support the auto state.

Supported Load Times
Stub modules must support the same load times as the base module. They can support additional load times if desired.

Unloadable Flag
Stub modules must be marked unloadable.

Initialization Functions
Stub modules must have an initialization function declared as follows:

```c
initfunc first_loadtime modulename_load static
```

Where first_loadtime is the earliest load time supported by the stub module. This calls the stub module's load function even if the stub module is statically bound.

Exports
If the module exports any interfaces, these must be listed in the metadata for the stub module, not in the metadata for the base module.

The stub module must list all exported symbols and modulename_modinfo in the exports only block.

Dependencies, Tunables, Driver Details
Stub modules do not need these.

Following is an example of a module metadata file for a stub module:

```c
module sample_stub {
    desc       "Auto-load Stubs for sample Module"
    type       misc
    version    1.0.0
    states     loaded static
    loadtimes  driver_install run
    unloadable
    initfunc   driver_install sample_stub_load static
    exports only {
        symbol sample_modinfo
        ...
    }
}
```

Module Metadata for Base Modules

If a module xyz has autoload stubs in a stub module xyz_stub, the metadata for module xyz must contain a dependency on xyz_stub:

dependency module xyz_stub
Also, the `states` line in the metadata for module `xyz` must list the auto state as a supported state.

If the module exports any interfaces, list these in the metadata for `xyz_stub`, not in the metadata for `xyz`.

**Module Metadata for Client Modules**

Modules that need the services of `xyz` must declare their dependency on `xyz_stub`, not on `xyz`, as follows:

```plaintext
dependency module xyz_stub
```

**Code for Stub Modules**

Aside from the module metadata, stub modules have a single source file, usually called `modulename.mc` (for example, `sample_stub.mc`). This source file has a special format, consisting of a set of macro invocations. Specifically, it invokes the `MODULE` macro, then an entry for each function that needs an autoload stub, then an invocation of the `END_MODULE` macro. The entries for each function are invocations of the `STUB`, `WSTUB`, or `USTUB` macros.

When writing a stub module, you must choose whether to support unload of the base module. It is more flexible and convenient for system administrators and you if the base module is unloadable. However, a base module that has autoload stubs pays a small performance penalty, at each call into the base module from outside it, to support unloading.

If the base module supports unloading, all the autoload stubs in the stub source file must be declared using the `USTUB` macro. The `STUB` and `WSTUB` macros are not permitted in stubs for unloadable modules.

If the base module does not support unloading, you must choose whether to use the `STUB` or `WSTUB` macro for each stub. (The `USTUB` macro is not permitted if the base module does not support unload.) The difference between `STUB` and `WSTUB` is what happens if a function in the base module is called and the base module is not already loaded. If the stub is declared with `STUB`, the module is loaded. If the stub is declared with `WSTUB`, an error is returned to the caller of the function.

For example, a module exports four functions: `open`, `close`, `read`, and `write`. If the module is unloadable, the stubs for all four functions must be declared with `USTUB`. If not, declare the stub for `open` with `STUB` so that it loads the module. Declare the other three stubs with `WSTUB` because they must not be called unless the module has already been loaded.

**MODULE Macro**

The stub source file must begin with an invocation of the `MODULE` macro. It takes two parameters: the base module name, and a flag indicating whether or not the base module is unloadable. The possible values of this flag are `STUB_UNLOADABLE` or `STUB_LOADONLY`.

```
NOTE: There is a small performance penalty for each call into a module that has autoload stubs with `STUB_UNLOADABLE`.
```

For example:

```plaintext
MODULE(sample, STUB_UNLOADABLE)
```

**STUB Macro**

The `STUB` macro declares an autoload stub that loads the base module if it is not already loaded. It optimizes performance by sacrificing support for unloading of the base module. Use this macro only if the `MODULE` macro was invoked with the `STUB_LOADONLY` parameter.

The `STUB` macro has three parameters: the base module name, the function name, and the name of a core kernel function to call if the module load fails.
NOTE: The core kernel provides mod_zero, mod_minus, mod_einval, and mod_enoload. These functions return 0, -1, EINVAL, and ENOLOAD, respectively.

For example:

STUB(sample, open, mod_einval)

WSTUB Macro

The WSTUB macro declares an autoload stub that does not load the base module if it is not already loaded. It optimizes performance by sacrificing support for unloading of the base module. Use this macro only if the MODULE macro was invoked with the STUB_LOADONLY parameter. The WSTUB macro has three parameters: the base module name, the function name, and the name of a core kernel function to call if the module is not already loaded.

NOTE: The core kernel provides mod_zero, mod_minus, mod_einval, and mod_enoload. These functions return 0, -1, EINVAL, and ENOLOAD, respectively.

For example:

WSTUB(sample, close, mod_einval)

USTUB Macro

The USTUB macro declares an autoload stub that loads the base module if it is not already loaded. It supports unloading of the base module. Use this macro only if the MODULE macro was invoked with the STUB_UNLOADABLE parameter. The USTUB macro has four parameters: the base module name, the function name, the name of a core kernel function to call if the module load fails, and the number of words of arguments that the function expects.

NOTE: The core kernel provides mod_zero, mod_minus, mod_einval, and mod_enoload for use in the third parameter. These functions return 0, -1, EINVAL, and ENOLOAD, respectively.

Example:

USTUB(sample, read, mod_einval, 2)

END_MODULE Macro

After all the autoload stubs are declared with the preceding macros, you must end the source file with the END_MODULE macro. This macro has one parameter, the base module name.

Example:

END_MODULE(sample)

Stub Source File Example

The following is a complete example of a stub source file named sample_stub.mc that contains autoload stubs for the sample module:

MODULE(sample, STUB_UNLOADABLE)
USTUB(sample, open, mod_einval, 4)
USTUB(sample, close, mod_einval, 3)
USTUB(sample, read, mod_einval, 2)
USTUB(sample, write, mod_einval, 2)
END_MODULE(sample)

Building a Stub Module

To build a stub module, follow the instructions in Chapter 6 (page 157).
The sample Makefile.bld in the Driver Development Kit contains a rule that builds stub source files (with the .mc extension). It runs the m4 macro processor against the stub source file, using the definitions of the macros in the stubs.m4 file (also in the Driver Development Kit). The result is an assembly language source file, which is assembled into an object file. This object file links with the stub module metadata object file to form the stub module.

**Module Dependencies Using a Stub Module**

This section describes how to define the dependencies between two modules using autoload stubs. If a module is dependent on a base module for a function, it must have a dependency on a stub module associated with base module, not directly on the base module.

For example, Module B (modB) is the base module, Module A (modA) is the dependent module on base module, and modB_stub is the stub module defined for the base module. Module A (modA) modmeta is as follows:

```
modA.modmeta -
module modA{
  desc          "ModuleA DLKM MODULE"
  type           misc
  version 1.0.0
  states         loaded static
  loadtimes      driver install run
  unloadable
  dependency     modB_stub 1
}
```

1. Calls out a dependency on base module B for its functions. Module A is calling out a dependency on modB_stub (the stub file), not on modB directly.

Module B (modB) modmeta is as follows:

```
modB.modmeta -
module modB {
  desc          "ModuleB DLKM module with autoload stubs"
  type           misc
  version 1.0.0
  states         auto loaded static 1
  loadtimes      driver install run
  unloadable
  dependency     modB_stub 2
  exports only {
    symbol          func1_modB 3
    symbol          func2_modB 3
  }
}
```

1. Defines the auto state for autoloading.
2. Calls out a dependency on its stub (modB_stub) for functions, as in module A.
3. Exports the func1_modB and func2_modB functions to all other modules.

Module B (modB) modmeta for the module B stub module is as follows:

```
modB.modmeta -
module modB stub {
  desc          "Autoload stub for DLKM module modB"
  type           misc
  version 0.1.0
  states         loaded static
  loadtimes      driver install run
  unloadable
  initfunc         driver install modB_stub_load static 1
  exports only {
    symbol          func1_modB 2
    symbol          func2_modB 2
  }
}
```
symbol  modB_modinfo
}
}

1. Calls modB_stub_load. The driver must call the modB_stub_load function at driver
install time, if modB is statically bound to the kernel.
2. Defines the func1_modB and func2_modB export functions. The functions are exported
by base module B.
3. Exports modB_modinfo. You must list this symbol in the exports only section; without
it the autoload stub cannot function. You do not need to define modB_modinfo. This symbol
is defined by the MODULE macro specified in the stub module source file.

The source file for the stub module is defined as follows:

modB_stub.me -
MODULE(modB, STUB_UNLOADABLE)
USTUB(modB, modB_func1, mod einval, 1)
USTUB(modB, modB_func2, mod einval, 1)
END MODULE(modB)

Autoload Procedure

Using the example in the previous section, an autoload stub resolves the module dependencies
as follows:
1. Puts modB in the auto state.
   This action also registers modB with the DLKM infrastructure. If modB is not set to auto
   state, autoloading on modB fails when modA calls modB's exported functions. Because modB
   depends on modB_stub, modB_stub also loads.
2. Loads modA.
   Because modA has a dependency on modB_stub, modB_stub loads automatically.
3. modA calls modB_func1.
   The call goes to modB_stub. Because this is the first invocation, modB_stub loads modB
   and replaces all the stubs with real functions. Then it jumps to modB_func1.
4. modA calls modB_func2.
   The call goes to modB_stub, but because modB is already loaded, it jumps to modB_func2.
   While modB_stub and modA load, you can unload base module modB, install a new version,
   and put modB back to the auto state; modA is not affected.
5. Unloads modA.
   This unloads modA only. It does not have any effect on the states of modB or modB_stub
   because there is no direct dependency between modA and modB. Therefore, Kernel
   Configuration (KC) does not unload modB when modA is unloaded.
6. Unloads modB.
   If modB is now unloaded, KC also unloads modB_stub, because KC knows the modB_stub
   was loaded only because of a dependency.

Examples

This section shows WSIO and STREAMS driver examples of the various techniques and functions
described in this chapter. Additional examples are available as part of sample drivers in the
Driver Development Kit.
static int
iklpm_load(void *arg)
{
    msg_printf("%s: Enter iklpm_load \n", DRIVER_NAME);
    /*
    * Use the drv_info passed to the driver as an
    * argument instead of using the static version.
    */
    iklpm_wsio_info.drv_info = (drv_info_t *)arg;
    /*
    * Register the driver with WSIO.
    * Note: 0, for failure
    */
    if(!wsio_install_driver(&iklpm_wsio_info)) {
        printf("iklpm_load: wsio_install failure!\n");
        msg_printf("%s: Enter iklpm_load wsio_install_driver" 
" failed !!!\n", DRIVER_NAME);
        return ENXIO;
    }
    /*
    * Add the driver attach function to the WSIO attach
    * list to claim the PCI devices.
    */
    if(mod_wsio_attach_list_add(MOD_WSIO_PCI,
                                iklpm_attach, "iklpm")) {
        /*
        * Attach list add failed: uninstall driver and return
        */
        msg_printf("iklpm - attach_list_add failed...
");
        wsio_uninstall_driver(&iklpm_wsio_info);
        return ENXIO;
    }
    msg_printf("%s: Exit iklpm_load \n", DRIVER_NAME);
    return 0;
}

static int
iklpm_unload(void *arg)
{
    int ret;
    ikoninfo_t *tmp_info;
    struct isc_table_type *isc = NULL;
    msg_printf("%s: Enter iklpm_unload \n", DRIVER_NAME);
    /*
    * Use the drv_info passed to the driver as an
    * argument instead of using the static version.
    */
    iklpm_wsio_info.drv_info = (drv_info_t *)arg;
    /*
    * Driver finds only one card and detaches it.
    */
    while(infolist) {
        isc = infolist->isc;
        if(isc == NULL) {
            return ENXIO;
        } else {
            iklpm_detach(isc);
        }
        tmp_info = infolist;
        infolist = infolist->next;
        if(tmp_info)
            kfree((void*)tmp_info, M_IHV);
    }
    /*
* Remove the driver attach from the DLKM attach list. */
if(mod_wsio_attach_list_remove(MOD_WSIO_PCI, iklpm_attach)) {
    printf("iklpm_unload: mod_wsio_attach_list_remove "
            "failure!\n");
    return ENXIO;
}
/*
 * Uninstall the driver. If it fails, go back to the load state
 * and undo what has been done in the unload routine.
 */
if (wsio_uninstall_driver(&iklpm_wsio_info)) {
    ret = iklpm_attach((uint32_t)isc,
                        (struct isc_table_type*)isc->if_id);
    if (ret)
        panic("iklpm_unload: wsio_uninstall "
              "and re-attach both failed");
    msg_printf("%s: Exit iklpm_unload \n",DRIVER_NAME);
    return 0;
}

STREAMS Driver Load and Unload Functions

STATIC int
dlmsd_load (drv_info_t *drv_info)
{
    printf ("dlmsd_load called\n");
    /*
     * Update the inst_major field of streams_info_t to
     * c_major field of drv_info_t structure.
     */
    dlmsd_mod_strdata.inst_major = drv_info->c_major;
    /*
     * Call str_install
     */
    return (str_install (&dlmsd_mod_strdata));
}
STATIC int
dlmsd_unload (drv_info_t *drv_info)
{
    printf ("dlmsd_unload called\n");
    /*
     * Check if driver can be safely unloaded.
     * Otherwise return a nonzero error code.
     * Free all the allocated resources
     * Uninstall the driver
     */
    return (str_uninstall (&dlmsd_mod_strdata));
}
Kernel modules are pieces of specialized code that run as a part of the operating system kernel, providing services to the entire system. Examples include device drivers and file systems. Because these modules must cooperate with the rest of the kernel, there are special techniques and tools needed to build them and integrate them with the kernel. This chapter gives detailed information about how to build and configure an HP-UX kernel module.

When writing a kernel module, follow these steps:

- **Step 1: Creating a Source Code Directory**
- **Step 2: Creating a Module Metadata File**
- **Step 3: Writing the Module Source Code** (page 164)
- **Step 4: Adding DLKM Support** (page 164)
- **Step 5: Adding Tunable Parameter Support** (page 164)
- **Step 6: Embedding Debug Information** (page 164)
- **Step 7: Embedding Version Information** (page 164)
- **Step 8: Creating Module Preparation Scripts** (page 165)

### Step 1: Creating a Source Code Directory

Create a directory for the module's source code.

### Step 2: Creating a Module Metadata File

Each kernel module has a module metadata file, which describes the module's characteristics and capabilities. Each archive library must also have a module metadata file, independent of the module metadata files for the modules in the library. The format of a module metadata file is defined in `modmeta(4)`, which is available in the *HP-UX 11i v3 Driver Development Kit*. The DDK is at:


This manpage is available in the `/opt/ddk/11.31/BE` directory.

Module metadata files replace the master files used in previous releases of HP-UX, and contain the following types of information:

- Module name (required)
- Module version (required)
- Module type (required)
- Description (required)
- Supported states (required)
- Supported load times (required for modules that support the `loaded` state)
- Unloadable flag
- Dependencies on other modules or interfaces the module exports
- Interfaces or symbols exported by the module
- Tunable parameters
- Initialization functions
- Driver details (required for device driver modules)

The information can appear in any order within the `module{}` block.
## Module Metadata Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module Name</td>
<td>Each module must have a name. Names start with a letter, and consist of letters, digits, and underscores. Names can be at most 32 characters.</td>
</tr>
<tr>
<td><strong>NOTE:</strong> Names of STREAMS drivers and STREAMS modules are limited to 8 characters. Typical module names are 4-10 characters, usually in lower case. Choose simple, easy-to-remember module names. For driver modules, see the driver naming guidelines in the Chapter 4 (page 77).</td>
<td></td>
</tr>
<tr>
<td>Module Version</td>
<td>Each module must have a version. A module version is a series of three decimal integers separated by periods, such as 1.0.0 or 2.4.12. Each integer is limited to a maximum of six digits. The first integer is the major version number. Change this integer when the module changes in a completely incompatible fashion. The second integer is the minor version number. Increment this integer when the module changes in a backwards compatible fashion. The third integer is the revision number. Change this integer when the module changes in an externally invisible fashion. New modules typically start with a 1.0.0 version.</td>
</tr>
<tr>
<td><strong>NOTE:</strong> The version number must be changed each time the module code is changed. If this does not happen, the kernel configuration tools can continue to use the unchanged code, because they cannot see the change.</td>
<td></td>
</tr>
<tr>
<td>Module Type</td>
<td>Each module has a type. The supported module types are as follows:</td>
</tr>
<tr>
<td>filesys</td>
<td>File system module</td>
</tr>
<tr>
<td>misc</td>
<td>Miscellaneous module</td>
</tr>
<tr>
<td>streams_driv</td>
<td>STREAMS driver module</td>
</tr>
<tr>
<td>streams_mod</td>
<td>STREAMS module</td>
</tr>
<tr>
<td>wsio_class</td>
<td>WSIO class driver module</td>
</tr>
<tr>
<td>wsio_intfc</td>
<td>WSIO interface driver module</td>
</tr>
<tr>
<td><strong>NOTE:</strong> Native DLPI drivers, which are STREAMS/WSIO hybrids, are streams_driv type.</td>
<td></td>
</tr>
<tr>
<td>Description</td>
<td>Each module must have a description, which is displayed by kcmodule and kcweb on request. The description must expand any acronyms in the name of the module, and briefly explain the module’s purpose. If the module is not supplied by HP, the description must identify the supplying company. The description has a maximum of 80 characters.</td>
</tr>
<tr>
<td>Supported States</td>
<td>The states statement in the module metadata file explicitly specifies the states the module supports:</td>
</tr>
<tr>
<td>static</td>
<td>Specifies the traditional model that statically binds the module into the main kernel executable, /stand/vmunix.</td>
</tr>
</tbody>
</table>
loaded  For dynamically loaded modules, specifies that the module is forced to be loaded. For more information on dynamic loading, see “Writing a DLKM Driver” (page 137).

auto  For dynamically loaded modules, specifies that the module is loaded only when its services are needed. For more information on dynamic loading, see “Writing a DLKM Driver” (page 137).

Most modules must support these states.

In addition to these three states, an administrator can put a module into one of the following states, with support implied in the module metadata file:

unused  Specifies that the module is not in use. All modules must support the unused state. Do not list the unused state in the states statement.

best  Specifies that the module be put into the state identified as its best state. The first state listed on the states statement indicates the best state of the module. Typically, this is auto, if supported by the module. Otherwise, loaded, if supported by the module. Otherwise, static. This state is used when an administrator requests the module, but does not specify a state.

NOTE:  A module inherits any changes that HP makes to the best state for a module in a patch or a future release of HP-UX.

Administrators control the use of the module by changing its state using the kcweb GUI, the kcmodule command, or in a system file like /stand/system. Table 6-1 shows what happens when an administrator requests a module state change.

Table 6-1 Module State Change

<table>
<thead>
<tr>
<th>From State</th>
<th>To State</th>
<th>static</th>
<th>loaded</th>
<th>auto</th>
</tr>
</thead>
<tbody>
<tr>
<td>unused</td>
<td></td>
<td>vmunix rebuilt with module; used at next boot.</td>
<td>Module loaded dynamically. No reboot.¹</td>
<td>module marked for load at next use. No reboot.²</td>
</tr>
<tr>
<td>static</td>
<td></td>
<td>vmunix rebuilt without module; is unused at next boot.</td>
<td>vmunix rebuilt without module; is loaded dynamically during next boot.</td>
<td>vmunix rebuilt without module; marked for load at first use after next boot.</td>
</tr>
</tbody>
</table>

¹ For loaded modules, the module is loaded on the next reboot.
² For loaded modules, the module is loaded on the next reboot.
### Table 6-1 Module State Change (continued)

<table>
<thead>
<tr>
<th>From State</th>
<th>To State</th>
<th>State</th>
<th>loaded</th>
<th>auto</th>
</tr>
</thead>
<tbody>
<tr>
<td>unused</td>
<td>static</td>
<td>loaded dynamically. No reboot.(^2)</td>
<td>Module remains loaded; marked for load at first use after next boot.</td>
<td></td>
</tr>
<tr>
<td>loaded</td>
<td>vmunix rebuilt with module; used at next boot.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>auto</td>
<td>Module not loaded at next use or at first use after next boot.</td>
<td>vmunix rebuilt with module; used at next boot.</td>
<td>Module loaded dynamically. No reboot.(^2)</td>
<td></td>
</tr>
</tbody>
</table>

1. If the module is busy, the change is held for next boot.
2. To force the change to hold for next boot, add the \(-h\) flag to the \texttt{kcmodule} command.

#### Supported Load Times

Each module that supports the \texttt{loaded} state must specify one or more dispatch points at which they can be safely loaded. For more information, see Chapter 5 (page 137).

The first load time listed on the \texttt{loadtimes} line indicates the preferred load time for the module. This load time is normally used. Other load times are used only if needed to satisfy dependencies or post-boot administrator requests.

#### Unloadable Flag

Once loaded into kernel memory, some modules can never be unloaded.

**IMPORTANT:** HP strongly recommends that all loadable modules support unloading.

If a module supports unloading, set the \texttt{unloadable} statement in the module's metadata file. For more information, including restrictions on what types of modules can support unloading, see Chapter 5 (page 137).

Only modules that support the \texttt{loaded} state can be marked \texttt{unloadable}.

#### Dependencies

The HP-UX kernel modules cannot function without services provided by other kernel modules. The dependencies of a kernel module must be listed in its module metadata for the kernel configuration tools to ensure they are satisfied. HP-UX supports two kinds of dependencies for kernel modules: dependencies on other modules, and dependencies on interfaces. (In this context, interfaces are named and versioned collections of global symbols, typically functions, provided by a kernel module.

In the first case, the metadata for module A can specify a dependency on version 2.1 of module B. This dependency can be satisfied only if the kernel configuration contains a compatible version of module B.

In the second case, the metadata for module A can specify a dependency on version 4.0 of interface C. This dependency can be satisfied by any module that exports a compatible version of interface C.
Exports

Any module can declare that it exports a specific version of a specific named interface. This enables dependencies of other modules on that interface to be satisfied. Most modules do not export interfaces.

Modules can hide all global symbols (functions and variables) except those specifically identified for external use. This improves modularity and reduces the potential for namespace collisions between modules. It also enables improved compiler optimizations for hidden symbols. When using this technique, only those symbols identified in the module metadata can be referenced by code outside the module. Functions and variables accessed, for example, through pointers, switch tables, and initialization functions, can be used by code outside the module. The restriction applies only to the use of the functions and variables by name.

NOTE: To be exported, the symbols must start with a letter and be 32 characters or less.

Modules that support the loaded state automatically hide all global symbols (except for the module wrapper data structure). If such a module needs other symbols to be visible externally, it must list them in the module metadata. For more information, see Chapter 8 (page 183). (The module wrapper data structure need not be listed; it is handled automatically.)

Tunable Parameters

Any module can declare tunable parameters. These are integer variables that control the behavior of the kernel module. Their values are set by the system administrator. For more information, see Chapter 8 (page 183).

Initialization Functions

Many modules have tasks that must be performed during module initialization. Depending on when modules are initialized, the facilities needed to perform those tasks might not be available. For that reason, module initialization tasks are often put in initialization functions registered for specific dispatch points. A dispatch point is a point in time during the kernel boot process at which particular tasks must be performed. For example, initializing tunable parameters.

When a module registers an initialization function at a dispatch point, the system uses the following algorithm. If the module is loaded or statically bound when the kernel boot process reaches the dispatch point, the initialization function is called at that time. Otherwise, the initialization function is called when the module loads.

Modules register initialization functions at particular dispatch points, with the initfunc statement in their module metadata file as follows:
Some initialization functions must be called only when the module is in a particular state. For example, `driver_install` initialization functions must be called only when a module is in the `static` state. In these cases, add the required `module_state` at the end of the `initfunc` statement. If you do not specify a `module_state`, the initialization function is called regardless of the module state. When multiple initialization functions are defined for the same dispatch point, the order in which they are called is undefined. They can be called in parallel.

Dispatch points are also the points at which modules load. For more information on module load times, see Chapter 5 (page 137).

The following dispatch points are available:

- **tunable_init**: Tunable initialization functions must be registered at this dispatch point. Each module that defines tunable parameters must have one or more such functions.

  ![NOTE:](image) If a tunable initialization depends on knowing system memory, this initialization must be done on or after the `driver_install` dispatch point.

- **early_boot_load**: Modules that must be loaded early in the kernel boot process must list this point as their preferred load time. Few kernel services are available at this time, so most modules cannot support loading at this time.

  ![NOTE:](image) Only `misc` type modules can be loaded at this point. This dispatch point is supported on Integrity systems only.

- **driver_install**: Device driver modules must list this point as their preferred load time. All driver modules must have an installation function registered at this point. Most kernel services are available at this time, so most dynamically loadable modules can support loading at this time.

- **run**: This is not a real dispatch point, but it can be used in a module `loadtimes` statement to indicate that a module can be loaded after the kernel boot process is complete. Most dynamically loadable modules must support `run`. Initialization functions cannot be registered for this point.
Driver Information

Device driver modules can declare additional information about their driver characteristics. Some of this information is required for drivers that support the loaded state. This additional information includes:

- **Driver Type**
  Indicate block and character devices with the `block` and `char` statements, respectively (or both `block char`, if appropriate).

- **Driver Class**
  The driver class specifies the class to which the driver belongs. This statement is required for drivers that support the loaded state.

- **Driver Flags**
  The possible flags for drivers include:
  - `save_conf`
  - `mp_safe`
  - `scan`
  - `defer_scan`
  - `blklist`
  - `map_buf_to_kern`
  - `allcloses`
  - `clonesmaj`
  - `svr3_open`
  - `svr4_open`
  For information about these flags, see Chapter 4 (page 77).

Driver Structure Definitions

Every driver must have a `drv_info` structure containing basic information about the driver. This structure is described in detail in Chapter 4 (page 77). If a `driver{}` section is provided in the module metadata (required for loadable drivers), this structure is automatically generated based on the information in that section. The structure generated is named `modulename_drv_info`.

Sample Device Driver Module Metadata File

Following is an example of a module metadata file for a device driver. This driver, `mydriver`, supports all possible module states, and can be loaded with the other drivers during boot or manually after boot is complete. It has an initialization function to register itself with the driver infrastructure when it is in static state. (Its load function serves the same purpose when it is in loaded state.) It is dependent on the WSIO services in the core kernel. It is a character (raw) device.

```plaintext
module mydriver {
    desc       "My Sample Driver"
    type       wsio_intfc
    version    1.0
    states     auto loaded static
    loadtimes  driver_install run
    dependency wsio
    initfunc   driver_install mydriver_install static
    initfunc   mod_load       mydriver_load
```

Step 2: Creating a Module Metadata File
Step 3: Writing the Module Source Code

Write the source code for the module. For WSIO driver modules, see Chapter 4 (page 77). For STREAMS drivers and modules, see the STREAMS/UX Reference Manual.

Step 4: Adding DLKM Support

If the module supports dynamic loading, write the dynamic loading data structures and the load and unload functions for the module. For information, see Chapter 5 (page 137).

Step 5: Adding Tunable Parameter Support

If the module defines tunables, you must write a tunable initialization function for the module, tunable handler functions for each tunable, and tunable teardown code in the module unload function. In addition, you must document your tunables and their behavior. For more information, see Chapter 8 (page 183).

Step 6: Embedding Debug Information

It is easier to debug problems with a module if the kernel debuggers have knowledge of the data structures used by the module. This is true both during development and after release. For this reason, it is common practice to include debug information about the data structures in the module.

The C compiler generates debug information about a data structure whenever the structure is visible to a source file compiled with the -g flag. However, that flag also causes the compiler to generate source line debug information when it compiles source files containing actual code. Source line debug information can be useful during development and unit testing of a module, but it cannot be released because it makes the kernel executable too big to support. HP does not support modules that contain source line debug information, or systems that have such modules installed.

For this reason, HP recommends that you create a separate C source file containing only data structure debug information. This source file contains #include directives that makes all of the module data structures visible, but it contains no actual code. This source file is compiled with -g and linked into the module. As a result, all the module data structure debug information is included in the module. The other source files in the module are compiled without -g, so the module contains no source line debug information.

Step 7: Embedding Version Information

A useful support technique is to embed in a module some identification of its version. Support engineers can determine which version is in use when a problem occurs. The most common technique is to embed a what string. This is any string that begins with the four characters @(#). It is called a what string because the what command searches a binary file for strings starting with those characters.
You can embed a `what` string in a source file by adding a line similar to the following:

```c
#include VERSIONID "@(#) ...
```

However, this requires manually updating the string each time the version is to be updated, which can be a cumbersome process.

**NOTE:** The practice of putting version strings in static `char[]` variables is deprecated because the compiler often eliminates unused variables.

Instead of manually embedding a `what` string, a module's build process can automatically generate a separate C file containing only a `what` string, and to compile and link that C file with the rest of the module. The `what` string can therefore easily contain a timestamp, source code revision string, or other convenient identifier of the module version.

### Step 8: Creating Module Preparation Scripts

Modules can supply a module preparation script, containing commands that run before or after booting, loading, or unloading the module. These scripts are optional. They are documented in `modprep(8)`, available in the HP-UX 11i v3 Driver Development Kit at:


This manpage is available in the `/opt/ddk/11.31/BE` directory.

Common tasks for module preparation scripts include the following:

- Check that a module is compatible with the system before it loads.
- Create device special files for driver modules.
- Make sure a module is not in use before unloading it.

For an example of a module preparation script, see “Sample modprep Script” (page 166).

**NOTE:** These scripts are often run before file systems other than the root are mounted. They cannot rely on the presence of mounted file systems. For example, they can use only those commands that are in `/sbin`.

Module preparation scripts are called as follows:

```
modprep modulename operation [status]
```

The first parameter is the name of the module for which the script is being called. This enables a single script to be used for multiple modules.

The second parameter is the operation that is being performed. The possible operations are `preload`, `postload`, `preunload`, and `postunload`.

For the `preload` and `preunload` operations, there are no additional parameters. The script must perform any necessary actions to prepare the system for loading or unloading the module, and any necessary consistency checks to ensure that the load or unload will be successful and safe. The script must exit with a status code of 0 to allow the load or unload to proceed, or 1 to abort the operation.

For the `postload` and `postunload` operations, the script is given an additional parameter indicating the status of the load or unload operation. The status parameters are as follows:

- **LOAD** The load operation succeeded. The script must perform any necessary `postload` operations.
- **UNLOAD** The unload operation succeeded. The script must perform any necessary `postunload` operations.
- **FAIL code** The load or unload failed; the system call returned `code`. The script must undo any steps taken by the previous `preload` or `preunload` operation.
The module was loaded during boot or statically bound into the kernel. The script must perform both preload and postload actions because the script was not previously called for preload operation.

The module load or unload was aborted due to a problem with some other module. The script must undo any steps taken by the previous preload or preunload operation.

Scripts are called for preload only when loading a module at run time. They are not called before loading a module during the kernel boot process, for statically bound modules, or for modules that are autoloaded at first use.

Scripts are not called for postload for modules that are autoloaded. They are called for modules loaded during boot, loaded at run time, or statically bound into the kernel.

Scripts are not called for preunload or postunload during system shutdown. They are called only for modules unloaded at run time.

Any time a module preparation script is called with an argument it does not recognize, it takes no action, prints no message, and exits with an exit code of 2.

### Sample modprep Script

This section contains examples of the various techniques and functions described in this document. Additional examples are available as part of sample drivers in the Driver Development Kit.

```bash
#!/sbin/sh
#
# Copyright (C) 2007 Hewlett-Packard Company.
#
# @(#) Simple modprep script example.
# preload: no op
# postload: create device special file
# preunload: show processes that are accessing the device
# file if any
# postunload: remove device special file
#
# See modprep(9E) for information on how and when this script is invoked.

#PATH=/sbin:/usr/sbin:/usr/bin
export PATH

modulename=$1; shift
operation=$1; shift

case $operation in
  'preload')
    # no-op    exit 0
  ;;

  'postload')
    status=$1; shift
    if [[ $status = 'FAIL' || $status = 'ABORT' ]]; then
      # failed to load. Nothing to do.
      exit 0
    fi
    if [[ $status != 'LOAD' && $status != 'BOOT' ]]; then
      # Unknown command. Exit with no error/warning message.
      exit 2
    fi
    # Create /dev/moduledev
    MOD_MAJOR=$(lsdev -h -d $modulename | awk '{print $1}')
    if [[ "$MOD_MAJOR" == "" ]]; then
      # FATAL ERROR: Impossible.
    fi
  esac
```

166 Writing a Kernel Module
Step 9: Building a Kernel Module

To build a kernel module, follow these steps:
1. Put all the source code for the module into a directory on an HP-UX system. The system must be running the version of HP-UX for which the kernel module is intended. In addition, the system must have the HP ANSI C compiler and the HP-UX 11i v3 Driver Development Kit installed.

**NOTE:** This is necessary for nondriver modules. The DDK contains the `kmsecgen`, `modlink`, and `modmeta` build tools that are needed for all module types.

The DDK is available at:

http://www.hp.com/go/hpux_ddk

It installs the necessary files into the `/opt/ddk/11.31/BE` directory.

2. Copy the sample makefiles (`Makefile` and `Makefile.bld`) from the DDK into the module source directory. Tailor the makefiles to the needs of the module being built, following the instructions in the comments in those files.

### Makefile

The sample `Makefile` is sets up the build environment and invokes `Makefile.bld`, which performs the actual module build. `Makefile` also has some convenience rules to help with common tasks during development of the module.

Customize the `Makefile` by setting the values of the `ARCH`, `MODULE`, `DEV_B`, and `DEV_C` variables. Set the `ARCH` variable to the architecture for which the module is being built. The default is the architecture on which the build is running; a native build.

Set the `MODULE` variable to the name of the module being created. This must match the module name as specified in the module metadata file.

To use the `dev` target in the `Makefile` to create device special files for the module during development, set the `DEV_B` or `DEV_C` variables, or both. The `DEV_B` and `DEV_C` variables are set to the path names of the device special files, if any, for the module. `DEV_B` is the block file and `DEV_C` is the character file. Uncomment the rules corresponding to the variables that you set.

The `Makefile` default all target builds the module by invoking `Makefile.bld` with the appropriate build environment.

`Makefile` supports the following other targets for use during development and unit testing of the module:

- `dev` Creates the device special files for the module.
- `install` Copies the module to the module directory.
- `load` Starts using the module.
- `status` Displays the status of the module.
- `unload` Stops using the module.
- `uninstall` Removes the module from the module directory.
- `clobber` Removes device special files and build output.

### Makefile.bld

The sample `Makefile.bld` is responsible for the actual module build after the build environment has been set up by the sample `Makefile`.

Customize the `Makefile.bld` by setting the `OBJJS` and `HEADERS` variables. Set the `OBJJS` variable to the list of object files that are a part of the module. This is a single module metadata object file, and one or more compiled C or assembly language source files.
The HEADERS variable can be set to the list of header files used by the source files for the module. These are listed as make dependencies for the source files. Alternatively, the exact dependencies for each file can be listed on the target line for each file at the bottom of the file.

The middle section of Makefile.bld defines the preprocessor definitions, compiler options, and linker options needed for the build.

**IMPORTANT:** The definitions and options listed in the sample Makefile.bld are those used to build the core HP-UX kernel code. Use caution when making any changes to them.

The bottom section of Makefile.bld contains target lines and build rules for the module. The rule for the module uses the modlink tool to link the object files for the module into a single module object file. For modules that support dynamic loading, it also adds dynamic loading data to the module.

The `${MODULE}.o` rule in Makefile.bld causes the file to be compiled with the necessary preprocessor definitions and command-line options. Copy and modify this rule as needed for the source files of the module. Create one such rule for each target listed in the OBJS variable.

**NOTE:** Compiler options shown in the sample Makefile are critical. Using different options can result in a module that does not build, load, or function properly.

If the module has separate source files containing debug information or version information, as described in “Step 6: Embedding Debug Information” (page 164) and “Step 7: Embedding Version Information” (page 164), the corresponding object files must be listed in the OBJS variable. Build rules are needed for each of them.

The rule for the module metadata file runs the modmeta translator, then compiles the resulting C file with the appropriate preprocessor definitions and command line options. Use the #include statement in other source files in the module to include the header file.

### Command Line Options

This section describes the command line options used in the sample Makefile.bld. For more information, see cc(1).

**Table 6-2 Sample Makefile.bld Compiler Options**

<table>
<thead>
<tr>
<th>Command Option</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Ae</td>
<td>Extended ANSI mode</td>
</tr>
<tr>
<td>+DD64</td>
<td>64-bit data model</td>
</tr>
<tr>
<td>+DSitanium2</td>
<td>Tune for Itanium processors (Integrity only)</td>
</tr>
<tr>
<td>+kernel</td>
<td>Compiling kernel code</td>
</tr>
<tr>
<td>+O2</td>
<td>Level 2 optimization</td>
</tr>
<tr>
<td>+Oshortdata=0</td>
<td>No short data references (Integrity only)</td>
</tr>
<tr>
<td>+objstatvars</td>
<td>Include static variables in the symbol table (Integrity only)</td>
</tr>
<tr>
<td>+DS2.0</td>
<td>Tune for PA-RISC 2.0 (PA-RISC only)</td>
</tr>
<tr>
<td>+ES1.Xindirect_calls</td>
<td>Special handling of indirect function calls (PA-RISC only)</td>
</tr>
<tr>
<td>+ESsfc</td>
<td>Inline code for function pointer comparisons (PA-RISC only)</td>
</tr>
<tr>
<td>+ESssf</td>
<td>8-byte aligned stack (PA-RISC only)</td>
</tr>
<tr>
<td>+ordering_unaware</td>
<td>Ordered completion of volatile load and store (PA-RISC only)</td>
</tr>
<tr>
<td>+Oentriesched</td>
<td>Instruction scheduling in entry and exit code (PA-RISC only)</td>
</tr>
</tbody>
</table>
Table 6-2 Sample Makefile.bld Compiler Options (continued)

<table>
<thead>
<tr>
<th>Command Option</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>+Ofastaccess</td>
<td>Fast access to global data (PA-RISC only)</td>
</tr>
<tr>
<td>+Omultiprocessor</td>
<td>Optimize for multiprocessor code (PA-RISC only)</td>
</tr>
<tr>
<td>-Wp,-H300000</td>
<td>Increase preprocessor macro table size (PA-RISC only)</td>
</tr>
<tr>
<td>+Xd</td>
<td>Do not store registers to memory for debugging (PA-RISC only)</td>
</tr>
<tr>
<td>+Xi</td>
<td>Do not use dyncall for position-independent code (PA-RISC only)</td>
</tr>
</tbody>
</table>

Step 10: Installing a Kernel Module

To put a kernel module on an HP-UX 11i v3 system, move the kernel module file (name is the module name) to /usr/conf/mod on the target system. If the module has a preparation script, move that file to the same location. Its name must be modulename.prep.

Package kernel modules delivered to customers in Software Distributor (SD) depots. For more information on creating depots, see Chapter 23 (page 511).

To add a depot containing a kernel module into the kernel configuration, the depot must contain a postinstall script that uses the mod_systemfile utility. For more information, see “Step 8: Creating Module Preparation Scripts” (page 165).

When a depot containing a kernel module is uninstalled, the module must be removed from the kernel configuration. To do so, the depot must contain a preremove script that uses the mod_systemfile utility to remove the kernel module from the configuration.

Step 11: Testing and Using a Kernel Module

After a module is installed on a system, bring it into the kernel configuration using the kcmodule command. For more information, see kcmodule(1M). Run kcmodule modulename=state, where state is any of the states supported by the module (loaded, static, and auto). This command changes the state of the module to the requested state.

For example, to bind a module into the static kernel, run kcmodule modulename=static. To load a module dynamically, run kcmodule modulename=loaded. To mark a module for automatic loading, run kcmodule modulename=auto. And to stop using a module (to unbind it from the static kernel or to unload it dynamically), run kcmodule modulename=unused. The kcmodule command rebuilds the static kernel when needed. It prints an appropriate message when a reboot is required to complete the change.

Before testing a module, save a copy of a known working kernel configuration so that it is available in case the system cannot run with the module under test. To save a kernel configuration, use kconfig -s. To boot the saved configuration on an Integrity server, specify the following boot loader command line:

HPUX> boot configname

To boot the saved configuration on a PA-RISC platform, specify the following boot loader command line:

ISL> hpux configname/vmunix

During testing, if a module shows that it must be changed, update the version number at each change. Failure to do so may cause the kernel configuration tools to continue using the unchanged code because they do not know that a change was made.

To update the module on a test system, remove the old module from the kernel configuration, replace the module file on disk, and then bring the module back into the configuration. For example:
Do not use the kcmodule command from an SD depot control script like postinstall or preremove. Always use the mod_systemfile utility from such scripts.

Developers must test their modules in each of the states they support, particularly if the module has code that runs during the boot process. Set the module to the desired state, then reboot the system to ensure that the module behaves properly during shutdown and reboot.

If a module fails to load, see the /var/adm/syslog/syslog.log file for information about the failure.

If the module has a preparation script, test the operation of that script in both success and failure cases.

### Updating an Older HP-UX Kernel Module

To update an older HP-UX kernel module, follow these steps:

1. Create a module metadata file for the module. It must be called modulename.modmeta. For an explanation, see “Step 2: Creating a Module Metadata File” (page 157). For syntax details, see modmeta(8).

2. If the module will support dynamic loading, write the dynamic loading data structures and load and unload functions for the module. See Chapter 5 (page 137).

3. If the module defines tunables, write a tunable initialization function for the module and tunable handler functions for each tunable. See Chapter 8 (page 183).

4. Write manpages for each tunable.

5. Add tunable teardown code in the module's unload function.

6. Copy the sample Makefile and Makefile.bld from the Driver Development Kit and modify them for the module. For modules in UXSCM, create a Files file for the module and add it to the list of Files files. See “Step 9: Building a Kernel Module” (page 167).

7. Modify the SD package so that it delivers only the module object file and module preparation script (if any), and any necessary manpages, user space code, and so on. See “Step 10: Installing a Kernel Module” (page 170).
Kernel Registry Services (KRS) is an HP-UX feature that provides a common method of accessing and maintaining the persistent structured data. This feature eliminates the need for individual kernel modules to create files or other mechanisms to store persistent data. In addition, the KRS data is available to the kernel earlier in the boot cycle than traditional UNIX system files.

This chapter discusses the following topics:

- “KRS Structure”
- “Adding the KRS Header” (page 178)
- “Defining Data Structures” (page 178)
- “Using KRS Routines” (page 178)

KRS Structure

Information maintained in KRS is structured hierarchically. This tree structure is flexible enough to store and maintain any type of information that KRS clients might require. Because the primary KRS clients are kernel components, the KRS tree is maintained in kernel space. This makes the information readily accessible with minimum latency.

KRS Tree Structure

The KRS tree is composed of a number of nodes. A node has the following characteristics:

- A name
- Zero or more children (also nodes)
- Zero or more values
- One or more parent using links (except the root node). See “Links” (page 175) for more information.

You identify a node in a manner similar to identifying a file with a UNIX file system path, by specifying a path of node names (separated by slashes), starting from the root node and ending at the node in question. Although the file system distinguishes between files and directories, KRS only contains nodes. KRS nodes define the tree’s structure, and like directories they also contain data. Figure 7-1 shows a sample KRS tree. The path specification for the tunables node is:

```
/software/config/tunables
```
Values

KRS data is maintained in the form of values. A value is a named data item of a given type. Table 7-1 shows the KRS value types.

**Table 7-1 KRS Value Types**

<table>
<thead>
<tr>
<th>Value Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KR_VTYPE_STRING</td>
<td>A NULL-terminated text string</td>
</tr>
<tr>
<td>KR_VTYPE_USER</td>
<td>User-defined data of given size</td>
</tr>
<tr>
<td>KR_VTYPE_INT32</td>
<td>A 32-bit integer</td>
</tr>
<tr>
<td>KR_VTYPE_INT64</td>
<td>A 64-bit integer</td>
</tr>
<tr>
<td>KR_VTYPE_UINT32</td>
<td>An unsigned 32-bit integer</td>
</tr>
<tr>
<td>KR_VTYPE_UINT64</td>
<td>An unsigned 64-bit integer</td>
</tr>
<tr>
<td>KR_VTYPE_TREEREF</td>
<td>A reference to a node in the tree</td>
</tr>
<tr>
<td>KR_VTYPE_ADDR</td>
<td>A memory address</td>
</tr>
</tbody>
</table>

1 KRS does not interpret data for this type; it is opaque data.

Keys

KRS keys are similar to open file descriptors. While a path specification identifies a node within the tree, a key identifies an open instance of a node. Keys are returned by KRS open operations and passed to read and write operations. Each key is an opaque handle to an open KRS node.

Nodes

Each node consists of two logical parts: a part that defines where the node resides in the tree (its tree context) and a part that defines the node’s data. Figure 7-2 shows the KRS node structure.
The **tree context** portion of the node contains the following:
- The node's name
- A pointer to the node's siblings (nodes at the same level under the same parent)
- A pointer to the data portion of the node

The **data** portion of the node contains the following:
- A pointer to the node's values
- A pointer to the node's children (nodes for which this node is the parent)

### Links

In KRS, links provide a mechanism to define more than one path to the same data, similar to linked files in a UNIX file system. When two or more nodes are linked, they share the same data portion. Links that would violate the acyclic nature of the tree are not permitted.

**Figure 7-3** shows node A and node B linked together. Each has its own tree context portion; each has its own name and parent. However, they share common data.

**Figure 7-3 Linked Nodes**

Linked nodes not only share the same values, they share the same sub-tree as defined by their common children. **Figure 7-4** shows this relationship.
Tree References

Table 7-1 “KRS Value Types” shows the KR_VTYPE_TREEREFF value type. Values of this type are actually keys for an open node somewhere else in the tree. This tree reference has the following benefits:

- Associates references to other nodes in the tree with the information in the defining node. This is in contrast to links where the nodes share the same data
- Saves the overhead of storing a full path specification string as the value and opening the node after the path value is read
- Tree reference values can be persistent across reboot

Persistence

Because the KRS tree is maintained in kernel memory, the information contained in the tree is normally lost every time the system is rebooted. To prevent this information loss, KRS provides a mechanism by which you can flag information to be maintained across reboot. Information flagged as being persistent is periodically saved to files on disk. When the system is booted, the information in these files is used to reconstruct the KRS tree.

Classes of Persistent Information

In KRS, all persistent information is one of the following:

- System-specific: Information that applies to the entire system. For example, the system’s name and hardware configuration.
- Kernel-specific: Information that applies to the kernel running on the system. For example, configured kernel modules and tunable values.
When persistent information is saved in files on disk, system- and kernel-specific information are saved in separate files. This supports the capability of booting different kernels on the same system.

System-specific information is maintained across system reboots and system updates (for example, patch installs). Kernel-specific information is stored in files associated with the kernel that generated the information (the kernel that was running when the tree was saved). When a specific kernel is booted, the tree is reconstructed from the system-specific information and the kernel-specific information associated with the kernel in question. The information is not valid across system updates.

### Inheritance

A node can inherit a value from an ancestor node. The following flags govern this behavior:

- **KR_EXPORT**: Children and grandchildren can use this value.
- **KR_NO_INHERIT**: Children and grandchildren cannot use this value.

If you do not specify the **KR_NO_INHERIT** flag and a value is not found on the target node, KRS searches each of the target node’s ancestors for the value.

### Flags

Many of the KRS interface functions expect a collection of flags as one of their arguments. These flags, used to convey information to the functions in question, are divided into two sets: **attribute flags** and **action flags**.

#### Attribute Flags

Attribute flags define various qualities of the information in the tree and can be applied to both nodes and values. Table 7-2 describes these flags.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KR_PERSISTENT</td>
<td>Item is persistent across reboot</td>
</tr>
<tr>
<td>KR_USRBUF</td>
<td>Use client’s buffer to hold value¹</td>
</tr>
<tr>
<td>KR_EXPORT</td>
<td>Value can be inherited</td>
</tr>
<tr>
<td>KR_USER_RDONLY</td>
<td>Item is read-only from user space</td>
</tr>
<tr>
<td>KR_PRUNE</td>
<td>Do not save the node or its ancestors to persistent storage</td>
</tr>
</tbody>
</table>

¹ The caller must maintain memory for a value with this attribute.

#### Action Flags

Action flags govern the behavior of the functions to which they are passed. Table 7-3 describes these flags.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KR_CREATE</td>
<td>Create node if it does not exist</td>
</tr>
<tr>
<td>KR_ALL</td>
<td>Create intermediate nodes as needed</td>
</tr>
<tr>
<td>KR_LINKS</td>
<td>Delete links with node</td>
</tr>
<tr>
<td>KR_PATH</td>
<td>Get path node names</td>
</tr>
<tr>
<td>KR_NO_INHERIT</td>
<td>Do not inherit values</td>
</tr>
<tr>
<td>KR_NOWAIT</td>
<td>It is not permissible to sleep</td>
</tr>
</tbody>
</table>
Adding the KRS Header

The main KRS header file, /usr/include/sys/krs.h, contains data definitions, constants, macros, and options used by the KRS calls and subroutines. All kernel modules that use KRS routines must include this header file.

Defining Data Structures

The following KRS data structures are used by the KRS routines:

- kr_ket_t
  uint64_t kr_key_t;
- kr_flags_t
  uint64_t kr_flags_t;
- kr_size_t
  uint32_t kr_size_t;
- kr_type_t
  uint32_t kr_type_t;
- kr_flag_field_t
  uint32_t kr_flag_field_t;
- kr_linkid_t
  #ifdef _KERNEL typedef void * kr_linkid_t;
  #else /* !_KERNEL */ typedef uint64_t kr_linkid_t;

Using KRS Routines

This section describes the steps to write a kernel module that uses the KRS routines:

1. Open a KRS node
2. Link a KRS node (optional)
3. Perform operations on a node
4. Close a KRS node
5. Delete a KRS node
6. Release a KRS reference

Opening a KRS Node

Before performing any operations on a KRS node, you must first acquire a key by calling kr_open_node.

kr_open_node
int kr_open_node(
    kr_key_t root_key,
    char * path,
    kr_flag_t flags,
    kr_key_t *new_key
) ;

See kr_open_node(9F) in the Driver Development Reference manual for more information.

The following code example shows how to use the kr_open_node routine to create a KRS node:

int kr_open_node(kr_key_t, char *, kr_flags_t, kr_key_t *); int kr_close_node(kr_key_t); myfunc(){
int rv;
kr_key_t mykey;
rv = kr_open_node(KR_NOKEY, "/mynode", KR_CREATE, &mykey);
if (rv != KR_SUCCESS) {
    return ERROR;
}

kr_close_node(mykey);

Linking a KRS Node

kr_link_node
int kr_link_node(
    kr_key_t skey,
    kr_key_t root_key,
    char *path,
    kr_flag_t flags,
    kr_key_t *new_key
);

See kr_link_node(9F) in the Driver Development Reference manual for more information.

Performing Operations on a KRS Node

Use the following KRS routines to perform operations on a KRS node after you have opened a node or linked to a node:

kr_get_value
int kr_get_value(
    kr_key_t key,
    char *vname,
    kr_type_t *type_buf,
    kr_size_t *size_buf,
    void *buf,
    kr_flags_t flags
);

See kr_get_value(9F) in the Driver Development Reference manual for more information.

kr_set_value
int kr_set_value(
    kr_key_t key,
    char *vname,
    kr_type_t *type,
    kr_size_t *size,
    void *buf,
    kr_flags_t flags
);

See kr_set_value(9F) in the Driver Development Reference manual for more information.

kr_delete_value
int kr_delete_value(
    kr_key_t key,
    ()vname,
    kr_flags_t flags
);

See kr_delete_value(9F) in the Driver Development Reference manual for more information.
int kr_get_vinfo(kr_key_t key, char *vname, kr_type_t *type_buf, kr_size_t *size_buf, kr_flag_field_t *flag_buf);

See kr_get_vinfo(9F) in the Driver Development Reference manual for more information.

int kr_get_node_info(kr_key_t key, kr_flag_field_t *info_buf, kr_flag_field_t *vtyp_buf, kr_flag_field_t vclass, kr_flag_field_t vphase, kr_flag_field_t *flag_buf, kr_linkid_t *lid_buf);

See kr_get_node_info(9F) in the Driver Development Reference manual for more information.

int kr_get_node_names(kr_key_t key, char *names[], int *n_name, char *buf, kr_size_t *buf_sz, kr_flags_t flags);

See kr_get_node_names(9F) in the Driver Development Reference manual for more information.

int kr_get_value_names(kr_key_t key, char *names[], int *n_name, char *buf, kr_size_t *buf_sz);

See kr_get_value_names(9F) in the Driver Development Reference manual for more information.

int kr_set_node_flags(kr_key_t key, kr_flags_t flags, kr_flag_field_t mode);

See kr_set_node_flags(9F) in the Driver Development Reference manual for more information.

int kr_set_value_flags(kr_key_t key, char *vname, kr_flags_t flags,
kr_flag_field_t mode
);

See kr_set_value_flags(9F) in the Driver Development Reference manual for more information.

kr_get_mod_time
int kr_get_mod_time(
    kr_flag_field_t class,
    kr_flag_field_t class,
    uint64_t *mtime
);

See kr_get_mod_time(9F) in the Driver Development Reference manual for more information.

Closing a KRS Node

kr_close_node
int kr_close_node(
    kr_key_t key
);

See kr_close_node(9F) in the Driver Development Reference manual for more information.

Deleting a KRS Node

kr_delete_node
int kr_delete_node(
    kr_key_t key,
    kr_flag_t flags
);

See kr_delete_node(9F) in the Driver Development Reference manual for more information.

The following code example shows how to use the kr_delete_node routine:

```c
int kr_delete_node(kr_key_t, kr_flag_t);
myfunc()
{
    kr_key_t mykey;
    /* Error checking omitted */
    kr_open_node(KR_NOKEY, "/mynode", KR_NOFLAGS, &mykey);
    kr_delete_node(mykey, KR_NOFLAGS);
    kr_close_node(mykey);
}
```

Releasing a KRS Node Reference

kr_release_reference
int kr_release_reference(
    kr_key_t key
);

8 Adding Tunable Parameters to Kernel Modules

Each kernel module can define any number of tunable parameters (tunables), which are integer variables set by the administrator that control the behavior of a module. Tunables are defined in the module metadata file, initialized by a tunable initialization function in the module’s code, and controlled by tunable handler functions in the module. These functions generate and change the value of the tunable. Then, they call the tunable infrastructure to get or set the tunable value.

To add tunable parameters to a kernel module, follow these steps:

- “Step 1: Defining a Tunable Parameter”
- “Step 2: Adding the Tunable to the Metadata File” (page 185)
- “Step 3: Writing Tunable Handler Functions” (page 185)
- “Step 4: Writing Tunable Initialization Functions” (page 193)
- “Step 5: Writing Tunable Teardown Code” (page 194)
- “Step 7: Creating Tunable Documentation” (page 195)
- “Step 6: Building and Testing the Kernel” (page 195)
- “Additional Tasks” (page 195)

The following sections refer to functions whose names start with `ktune_`. These functions, and the structure and type definitions needed to use them, are declared in the `<sys/ktune.h>` header file. For manpages for these functions, see the `HP-UX 11i v3 Driver Development Reference`.

### Step 1: Defining a Tunable Parameter

Each tunable is defined with a tunable block in the module metadata file. For the file syntax, see `modmeta(4)`. The following information is given in a tunable definition block:

<table>
<thead>
<tr>
<th>Name</th>
<th>Each tunable has a name that must be carefully chosen. Use the following guidelines to choose a tunable name:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• The name cannot exceed 32 characters. Most names are in the 8-16 character range. Names must start with a</td>
</tr>
<tr>
<td></td>
<td>letter, and can contain only letters, digits, and underscores.</td>
</tr>
<tr>
<td></td>
<td>• The tunable name must begin with the module name, module vendor name, or some other mnemonic that ensures</td>
</tr>
<tr>
<td></td>
<td>the tunable name is unique. No two modules are allowed to have tunables with the same name. This rule also</td>
</tr>
<tr>
<td></td>
<td>ensures that related tunables sort together.</td>
</tr>
<tr>
<td></td>
<td>• Separate subsequent words with underscores. Use lower case letters unless the meaning cannot be understood</td>
</tr>
<tr>
<td></td>
<td>without capitalization. Do not start or end a name with an underscore.</td>
</tr>
<tr>
<td></td>
<td>• Omit <code>num</code> and <code>max</code> from tunable names unless they are needed to contrast with another tunable name.</td>
</tr>
<tr>
<td></td>
<td>• Use common abbreviations.</td>
</tr>
</tbody>
</table>

| Description | Each tunable also has a one-line description. Descriptions are limited to 80 characters, and must be clear English without acronyms or jargon. For consistency with others, the description can either be a noun phrase completing the sentence "This tunable is the...", or an active phrase in active voice, such as "Enables xxxx functionality." List the units in parentheses at the end, along with any sentinel values (for example, 0=unlimited). The first word must be capitalized; following words use sentence case. Do not end the description with a period. |
Visibility

Limit the set of visible tunables to those used by the bulk of the customer base. Mark any tunable that is intended for use only by developers or support personnel, or by a specific small number of customers, with the private flag in the tunable definition. Private tunables can be set and queried like other tunables, except that they do not appear in kctune or kcweb lists of tunables. They must be explicitly requested by listing their names on the kctune command line.

Sign

If the tunable takes negative values, the signed flag must be specified in its definition. Otherwise, tunable values are displayed as unsigned quantities.

Percentage

If the tunable accepts values specified as percentages, the percentage_tuneflag must be specified in its definition. It must also have a tunable handler that responds to the KTOP.GETSCALE request.

Default value

The default value for a tunable minimizes the need for administrators to change it. HP strongly encourages the use of algorithms to compute default values whenever applicable.

Failsafe value

The value used when stored tunable value information is not available or when the administrator requests a failsafe boot. It must be a fixed value (not computed at run time), and must be chosen as the safest possible value to use during system recovery operations. This is often a different value than the algorithmically computed default value.

The failsafe value is not specified in the tunable definition in the module metadata. It is given as a parameter to the ktune_get call when the tunable is initialized during boot. See “Step 4: Writing Tunable Initialization Functions” (page 193).

Minimum and maximum values

The minimum (min) and maximum (max) values must encompass the full range of values that the kernel code supports. Do not take into account any limits imposed by hardware, because those can change from one platform to another; they must be checked at run time in the tunable handler function. Do not limit min and max to the range of values that have been tested. This enables you to test a broader range later. Document recommended and tested ranges in the tunable manpage. See the “Step 7: Creating Tunable Documentation” (page 195).

Valid values

Some tunables allow zero as a valid value, though it falls outside the min and max values. Such tunables must have the zero_ok constraint in their tunable definition.

Some tunables are restricted to values that are powers of two. Such tunables can have constraint powerof2 in their tunable definition.

Many tunables have constraints based on the values of other tunables. For example, tunable A must be less than tunable B plus 10. These constraints can also be described in the tunable definition.

The constraints previously listed (min, max, zero_ok, powerof2, and the relationship between two tunables) are enforced automatically by the tunable infrastructure. Tunables can have other constraints on their values, but they are not automatically enforced. The module must supply a tunable handler function to validate any other constraints. For a discussion of tunable handler functions, see “Step 3: Writing Tunable Handler Functions” (page 185).
Step 2: Adding the Tunable to the Metadata File

Add the tunable to the module’s metadata file. For information on the file format, see modmeta(4). The following is an example tunable definition for nkthread from a module metadata file:

```plaintext
tunable nkthread {
    desc       "Maximum number of threads on the system"
    default    8416
    min        200
    max        4194304
    constraint nkthread >= nproc + 100
}
```

Step 3: Writing Tunable Handler Functions

Tunable handler functions contain all the code that manipulates a specific tunable. A tunable must have one or more handler functions if:

- Any constraints on the tunable’s value cannot be specified in the module metadata.
- The tunable has a computed (rather than hard-coded) default value or computed minimum or maximum.
- The tunable can be changed without a reboot.
- Modules are to be notified when the tunable is changed.

**NOTE:** HP strongly recommends that all public tunables support changes without a reboot. Computed defaults are encouraged to minimize the need for administrator intervention.

Following is the prototype for a tunable handler function:

```plaintext
int mytunehandler (  
    ken_id_t eventid,  
    int reason,  
    ken_instance_t instance,  
    void *handler_data,  
    ktune_event_t *event_data,  
    int *result  
);  
```

For tunable handlers, the `event_data` pointer points to a `ktune_event_t` structure, as defined. The `handler_data` pointer is a void pointer for use by the module registering the handler; it always contains whatever value was given to `ktune_register_handler`.

The reason codes are as follows:

- **KEN_REGISTER** Tells the handler that it was just registered. Usually this code is ignored, but the handler can use it to initialize static variables if needed.
- **KEN_UNREGISTER** Tells the handler that it was unregistered. Modules that contain handlers must not allow themselves to be unloaded until the handler is called with this reason code.
- **KEN_EVENT** Indicates that a tunable-related event is in progress. See “Dynamic Changes to Tunable Values” (page 188).
- **KEN_BACKOUT** Indicates that the work done by a previous KEN_EVENT invocation must be reversed. This happens when a transactional tunable change fails. See “Dynamic Changes to Tunable Values” (page 188).
The handler function must take no action and return KEN_DONE when it is called with any reason code it does not support.

When called with a reason code of KEN_EVENT or KEN_BACKOUT, the handler must consult the operation code in the ktune_event_t structure to determine what to do.

**NOTE:** No ktune_event_t structure is passed to the handler when called with a reason code of KEN_REGISTER or KEN_UNREGISTER. All handlers are called with these reason codes when the handler is registered and unregistered, respectively. Therefore, handlers must not attempt to dereference the event_data pointer until they have checked for a KEN_EVENT or KEN_BACKOUT reason code.

The handler must be sensitive to the state of the system when it is called. Handlers are never called from an interrupt context, but they are often called during the boot process. Therefore, handlers cannot assume they are in process context, or that all kernel services are available.

The ktune_event_t structure fields in this section are as follows. For more information about the structure field, see ktune_event_t(9S).

- `kte_txnid`: Specifies the ID number of the transaction being serviced (if any). This is used as a parameter to the various access routines that handlers can call.
- `kte_tuneid`: Specifies the ID number of the tunable for which the handler is being called. This is used as a parameter to the various access routines that handlers can call.
- `kte_op`: Specifies the operation code that tells the handler what to do. Possible values include:
  - KTOP_CAPABLE
  - KTOP_VALIDATE
  - KTOP_GETSCALE
  - KTOP_GETDEFAULT
  - KTOP_PREPARE
  - KTOP_COMMIT
  - KTOP_NOTIFY
- `kte_flags`: Specifies operation flags.

The following sections describe the meanings of each of the operation codes and the actions handlers take when receiving them. If a handler receives an operation code that it does not recognize or implement, it must do nothing and return KEN_DONE.

For more information on the ktune_handler handler function and the ktune_event_t structure, see the HP-UX 11i v3 Driver Development Reference.

For an example of a tunable handler function showing all of the operation codes and capabilities of a tunable handler, see “Tunable Handler Function Example” (page 190).

### Handler Capabilities

The tunable infrastructure must know the capabilities of each of the tunable handlers registered with it. Handlers must provide this information upon request. When a handler is called with a KEN_EVENT reason code and a KTOP_CAPABLE operation code, the handler must set its result parameter to the bitwise OR of all of the KTOP_* operation codes it implements, then return KEN_DONE.

The handler must always return the same bitmask of capabilities each time it receives this request.
Validating Tunable Values

Tunable handler functions must validate proposed values for the tunables they control. They must do the following:

- Recognize and accept appropriate values.
- Reject illegal values with clear, specific error messages.
- Accept legal but questionable values, and generate clear, specific warning messages.

Tunable handler functions do not need to validate the constraints on the tunable values expressed in the tunable definition in the module metadata; those are validated by the tunable infrastructure. However, any other constraints on the tunable values must be validated by tunable handler functions.

When a handler is called with a KEN_EVENT reason code and a KTOP_VALIDATE operation code, it is being asked to validate a proposed value for the tunable it controls. The handler must query the proposed (pending) value of the tunable by calling ktune_pending and passing it the tunable ID and transaction ID that were given to the handler in the ktune_event_t structure. The tunable ID can also be retrieved using the ktune_id call. It can then perform any necessary validation checks on the value.

Some validation checks compare the value of the tunable against a transient variable describing current system operation (for example, amount of free memory). Do not perform these checks when the tunable change is being held for next reboot. Validation handlers can look for the KTF_NEXTBOOT flag in the kte_flags field of their event_data parameter to determine whether changes are being held for next reboot.

If the tunable value is not valid, the handler must set its result parameter to an appropriate error code, usuallyEINVAL orENOMEM. Error codes are listed in settune_txn(2).

The handler must also log an appropriate error message using ktune_error. Error messages must be written in plain English with a minimum of jargon and a maximum of useful information. They must indicate what you can change to address the problem. They must be written in complete sentences, not sentence fragments. HP recommends that you use interpolated values to customize the message.

The following are examples of error messages:

GOOD ERROR: The proposed value of “foobrotz” (12) is not valid because the value must be a power of two.

GOOD ERROR: The value of “blarney” must be greater than the value of “truth”. The proposed values of 20 for “blarney” and 80 for “truth” are not valid.

BAD ERROR: maxuprc invalid.
   This message is not a complete English sentence, and contains no information about valid values.

BAD ERROR: too many clists.
   This message is not a complete English sentence.

BAD ERROR: 40 is not a valid value for “razzle” because init_razzle_dazzle returned an error code of 3.
   This message is a complete English sentence, but it contains code references that have no meaning to the administrator. There is no way for an administrator to know how to fix this problem.

Do not hard code the names of any tunables in the error messages. Instead, put a percent sign (%) in the error message, and use the ktune_name function to generate the tunable name.

If the tunable handler finds that the proposed value is legal, it must ensure that the value makes sense and is reasonable. If not, the handler must register a warning message by calling
ktune_warning. The handler does not put an error code in its result parameter in this case. The guidelines for warning messages are the same as for error messages.

When responding to a validation request, handlers must not do anything that assumes the proposed value will be used. Administrators can validate values without applying them. Handlers are never asked to back out a validation.

**Constraint Handlers**

A common validation check enforces a required relationship between two or more tunables. For example, the tunable msgmnb must be less than the product of the tunables msgsz and msgseg. This constraint cannot be expressed in the module metadata (it involves more than two tunables), so it must be validated by tunable handlers.

You can write a validation check in the handler functions for each of these three tunables. The three checks look virtually identical, since they all need to get the pending value for all three tunables, make the comparison, and raise the appropriate error message.

To avoid this duplication, write a single, separate handler containing only that validation check, and register that handler for all three tunables. (A tunable can have any number of handlers registered for it.)

**Computed Default Values**

Some tunables have hard coded default values, which never change regardless of the characteristics of the system in use. However, in many cases it is better to compute the default value at run time, based on available physical memory, system hardware type, or other factors. Doing so helps approach the ideal that default values work for the majority of customers and rarely need to be changed.

To compute the default value for a tunable, the tunable must have a handler function that responds to the KTOP_GETDEFAULT operation code. (This code is used only with a KEN_EVENT reason code.) When the handler function receives this request, it must compute an appropriate default value, then give that value to the tunable infrastructure by calling ktune_savedefault.

**Dynamic Changes to Tunable Values**

Customers can change tunable values without rebooting. All public tunables must support this, and HP recommends support with private tunables as well.

Tunable changes are implemented in a transaction-based way. Several tunables can be changed in one operation, which succeeds only if all of the individual changes succeed. To fit within this scheme, a tunable must have handlers that respond to the following requests:

- A KEN_EVENT reason code and a KTOP_PREPARE operation code.
- A KEN_EVENT reason code and a KTOP_COMMIT operation code.
- A KEN_BACKOUT reason code and a KTOP_PREPARE operation code.

A transaction-based change follows these steps:

1. If any of the tunables in the transaction are set to percentage_tune, their handlers are called with a KTOP_GETSCALE operation to retrieve the scaling factors. The percentage values are then translated into scaled values for use in the transaction.
2. If any of the tunables in the transaction are set to default, their handlers are called with a KTOP_GETDEFAULT operation to retrieve the default values. These values are used in the rest of the transaction.
3. The handlers of all tunables in the transaction are called with a KTOP_VALIDATE operation to ensure that the values in the transaction are all valid. If any are not valid, the transaction stops.
4. The handlers of all tunables in the transaction are called with a `KEN_EVENT` reason code and a `KTOP_PREPARE` operation code. The handlers take necessary actions (for example, preallocation of memory and locking) to ensure that the tunable changes can succeed.

5. If an error occurs in any of the handlers in step 3, the transaction is aborted. Every handler that was successfully called in step 3 is called again with a `KEN_BACKOUT` reason code and a `KTOP_PREPARE` operation code. Those handlers can reverse any steps they took.

6. If all of the handlers succeed in step 3, they are called again with a `KTOP_COMMIT` operation code. This is their signal to apply the tunable change.

7. After all of the handlers complete step 5, they are called again with a `KTOP_NOTIFY` operation code. Modules that need to know when tunable changes are complete can register handlers that understand this operation code.

The simplest case of a tunable change is one that does not require any preallocation or locking and is guaranteed to succeed. In this case, the tunable handler only needs to implement the `KTOP_COMMIT` operation.

---

**TIP:** A prewritten tunable change handler is available for simple tunables that are plain integers. See `ktune_simple_dynamic(9F)` for more information.

If there are error conditions that prevent the tunable change from being applied, the handler must take steps during the `KTOP_PREPARE` phase to ensure that a later `KTOP_COMMIT` can succeed. Then it must also implement a backout path for the `KTOP_PREPARE` phase in case some other tunable change in the transaction fails and the operation never reaches the commit step.

### Avoiding Race Conditions

Supporting transaction-based changes prevents an administrator from being concerned about the order in which tunable changes are applied. Consider the following case:

- The `nkthread` tunable must always be greater than the `nproc` tunable.
- The `nproc` tunable is set to 100 and `nkthread` is set to 200.
- The administrator wants to raise `nproc` to 300 and `nkthread` to 400.

If the administrator makes these two tunable changes separately, the change to `nkthread` must be made first. Changing `nproc` first results in an error because the constraint is violated, and the system does not know that the violation is temporary.

If an administrator specifies that the two changes are to be made as a part of a single transaction, this error cannot occur. The tunable handlers must make sure that the changes are made in the proper order.

This kind of transaction processing is essential when switching between kernel configurations. In these cases, transaction-based changes usually affect every tunable in the system. The tunable handlers must apply changes in the proper order.

In the previous example, no technical problem prevents having `nkthread` from being less than `nproc` momentarily, as long as the situation does not last long. In cases like this, ignore the ordering issues, knowing that the changes will be applied in quick succession and the exact order does not matter.

However, in some cases the exact order does matter, and a race condition between changes to two tunables must be avoided. In these cases, write a single handler function that handles the changes to both tunables in the proper order.

### Automatic or Self-Tuning Tunables

An automatic (self-tuning) tunable can be changed without a reboot and has a computed default value. When the administrator sets an automatic tunable to `default`, the system uses the best value for that tunable, even if that value changes over time.
The module that owns the automatic tunable must watch for changing system conditions that cause a change in the computed default value, and automatically apply those changes. The module does not need to notify the tunable infrastructure each time this happens.

A module can find out whether the administrator has asked for automatic tuning in one of the following ways:

- Look for the KTF_DEFAULT flag in the kte_flags field of the ktune_event_t structure passed to the tunable handler during the KTOP_COMMIT or KTOP_NOTIFY operations. If the flag is set, the administrator has turned on automatic tuning. If the flag is clear, the administrator has disabled automatic tuning. The module can track this state in its own data structures.
- If the module does not want to track the state by itself, the tunable infrastructure can do it. The module can find out whether the tunable is in its default state at any time by calling ktune_isdefault or ktune_isauto.

In either case, if the tunable is in its default state, the module must continuously update its own copy of the tunable value as needed in response to system conditions. The tunable must respond to any KTOP_GETDEFAULT requests by returning the value currently in use.

If the administrator sets the tunable to a nondefault value, that disables automatic tuning. In these cases ktune_isdefault returns zero (false), and the module must not change its copy of the tunable value except on request.

**Notification of Tunable Changes**

Sometimes a module needs to know whenever the value of a tunable is changed. The tunable in question might belong to the same module or to another module. In either case, the interested module can register a handler for that tunable that responds to the KTOP_NOTIFY operation code. This operation code is generated at the end of any successful transaction-based tunable change.

**NOTE:** This operation code is not called when a module makes a change to an automatic tunable. It is called only when automatic tuning is turned on or off (when the administrator requests a change).

**Percentage Tunable Parameters**

Tunables can accept values as percentages. For example, consider a tunable named useless_threads. Set it to an absolute value (for example, 12 useless threads) or to a percentage (for example, 5 percent). In the latter case, if the number of threads on the system changes, the number of useless threads changes automatically to compensate.

Tunables that support percentage values must use the percentage_tune flag in their module metadata file and their handlers must respond to the KTOP_GETSCALE request. The tunable infrastructure issues the KTOP_GETSCALE request when it needs to translate a percentage into an absolute value. When the handler receives this request, it must call ktune_savescale with the absolute value that corresponds to 100 percent for this tunable. Subsequent KTOP_GETSCALE requests do not have to return the same value; the scaling factor can change over time.

**Tunable Handler Function Example**

This example shows the tunable handler for a tunable called num_widgets. It combines all the features that tunable handlers can provide; most tunable handlers are simpler and provide only some of these features.

There is one (fictional) widget per mounted file system. The num_widgets tunable controls the maximum number of these widgets. This is an automatic tunable; when it is automatically tuned its value is kept equal to the number of file systems mounted. Its value can be overridden but
cannot be set to less than the number of file systems mounted at the time of the change. Its
value controls the size of a kernel data structure, which must be reallocated whenever the value
changes.

```c
int num_widgets_handler(ken_id_t eventid,
    int reason,
    ken_instance_t instance,
    void *handler_data,
    ktune_event_t *event_data,
    int *result)
{
    static struct widget *new_widgets;
    static int            new_num_widgets;
    ktune_id_t            tuneid;
    ktune_txnid_t         txnid;
    uint64_t              value;
    int                   ret;

    // Why were we called?
    switch (reason) {
        // Keep track of registrations and unregistrations of this
        // handler, so we know when it is safe to unload the module.
        // (The module unload function will check this global variable,
        // and will refuse to unload the module if it is non-zero.)
        case KEN_REGISTER:
            num_widgets_handler_registered++;
            break;
        case KEN_UNREGISTER:
            num_widgets_handler_registered--;
            break;
        // Handle requests from the tunable infrastructure.
        case KEN_EVENT:
            tuneid = event_data->kte_tuneid;
            txnid = event_data->kte_txnid;
            // What is the request?
            switch (event_data->kte_op) {
                // Report the handler's capabilities on request.
                case KTOP_CAPABLE:
                    *result = KTOP_CAPABLE | KTOP_GETDEFAULT | KTOP_VALIDATE |
                              KTOP_PREPARE | KTOP_COMMIT;
                    break;
                // Compute the default value for the tunable.
                case KTOP_GETDEFAULT:
                    ktune_save_default(txnid, tuneid,
                                      num_fs_mounts);
                    break;
                // Validate that proposed values are legal and appropriate.
                case KTOP_VALIDATE:
                    // Get the proposed value.
                    if (ret = ktune_pending(txnid, tuneid, &value)) {
                        *result = ret;
                        break;
                    }
                    // Check the proposed value against the number of
                    // file systems that are currently mounted. If the
                    // proposed value is lower, there is a potential problem.
                    if (value < num_fs_mounts) {
                        // The proposed value is too low.
                        // If the proposed change is being held for next boot,
                        // accept it but print a warning. If the change
                        // is to the currently running system, it is illegal.
                        if (event_data->kte_flags & KTF_NEXTBOOT) {
                            ktune_warning(txnid, "The value of %s (%ld) is "
                                           "not adequate to mount all %d "
                                           "currently mounted file systems "
                                           "after next boot.",
                        }
                    }
    
```
ktune_name(tuneid), value,
  num_fs_mounts);
} else {
  ktune_error(txnid, "The value of %s (%ld) may not "
  "be less than the number of mounted "
  "file systems (currently %d).",
  ktune_name(tuneid), value,
  num_fs_mounts);

  *result = EINVAL;
  break;
}
}

// Make sure that the value can be applied, and establish
// controls to keep it that way until the change is committed.
// This is phase one of a two-phase commit process.
case KTOP_PREPARE:
  // Get the desired value.
  if (ret = ktune_pending(txnid, tuneid, &value)) {
    *result = ret;
    break;
  }

  // Lock the number of mounts, to be sure that no mount
  // operations take place.
  w_spinlock(fs_mount_lock);
  // Check the desired value against the number of file
  // systems that are currently mounted. If the desired
  // value is lower, the change is illegal. (This repeats
  // the change done by KTOP_VALIDATE since file systems
  // might have been mounted since KTOP_VALIDATE was called.)
  if (value < num_fs_mounts) {
    ktune_error(txnid, "The value of %s (%ld) may not "
    "be less than the number of mounted "
    "file systems (currently %d).",
    ktune_name(tuneid), value,
    num_fs_mounts);

    *result = EINVAL;
    w_spinunlock(fs_mount_lock);
    break;
  }

  // Establish a new maximum number of mounts, as the lower
  // of the current num_widgets value and the desired one.
  // This ensures that the tunable change can be committed
  // later.
  max_fs_mounts = min(num_widgets, value);
  // Now mounts can continue (up to the new
  // maximum) while waiting for the change to be committed.
  w_spinunlock(fs_mount_lock);
  // Allocate the new widgets array with the new size.
  // Do this now to prevent an out of
  // memory error during the commit.
  new_num_widgets = value;
  new_widgets = kmem_arena_alloc(widget_arena,
    new_num_widgets*sizeof(struct widget), M_WAITOK);
  bzero(new_widgets, new_num_widgets*sizeof(struct widget));

  // We are done preparing. We have not actually changed
  // anything, but we have ensured that we can do so when the
  // tunable change is committed.
  break;

  // Now commit the tunable change.
  case KTOP_COMMIT:
    // Grab the lock on the widget list, move the widget
    // data to the new array with the new size, free the old
    // array, and release the lock.
  w_spinlock(widget_lock);
bcopy(new_widgets, widgets,
    max_fs_mounts*sizeof(struct widget));
num_widgets = new_num_widgets;
kmem_arena_free(widgets);
widgets = new_widgets;
w_spinunlock(widget_lock);
    // Lock out mount operations. Note whether the tunable
    // is being autotuned, so the mount function can do the
    // right thing. Update the maximum number of mounts.
    // Release the lock.
w_spinlock(fs_mount_lock);
num_widgets_autotune = event_data->kte_flags & KTF_DEFAULT;
max_fs_mounts = num_widgets;
w_spinunlock(fs_mount_lock);
break;
}
break;
    // Reverse a step we took previously.
case KEN_BACKOUT:
    switch (event_data->kte_op) {
    // Reverse the KTOP_PREPARE step from above.
    // Do not make the tunable change after all.
    case KTOP_PREPARE:
        kmem_arena_free(new_widgets);
        max_fs_mounts = num_widgets;
brea
    }
break;
}
return KEN_DONE;

The handler is closely tied to the file system mount and unmount functions in the example. The mount function must contain code as follows:

// Ensure we have enough widgets for the mount.
r_spinlock(fs_mount_lock);
    if (num_fs_mounts >= max_fs_mounts) {
        // We do not. If the num_widgets tunable is being autotuned,
        // allocate more widgets. Otherwise, fail the mount.
        if (num_widgets_autotune) {
            resize_widgets(num_fs_mounts+1);
        } else {
            r_spinunlock(fs_mount_lock);
            return ENOSPC;
        }
    }
r_spinunlock(fs_mount_lock);

The unmount function must reduce the widgets array if the tunable is being autotuned:

r_spinlock(fs_mount_lock);
    if (num_widgets_autotune) {
        resize_widgets(num_fs_mounts-1);
    }
r_spinunlock(fs_mount_lock);

The resize_widgets function used here is a combination of the KTOP_PREPARE and
KTOP_COMMIT operations in the handler.

Step 4: Writing Tunable Initialization Functions

Each module that defines tunables must provide a function to initialize its tunables. It can supply
one such function for each tunable, or one function that initializes all of the module’s tunables,
or a combination of both. Most modules have only one tunable initialization function. Initialization
functions are registered in the module metadata using lines like the following:
The initialization routines must do two things. First, they must register the handlers for the module’s tunables. Tunable handler functions are discussed in “Step 3: Writing Tunable Handler Functions” (page 185). To register a tunable handler, the initialization routine must call the following:

```c
int ktune_register_handler(
    int version,
    const char *tunable,
    int ken_flags,
    int ken_order,
    const char *description,
    ken_handler_t handler,
    void *handler_data
);
```

Second, after all the handler functions are registered, the initialization routine must retrieve the administrator's chosen value of the tunable from the tunable infrastructure and use it to initialize or allocate any variables that depend on the tunable. This step also notifies the tunable infrastructure that the tunable is in use, and provides the failsafe value to use if saved tunable data is not available. To retrieve the value of a tunable from the tunable infrastructure, the initialization routine must call the following:

```c
uint64_t ktune_get(
    const char *tunable,
    uint64_t failsafe
);
```

The `ktune_get` routine cannot be called from an interrupt context. While `ktune_get` runs, the handler functions for the tunable are called to validate the configured value. If the tunable is supposed to be in its default state, or the configured value is invalid, the handlers are also called to compute a default value for the tunable. For a discussion on how the handlers must handle these requests, see “Step 3: Writing Tunable Handler Functions” (page 185).

Calls to `ktune_get` must specify a failsafe value for the tunable. This is not the same as the default value. The failsafe value is used if saved tunable values are not available, or if the administrator specifies a boot-time flag requesting failsafe mode. It is intended for temporary use when performing disaster recovery. Choose failsafe values that work acceptably on all types of systems and under the widest conditions possible.

Following is a sample initialization function for `nkthread`:

```c
ret = ktune_register_handler(KTUNE_VERSION, "nkthread",
    0, KEN_UNORDERED,
    "dynamic nkthread",
    nkthread_handler, NULL);
if (ret != 0) ...
nkthread = ktune_get("nkthread", NKTHREAD_FAILSAFE);
```

### Step 5: Writing Tunable Teardown Code

The unload function for a dynamically loadable module must reverse the steps taken by the tunable initialization functions for that module. That is, it must call the following to unregister any tunable handlers that were registered by the initialization functions:

```c
int ktune_unregister(
    int version,
    const char *tunable,
    int ken_flags,
    int ken_order,
    ken_handler_t handler,
```
void ***handler_data
);

The teardown code for nkthread looks something like this:
ret = ktune_unregister_handler(KTUNE_VERSION, "nkthread",
0, KEN_UNORDERED,
nkthread_handler, NULL);
if (ret != 0) ... 

Step 6: Building and Testing the Kernel
Test that the new kernel builds and boots on different systems. Ensure that \texttt{kctune} is can see and modify the new tunable. Run all the functional tests for the module and make sure they pass.

Step 7: Creating Tunable Documentation
Every public tunable must have a manpage, which is delivered with all other manpages to the customer system. These manpages have the same names as the tunables they describe. Closely related tunables can share a single manpage (indexed to both names), but HP recommends that you create a separate manpage for each tunable.

Tunable manpages have a common form, with a set of questions to be answered for each tunable. HP recommends that you start with the manpage for an existing tunable and modify it.

Include these manpages in the same Software Distributor depot as the kernel module that defines the tunables. They must be delivered to $/usr/share/man/man5$.

Additional Tasks
This section includes step-by-step instructions to change a tunable parameter's default attributes and to change the tunable value dynamically.

Changing Tunable Default Values or Attributes
To change tunable default values or attributes, follow these steps:

1. For any changes to tunable attributes, update the tunable manpage.
2. To change a hard coded default value for a tunable, edit the tunable definition in the module's metadata file and change the default value that appears in it. To change an algorithmic default value, edit or create a tunable handler function for the tunable that responds to a KTOP_GETDEFAULT request. See “Computed Default Values” (page 188).
3. To change the hard coded constraints on a tunable's value, edit the tunable definition in the module's metadata file and change the constraints that appear in it. To add, remove, or change any other constraints on the tunable's allowed values, edit or create a tunable handler function for the tunable which implements a KTOP_VALIDATE request. See “Validating Tunable Values” (page 187).
4. To hide a tunable from end users, edit the tunable definition in the module's metadata file and add the \texttt{private} flag.
5. To change the failsafe value for a tunable, edit the code of the initialization function for that tunable. The failsafe value is one of the parameters to the \texttt{ktune\_get\_call} that initializes the tunable.

How to Make a Tunable Dynamic or Self-Tuning
To make a tunable parameter dynamic or self-tuning, follow these steps:
1. Edit or create a handler function for the tunable. Make the handler function respond to the \texttt{KTOP\_COMMIT} and (if necessary) \texttt{KTOP\_PREPARE} requests. See “Dynamic Changes to Tunable Values” (page 188).

\textbf{TIP:} Before writing a new handler function to support dynamic changes, check to see whether the supplied \texttt{ktune\_simple\_dynamic} function can be used instead. For more information, see its manpage in the \textit{HP-UX 11i v3 Driver Development Reference}.

2. If a new handler function is created, add code that registers the new handler function to the initialization routine for the tunable, and add code to the module's unload function to unregister the new handler. If the supplied \texttt{ktune\_simple\_dynamic} handler is used, add code to register and unregister it.

3. When an administrator turns on automatic tuning (for tunables that support it), the handler is called with a \texttt{KTOP\_COMMIT} request with the \texttt{KTF\_DEFAULT} flag set. The subsystem code can then use whatever value for the tunable it wants, and change that value at will, until such time as it receives another \texttt{KTOP\_COMMIT} request with that flag clear. See “Automatic or Self-Tuning Tunables” (page 189). While automatic tuning is turned on, the tunable handler must respond to a \texttt{KTOP\_GETDEFAULT} request by returning the value currently in use.

4. Update the manpage for the tunable.
9 Writing PCI Drivers

This chapter presents routines and conceptual material specifically for drivers of PCI devices. In HP-UX 11i, most of the PCI I/O memory accessor services have been replaced by new WSIO services. These services map and access I/O port space, I/O registers, and PCI configuration space. For additional information on these WSIO services, see Chapter 4 (page 77).

In this version of HP-UX 11i, the PCI services are supported, but will be obsoleted. New driver writers must use the new WSIO services and existing drivers must be ported to use the new WSIO services.

The HP-UX PCI Services routines are described in the HP-UX 11i v3 Driver Development Reference.

The examples in this chapter follow the routine naming conventions described in “Step 1: Choosing a Driver Name” (page 77).

PCI Overview

This section gives a brief overview of PCI. Before writing a driver for a PCI card, read the PCI Local Bus Specification, Revision 2.1.

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 located in PCI memory space or (less often) in PCI I/O space.

PCI Configuration Space

PCI configuration space holds specific registers for initialization and configuration of PCI devices. Some or all of this register space is the same for all PCI devices, enabling generic initialization software to recognize and configure all PCI compliant devices.

This space is accessed primarily at startup time, when initialization occurs, but can also be accessed 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” (page 206). 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 manufacturer for any card-specific registers. However, card-specific registers usually reside in PCI memory space or PCI I/O space.

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**

The input parameters for these macros are virtual addresses, which are mapped to PCI memory addresses. The macros have different effects depending on whether or not **PCI_LITTLE_ENDIAN_ONLY** is defined by the driver prior to including `<sys/pci.h>`. For more information, see “PCI_LITTLE_ENDIAN_ONLY Flag” (page 203).

On workstations, mapped PCI memory space can also be accessed directly. In this case, the driver must handle endian issues.

**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 addresses the ordering of transactions to and from PCI space. Transactions include the following:

- 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 must 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.

**Processor-Mastered PCI Transaction**

This section describes 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. Ordering is not a problem with reads; 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 are nonblocking (posted) transactions. 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 completes 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 simple. If a processor writes to registers A, B, and C in that order, the writes complete so that they are only observable in the same order. For example, you can never observe that B had been written but A had not yet been written. 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. 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 may not happen immediately. Writes are posted and complete eventually.

All posted writes must be flushed and completed before any read can complete. So, to assume a write's effects have occurred, a read must be performed to flush the writes posted in the queue. When coding register writes, most of the time, it is acceptable to not know when a register write completes. But in some cases take care.

For example, consider a driver's Interrupt Service Routine (ISR) managing the Interrupt Request Register (IRR) on a card. Clearing a bit in the IRR indicates that the interrupt has been serviced. This is done by posting a write to the IRR. If the driver posts this write and exits its ISR, it can be interrupted again immediately because the write had not reached the bit in the IRR to tell it to stop trying to interrupt. One solution to this potential problem is to read back the value in the IRR before exiting from the ISR. Most drivers do this so they can handle multiple interrupts in the same ISR invocation.

PCI Card-Mastered Transactions

Use the terms **DMA read** for a PCI card-mastered read from host memory and **DMA write** for a PCI card-mastered write to host memory. In current hardware implementations, transaction ordering of DMA reads and DMA writes are preserved only when the target memory locations are contained in the same processor cacheline. In other cases, DMA reads can pass DMA writes. Drivers must take this behavior into account.

If the driver needs the exact PCI producer consumer behavior as seen from the PCI card, ensure that the elements residing in host memory that require strict ordering are physically on the same cacheline. Current hardware implementations have cachelines that are multiples of 32 bytes in length. Be sure the flag or status elements are limited to 32 bytes aligned on **MAX_CACHELINE_SIZE** boundaries (defined in `<sys/dma.h>`).

Interleaved PCI Transactions

This section details transaction ordering of interleaved processor-mastered and PCI card-mastered reads and writes to host memory. If you expect PCI or PA ordering rules to apply in this situation, ensure that the producer consumer elements reside on the same cacheline. The following scenario does not meet the producer consumer transaction ordering requirements:

- Cacheline X holds the card's status — initially "working."
- Cacheline 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 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.

Ordering is not enforced between the two cachelines, and DMA reads can pass DMA writes. Thus, both the processor and the card reads can return the original value. This results in the card going to sleep and the processor not waking it up.

If the status and commands cannot be placed on the same cacheline, use other means to ensure correct behavior. For example, you can set a timeout to ensure the deadlock did not occur. In most cases, commands are written to the card register. That is, the command is not in host memory and the above scenario does not apply.

The following scenario does not meet the producer consumer transaction ordering requirements:
1. 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.
2. Processor begins updating the task list in main memory.
3. 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.
4. Posted processor write to the card arrives at the card, telling it to stop processing the list, which is already done.

DMA reads by a PCI master can 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. Avoid this situation by doing a processor read of PCI space immediately following the processor write, as follows:

1. Processor writes a command to the PCI card to stop processing the task list.
2. Processor does a dummy read of the PCI card to make sure the posted write to PCI space has completed. A read of card status can be required here to ensure the DMA engine has stopped fetching tasks.
3. Processor updates the task list in main memory.
4. Processor writes a command to the PCI card to resume task processing.

This behavior can occur on all supported PCI-based systems as of this writing. Drivers written for workstations must always ensure that, where necessary, posted writes are followed by dummy reads to ensure ordering. This behavior is not enabled automatically in servers due to chipset implementation.

PCI Endian Issues

PA-RISC is a big-endian architecture. For a multibyte quantity, the Most Significant Byte (MSB) has the lowest address and the Least Significant Byte (LSB) has the highest. Intel's Itanium processors are little-endian. Because PCI was derived from the PC world, it is little-endian.

When multibyte words are transferred between the PCI bus and the PA-RISC system bus, the bytes of the word are reversed or swapped by the hardware. This ensures that the receiving system can properly interpret and store the data from MSB to LSB. This does not happen when the data is transferred byte by byte.

Byte Swapping

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 all the bytes end up in the correct order on both sides of the interface. This means 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 such as disks connected to the built-in SCSI on the internal system bus can instead be connected to a SCSI card on the PCI bus.

Preswapping

Because of byte swapping, the interpretation of multibyte integers is problematic. Assume the transfer occurs from the big-endian system to the little-endian system, and swapping is being performed. If the byte array in question is a four-byte integer, it is stored in big-endian format, MSB at the lowest address, on the little-endian side. However, if a device on the little-endian side of the interface decides to interpret these bytes as a four-byte integer, the value it sees has 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 to be interpreted as an integer must 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. However, if the integer is stored in memory, it ends up reversed.
Several macros are provided in the file `<sys/pci.h>` to assist in swapping data.

**PCI Device Setup**

This section describes information that you need before attempting to set up a PCI device.

**Mapping Base Address Registers**

When an HP-UX system boots, **Processor Dependent Code** (PDC), **I/O Dependent Code** (IODC), and HP-UX system code map a PCI card’s memory space base address registers (BARs) into PCI memory space and I/O space BARs 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 BARs located in the PCI configuration space. If the system finds a suitable mapping, it writes the base of the range back into the corresponding BAR. This address is a PCI memory address if the BAR identifies itself as a memory BAR, and a PCI I/O address if the BAR identifies itself as an I/O BAR.

A driver’s `driver_attach` routine can then access the values loaded into the BARs in configuration space. A driver must not overwrite these addresses with different values except as follows. As long as response to memory or I/O accesses from the command register is not enabled, a driver can read the BAR contents, write all ones to the BAR 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 use these base addresses, another mapping must take place. The problem is the addresses placed in the BARs by the system do not contain virtual addresses usable by the computer. Instead, they contain PCI addresses used to communicate on the bus. If a BAR is a memory BAR, it contains a PCI memory address. If it is an I/O BAR, it contains a PCI I/O address. For more information, see “PCI Register Spaces” (page 197).

In either case, to use the PCI address in the BAR, a mapping to a PA resource must take place for the system to access the registers pointed to by the base.

Do not arbitrarily mask bit zero of a BAR. This bit indicates whether or not this particular register set responds to PCI memory cycles or PCI I/O cycles. During early PDC and IODC configuration, the defined BARs are written as prescribed by the PCI specification to determine size, alignment, and access type. If bit zero is set, PDC and IODC probing has determined that this particular register set only responds to I/O cycles. In this case, access the register set using the PCI services provided.

**Using PCI Memory Base Registers**

To use a PCI memory BAR, the driver must map the range of PCI memory space to a range of processor memory space by calling `map_mem_to_host`. The `map_mem_to_host` call takes the PCI memory address (obtained directly from the BAR) and a size as inputs, and returns a virtual address used to access that PCI address range. The accessor macros, `READ_REG_UINTn_ISC` and `WRITE_REG_UINTn_ISC`, take processor virtual memory addresses as arguments, not PCI memory addresses.

**NOTE:** After reading a PCI memory BAR’s value from PCI configuration space, mask off the bottom four bits before calling PCI services such as `map_mem_to_host`, because they have special values defined by the *PCI Local Bus Specification, Revision 2.1*.

After this virtual mapping is done, the machine uses processor memory-mapped I/O to access the range. Accesses to that range of processor memory space are transmitted into the PCI memory space. Loads and stores to these processor memory addresses result in loads and stores to the registers to be accessed.
For WSIO drivers, the *if_reg_ptr* member of the Interface Select Code (*isc*) structure is a virtual address corresponding to a BAR that already has virtual mapping done to make it usable by the driver and system. If *if_reg_ptr* is NULL, the driver must map the range itself. See “Mapping the Memory Base Register” (page 202). This is done in a PCI device’s *driver_attach* routine.

### Using PCI I/O Base Registers

To use a PCI I/O BAR, 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_PORT_HNDL* PCI structure. The PCI I/O space accessor functions *pci_read_port_uintN_isc* and *pci_write_port_uintN_isc* take port handles as arguments.

To map from a PCI I/O address to a port handle, the driver must read the I/O BARs 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*. The driver then uses this port handle (with an offset) to access registers in PCI I/O space.

---

**NOTE:** When reading a PCI I/O BAR’s value out of PCI configuration space, mask off the bottom two bits before calling services such as *pci_get_port_handle* because they have special values defined by the *PCI Local Bus Specification, Revision 2.1*.

### Automatic IRQ Determination

PCI drivers calling *wsio_intr_get_irq_line* must always pass -1 as the *irq_line_num* argument. This argument value causes the functions to read the needed *Interrupt Request* (IRQ) information from the PCI device or function configuration space Interrupt Pin or Interrupt Line registers and use it to set up the ISR properly. The IRQ information can be read from the Interrupt Line register.

### Mapping the Memory Base Register

Many cards have only a single range of registers (only a single memory BAR). For cards like these, use the *if_reg_ptr* field in the ISC structure.

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 BAR found is mapped, starting at 0x10 and searching up to 0x24, inclusive. These are the six defined BAR locations in PCI configuration space. A virtual address for accessing this register is stored in *if_reg_ptr*.
- If that base register size (the size of the register range) exceeds 8 KB, it is not mapped and *if_reg_ptr* is set to NULL. In this case, the driver must map the BARs 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 cannot be mapped and you must not attempt to access it.

These limitations are necessary to define which of many possible BARs are mapped, and to prevent unnecessary use of Translation Lookaside Buffers (TLB). If PCI Services do not map in any memory BAR, or if there are more registers than the first one found, the driver can read the BARs explicitly from the PCI device or function’s configuration space and get a PA virtual mapping with the *map_mem_to_host* kernel routine. For an example, see “Sample driver_install Routine” (page 208).

The limitations also prevent wasting of kernel resources on BARs that may not be mapped in the normal way; for example, a graphics card frame buffer is an enormous range that must be treated differently from a regular register range. PCI Services has arbitrarily decreed that anything bigger than 8 KB must 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 cannot know about system parameters. The basic guideline is do not alter unless necessary. Following are some examples of configuration registers that must not be altered:

Command Register

The command register must be written by drivers to enable bus mastering, memory space access, and I/O space access, and so on. 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 might have been previously set must not be overwritten. When a driver sets 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.

Latency Timer Register

This is set by the system. It must not be tampered with by individual drivers, as incorrect settings can degrade overall system performance.

Cache Line Size Register

This register is set by the system to match the machine's cacheline. Drivers do not know the cacheline size for the particular machine they are currently running on, so they must not change this register's contents.

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 must not be overwritten by drivers, with one exception. In some cases, a driver might need to determine the size and alignment of the range a BAR is mapped to. To get this information, the driver writes all ones to the register, reads the result back, and decodes it for the needed values, as described in the *PCI Local Bus Specification, Revision 2.1*. This is permitted only if the original value is read and stored first, then restored to the register after the size is determined. This must be done before enabling memory or I/O transactions to the card through the command register.

Interrupt Line Register

System-specific interrupt routing information is stored in this register. Writing a new value to it can cause the card to stop working.

PCI Device Operation

Most PCI drivers are written for cards whose primary method of accessing registers is through PCI memory space.

PCI_LITTLE_ENDIAN_ONLY Flag

HP recommends that drivers define the `PCI_LITTLE_ENDIAN_ONLY` flag before they include `<sys/pci.h>`. This provides better performance from I/O accesses.

PCI drivers written for workstations only (currently all third party drivers) can use direct C code constructs to access registers in PCI memory space. For example:
myClearRegs(regsToInit, size)
    u_int *regsToInit;
    int size;
    {
        int i;
        for (i=0; i<size; i++)
            *regsToInit++ = 0;
    }

These drivers can also use the READ_REG_UINTn_ISC and WRITE_REG_UINTn_ISC macros with the PCI_LITTLE_ENDIAN_ONLY flag defined in the <sys/pci.h> header. The choice of whether to directly access a register or to use one of the macros becomes whether or not 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 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 <sys/pci.h> header.

If the driver does not explicitly define PCI_LITTLE_ENDIAN_ONLY before including <sys/pci.h>, the macros expand into function calls that are guaranteed to byte swap correctly and perform the memory access. This is extra safe mode; it will always work on all bus adapters. The function calls guarantee PCI adapter independence. However, extra function call overhead is added to the register access, reducing its performance.

If the driver explicitly defines PCI_LITTLE_ENDIAN_ONLY, the performance loss due to the function call is removed. In this case, the macros are expanded by the preprocessor into a series of inline instructions that byte swap and perform the access without a function call. It assumes 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 exception of a few special PA internal system bus-based server PCI card projects. All regular drivers (that is, those that are not explicitly written to drive a specially equipped PA internal system bus based card) benefit from defining the PCI_LITTLE_ENDIAN_ONLY flag and must do so before including <sys/pci.h>.

The following pseudocode (resembling and summarizing the actual code in <sys/pci.h>) helps explain the flag's relation to the macros, and how and why to use it:

```c
#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

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 card register buffers.
PCI has no special routines to perform DMA. It uses the standard WSIO Services calls for bus-independent DMA, including the following:

- `wsio_init_map_context`
- `wsio_map_dma_buffer`, `wsio_fastmap_dma_buffer`, and `wsio_unmap_dma_buffer`
- `iovec` structure

In the *HP-UX 11i v3 Driver Development Reference*, see `wsio_init_map_context(9F)`, `iovec(9S)`, `wsio_fastmap_dma_buffer(9F)`, `wsio_map_dma_buffer(9F)`, and `wsio_unmap_dma_buffer(9F)`.

Certain combinations of WSIO mapping service calls can interact with PCI masters to create an inconsistent view of memory. See “PCI Masters and Coherence” (page 205).

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` to enable the device to master the bus. The use of this bit is illustrated in “Sample driver_attach Routine” (page 209).

**PCI Masters and Coherence**

Prefetching host memory by the hardware chipset can result in a PCI master reading stale data, even though the proper `dma_sync` calls were made. This does not occur if the mapping is done with `wsio_dma_map_buffer` with flags `IO_NO_SEQ` and `IO_SAFE` set. See `pci_errata(9F)` in the *HP-UX 11i v3 Driver Development Reference*.

**PCI Services Summary**

PCI Services are accessed through special PCI functions that enable device and interface drivers to be much smaller and more supportable.

These functions are summarized here and described in detail in the *HP-UX 11i v3 Driver Development Reference*.

- `pci_desc_bus_transactions_isc`: Enables a driver to describe the typical bus performance transaction size.
- `pci_get_port_hndl_isc`: Gets a system-defined handle for manipulating the range of PCI I/O space ports.
- `pci_read_cfg_uintN_isc`: Reads an 8-, 16-, or 32-bit unsigned integer from a PCI configuration register.
- `pci_read_port_uintN_isc`: Reads 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`: Deletes a handle returned by `pci_get_port_hndl_isc`.
- `pci_write_cfg_uintN_isc`: Writes an 8-, 16-, or 32-bit unsigned integer into a PCI configuration register.
- `pci_write_port_uintN_isc`: Writes little-endian data to a PCI I/O port previously identified by a call to `pci_get_port_hndl_isc`.
- `CONNECT_INIT_ROUTINE`: Associates a `driver_if_init` routine with the driver.
- `PCI_ATTACH_DEV_INIT_ERROR`: Notifies WSIO Services that an error occurred during a device’s initialization.
- `READ_REG_UINTn_ISC`: Reads and byte swaps 8-, 16-, or 32-bit data from a little-endian bus.
Byte swaps and writes 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.

Multiprocessor Safety

Code all PCI drivers to be multiprocessor (MP) safe. Specifically, they must not rely on processor priority levels to guarantee exclusive access to critical sections. Instead, they must protect their own critical sections using spinlocks, semaphores, and other methods of MP protection. See Chapter 3 (page 69).

Constants and Data Structures

The constant definitions and data structures are defined in the <sys/pci.h> PCI header file.

User-Visible PCI Specific Data Structure

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

Writing a 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 PCI-specific part of this example is the driver_attach
routine. The other parts are typical of all WSIO drivers. They are included here for context and completeness. For complete information on the structures and functions needed to write a WSIO driver, see Chapter 9 (page 197).

As an example, write a driver for a hypothetical PCI device, the ZZZ8109C PCI Blender card. The blender is a character device, so the driver must 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 is a block device is a SCSI adapter or disk or tape drive controller.

The 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 this is both an interface and a device driver, specify T_INTERFACE in the `wsio_drv_info_t` structure, since both cannot be specified.

Following the routine naming conventions described in “Step 1: Choosing a Driver Name” (page 77), name the driver ZZZ and place it in the (arbitrary) class blender.

Sample WSIO Setup and Structures

Include the necessary header files. For each kernel call and data structure the driver uses to discover which headers the driver requires, see the reference pages in the HP-UX 11i v3 Driver Development Reference. WSIO drivers require the `<wsio/wsio.h>` header file. PCI drivers also require the `<sys/pci.h>` header file.

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

Declare the driver’s routines that can be called by the kernel, used in the `drv_ops_t` structure.

```c
int ZZZ_open();
int ZZZ_close();
int ZZZ_read();
int ZZZ_write();
int ZZZ_ioctl();
```

Declare a `ZZZ_saved_attach` function pointer to store the old head of the PCI attach chain. The `ZZZ_attach` routine is added to the chain in the `ZZZ_install` routine.

```c
static int (*ZZZ_saved_attach)();
```

Define values for vendor ID (`ZZZ_VEN_ID`) and device ID (`ZZZ_DEV_ID`). These are needed for the comparison in `ZZZ_attach`.

```c
/* These must be initialized */
int ZZZ_VEN_ID = value;
int ZZZ_DEV_ID = value;
```

The `drv_ops_t` structure specifies the external driver routines to the kernel. The flags specify that the driver must be called on all device closes, and that it is MP safe. For more information, see “Step 3: Defining Installation Structures” (page 78).

```c
static drv_ops_t ZZZ_ops =
{
    ZZZ_open,                  /* open */
    ZZZ_close,                 /* close */
    NULL,                      /* strategy */
    NULL,                      /* dump */
    NULL,                      /* psize */
    NULL,                      /* reserved */
    ZZZ_read,                  /* read */
    ZZZ_write,                 /* write */
    ZZZ_ioctl,                 /* ioctl */
    NULL,                      /* select */
    NULL,                      /* option1 */
    NULL,                      /* pfilter */
    NULL,                      /* reserved */
};
```
The `drv_info_t` structure specifies the driver's name and class. The flags specify the driver is character type and MP safe. They also indicate that the driver configuration, including major number, must be saved and retained across reboots. For more information, see “Step 3: Defining Installation Structures” (page 78).

```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 */
};
```

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 it conforms to the I/O specifications. For more information, see “Step 3: Defining Installation Structures” (page 78).

```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 call to `wsio_install_driver`. For more information, see “Step 3: Defining Installation Structures” (page 78).

```c
static wsio_drv_info_t ZZZ_wsio_info = {
    &ZZZ_info,
    &ZZZ_ops,
    &ZZZ_data
};
```

**Sample WSIO Routines**

Sample `driver_install` and `driver_attach` routines are explained in this section.

**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 PCI attach chain. If the device has a `driver_dev_init` function, the `driver_install` routine links it into the `dev_init` chain.

```c
NOTE: The name of this routine must begin with the name of the driver, for example, ZZZ, and end with _install. For example, ZZZ_install.
```

```c
void
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 into the pci_attach chain.
   */
  ZZZ_saved_attach = pci_attach;
  pci_attach = ZZZ_pci_attach;
}

/*
 * Exit.
 */
return;

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 the
driver might want to claim. This sample driver put its driver_attach function on the
pci_attach chain, so the system calls it every time a new PCI device is discovered. The
driver_attach routine first checks if this is the type of hardware it supports, then claims it
and performs any card-required initialization.

- The driver must ensure that the contents of a memory or I/O BAR are not zero. All zeros
  indicate that either the specified configuration space register is not implemented by the PCI
device or function, or the system cannot find the resources to map the corresponding space
into the system. If alternate register mappings exist, and those base registers are not zero,
the driver can use those mappings instead.

- PCI Services do 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. The
driver must enable memory access, I/O access, and DMA, as shown in the following sample
driver_attach routine. To support online addition (OLA) of PCI cards, the
driver_attach routine must enable bus mastering.

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 the right card
   */
  if (!(id->vendor_id == ZZZ_VEN_ID &&
       id->device_id == ZZZ_DEV_ID) {
    goto exit0;
  }
  /*
   * If the system uses a standard bus interface chip,
   * check the subsystem vendor ID and subsystem ID
   * to make sure that this driver can claim this device.
   */
  */
* Get the card revision
*/
pci_read_cfg_uint8_isc(isc, PCI_CS_REV_ID, &rev_id);

/*
* Check the isc->if_reg_ptr before using it.
* If it is NULL, read the base register and map it.
* If isc->if_reg_ptr isn't NULL, PCI Services already
* did the mapping.
*/
if (isc->if_reg_ptr == NULL) {
    /*
    * Map the driver's own base address and
    * save the value in if_reg_ptr.
    * First, get the physical base memory address.
    * For ZZZ, memory is at reg 0x10.
    */
    pci_read_cfg_uint32_isc(isc, 0x10, &base_addr);
    /*
    * Make sure this is a memory BAR
    * instead of an I/O 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
    * significance).
    */
    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 PCI memory space.
*/
    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 an init routine to be run later.
* ZZZ_if_init() calls isrlink() to set up the
*/
interrupt handler ZZZ_isr()

CONNECT_INIT_ROUTINE(isc, ZZZ_if_init);

* If everything okay, claim this card.
isc_claim(isc, &ZZZ_wsio_info);

* Exit without error

err1:

/*
 * Clean up the mapping.
 */
unmap_mem_from_host(isc, isc->if_reg_ptr, 512);

err0:

/*
 * Indicate an attach error.
 */
PCI_ATTACH_DEV_INIT_ERROR(isc);

exit0:

/*
 * Always exit by calling the rest of the chain.
 * Use the link established in ZZZ_install().
 */
return ZZZ_saved_attach(parm, isc);

Other Driver Entry Point Routines

The other routines defined by the previous code must also be declared and written. These functions include the following:

ZZZ_if_init Initialization of the card after the driver_attach routine.
ZZZ_isr The driver interrupt service routine.
ZZZ_open The defined entry point for open in the drv_ops_t structure.
ZZZ_close The defined entry point for close in the drv_ops_t structure.
ZZZ_read The defined entry point for read in the drv_ops_t structure.
ZZZ_write The defined entry point for write in the drv_ops_t structure.
ZZZ_ioctl The defined entry point for ioctl in the drv_ops_t structure.

The code for these functions is driver-dependent. See “Step 6: Writing Entry Point Routines” (page 102) and driver_close(9E), driver_if_init(9E), driver_ioctl(9E), driver_isr(9E), driver_open(9E), driver_read(9E), driver_write(9E) in the HP-UX 11i v3 Driver Development Reference.
This chapter presents the overall architecture of the HP-UX networking subsystem for designing and writing a device driver for Network Interface Cards (NICs) on HP-UX 11i v3. The HP-UX 11i v3 LAN driver architecture has the following changes:

- Support for driver instance online deletion
- Support for the `nwmgr` command and System Management Homepage (SMH)
- Obsolescence of FDDI and Token Ring tightly coupled drivers

Network driver writers can use the infrastructure and services provided by the HP-UX Core DLPI Infrastructure, referred to as HP-DLPI. HP-DLPI supports Ethernet NICs. Network driver writers developing drivers for Ethernet NICs no longer need to implement their own DLPI layer. Network drivers that have their own DLPI implementation are also supported. However, HP recommends that you consider switching to the HP-DLPI infrastructure and services. HP-DLPI defines a new interface, referred to as the HP-DLPI to Driver Interface.

Under the HP-DLPI to Driver interface, network interface drivers with their own DLPI implementation are called loosely coupled drivers. Network interface drivers that use the new infrastructure and services provided by HP-DLPI are called tightly coupled drivers. Unless otherwise noted, any reference to the term driver in this chapter means a tightly coupled networking device driver for Ethernet NICs.

For more information, see the STREAMS/UX for HP 9000 Reference Manual, the HP 9000 Networking DLPI Programmer’s Guide or the Data Link Provider Interface Specifications, UNIX International.

**HP-UX Networking Interface Driver Architecture**

Both tightly coupled and loosely coupled drivers are depicted in the architecture. Under this architecture, the HP-UX networking subsystem includes four logical layers:

- Application
- Protocol Interface
- Network Protocol
- Data Link

Figure 10-1 shows the network interface driver architecture on HP-UX 11i v3.
The following sections describe each layer in detail, with emphasis on the data link layer components of the architecture.

**Application Layer**

The application layer consists of user space networking application and commands. These commands and applications use either the sockets or device special files presented by the Protocol Interface layer to interact with the HP-UX kernel components that implement the remaining parts of the HP-UX Network Interface Driver Architecture.

DLPI applications and LAN commands open the device special file corresponding to the data link layer and use the DLPI 2.0 Application Programming Interface (API) and HP extensions to this API to interact with the data link layer. The data link layer is always a STREAMS driver. As a result, the opening of the device special files results in the creation of a stream-head in the HP-UX kernel.

All other networking commands and applications use either device special files of the network protocol layer or the socket interface.
Protocol Interface Layer

The protocol interface layer supports the application layer as follows:

- Identifies different applications on the same host through abstract objects like sockets and stream heads.
- Enables user applications to use the services provided by the network protocol layer or the data link layer to send and receive data (using sockets or stream heads).

The protocol interface layer uses STREAMS to interface with the network protocol and data link layers to accomplish the these functions.

Network Protocol Layer

The network protocol layer consists of STREAMS modules that implement the following common protocol families:

- Internet protocols: For example, TCP/IP, UDP/IP, and ARP.
- SNA protocol suite
- OSI protocol suite
- X.25 protocol suite

Each network protocol family defines an address scheme. The address schemes of a protocol family are not compatible with that of another, and are not interoperable.

The components of this layer are implemented as STREAMS modules. The data link layer below this layer must implement a STREAMS driver to interface with the STREAMS modules. The interface between the network protocol layer and the data link layer uses the DLPI 2.0 API and HP extensions to this API.

Data Link Layer

In HP-UX, the data link layer uses DLPI 2.0 API and HP extensions to this API using STREAMS services to interface with the network protocol layer components above it.

In HP-UX, the data link layer consists of the following:

- An HP-DLPI infrastructure component
  
  Implements DLPI 2.0 API, the full set of HP extensions to this API, and services for drivers. These services are accessed through a well defined HP-DLPI to driver interface.

- Tightly Coupled Drivers
  
  Implement a WSIO network interface driver supporting Ethernet NICs. All tightly coupled drivers rely on the DLPI implementation and services provided by HP-DLPI infrastructure component.

- Loosely Coupled Drivers
  
  STREAMS drivers that implement not only their own (native) DLPI implementation, but also a WSIO network interface driver to support a NIC of any type. The interface between the native DLPI implementation and the WSIO network interface driver portion are private to the driver implementation.

You can either write a tightly coupled driver that uses the HP-DLPI Infrastructure for a DLPI implementation (plus other services) or write a loosely coupled driver that requires you to have your own DLPI implementation to interface with the upper layers of the architecture. In either case, an WSIO-based network interface driver must be developed to program the NIC hardware. Both loosely and tightly coupled drivers are fully supported in HP-UX 11i v3. However, you can reduce development time and costs by developing a tightly coupled driver.

The HP-UX LAN infrastructure now supports 64-bit MIB statistics. Network interfaces that operate at 650 Mb/s or faster must now support 64-bit packet counters and 64-bit octet counters.
For more information about 64-bit MIB statistics, see Chapter 11 (page 221) and Chapter 12 (page 293).

**HP-DLPI Infrastructure**

The HP-DLPI infrastructure implements a DLPI 2.0 API, and HP extensions to this API, and other services that can be used by other components of the Data Link Layer to interact with the upper layers. HP-DLPI infrastructure defines a HP-DLPI to driver interface to interact with the tightly coupled drivers below this layer to use the services. Currently, tightly coupled drivers for Ethernet NICs are supported. Loosely coupled drivers can also use a limited subset of the services.

The services provided by the HP-DLPI infrastructure include the following:

- **Systemwide repository for network interface information**
  During initialization, drivers register network interface information with the HP-DLPI infrastructure. Drivers also update this repository to reflect any changes over time. This repository is accessible to upper layers and user applications through well documented HP extensions to the DLPI 2.0 API. The repository is not persistent and is rebuilt after every boot. Loosely coupled drivers must use the registration and update services offered by the infrastructure to work seamlessly with HP-UX LAN commands and other applications. This is the only service available to loosely coupled drivers.

- **Negotiation of NIC or driver and network layer features and Fastpath support**
  STREAMS ioctl’s are defined by the HP-DLPI infrastructure to enable the network protocol layer components to retrieve the features supported by the driver over which they operate.

- **Support for creating templates for Fastpath operation**
  This includes support for building MAC and LLC header templates that can be cached and used by the network layer on the outbound path, and for setting up a buffer template cached by drivers and used in the inbound path. Caching of header templates by the network layer and the driver can improve performance in the data path, and hence the term Fastpath. When building these header templates, space is reserved for storing extraneous per-frame information that is not part of the frame or packet, but is relevant on a per-frame basis. This extraneous information, also referred to as Out-Of-Packet (OOP) data, is used to exchange per-frame information between the driver, HP-DLPI infrastructure, and the network protocol layer components in inbound and outbound fast paths.

- **Outbound frame processing**
  Builds MAC and LLC headers on the outbound slow path for Ethernet MAC types before handing them off to drivers.

- **Support for different service modes for upper layers**
  Supports LLC Type 1 (connectionless), LLC Type 2 (connection-oriented), and a RAW mode data link service, which can be used by upper layers through the DLPI 2.0 API and HP extensions to this API.

- **Control request processing and control information repository**
  HP-DLPI infrastructure processes protocol, promiscuous, and multicast control requests, and maintains a repository for this information. Tightly coupled drivers can retrieve the information from this repository.
• Support for notification of link events to the interested upper layer entities
  Inbound frame processing, including parsing and demultiplexing of the protocol value in
  the frame for Ethernet MAC types, multicast and promiscuous mode filtering, and provision
  for handling IEEE XID/TEST frames.
• Support for some online replacement and deletion operations
  Provides information for performing critical resource analysis (CRA) on driver instances.
  Also enables safe online deletion of driver instances while the instances are in use by the
  networking stack or user space operations.

Tightly Coupled Drivers

Tightly coupled drivers implement WSIO-based drivers to program Ethernet PCI NIC hardware
to send and receive data, and operations that control the NIC hardware.

Tightly coupled drivers use the DLPI implementation provided by the HP-DLPI infrastructure.
These drivers register with the HP-DLPI infrastructure and use the services provided by the
infrastructure using the well-defined HP-DLPI to driver interface. In addition, tightly coupled
drivers use the HP-UX kernel services, WSIO services, and Network Tracing and Logging (NetTL)
services to implement the rest of the driver functionality.

Unlike loosely coupled drivers, tightly coupled drivers do not have their own DLPI
implementation and do not register with the STREAMS subsystem. However, they can use certain
STREAMS services to allocate and release data buffers (mblks). Tightly coupled drivers provide
some driver entry points during registration with HP-DLPI. Because HP-DLPI is a STREAMS
driver, these driver entry points can be invoked by HP-DLPI in a STREAMS context. Therefore,
tightly coupled drivers might need to follow certain restrictions laid down for STREAMS drivers.
For example, drivers must not sleep when these entry points are invoked by HP-DLPI.

For more information, see Chapter 12 (page 293).

Figure 10-2 shows the Data Link Layer Architecture for tightly coupled drivers.
Figure 10-2 Data Link Layer Architecture for a Tightly Coupled Driver

Upper Layers of the HP-UX Network Interface Architecture

- **HP-DLPI Infrastructure**
  - Implementation of DLPI 2.0 API and HP extensions to the API for use by upper layers
  - Defines the HP-DLPI to Driver Interface
  - Interface repository services for registration and update of network interface information
  - Facilitates option negotiation by the network protocol layer based on the driver features or options
  - Fastpath support
  - Provides connection oriented, connectionless and raw mode services for use by upper layers
  - Event modifications between upper layers and drivers
  - Processing of protocol, multicast, and promiscuous requests and a repository for such control information
  - Inbound frame processing, including protocol demultiplexing, promiscuous/multicast filtering, and handling of IEEE XID/TEST frames
  - Outbound frame processing: building the LLC/MAC headers before handoff to drivers

- **Tightly Coupled WSIO Network Interface Driver for Ethernet MAC Types**

- **PCI Bus**

See the kernel, WSIO, NetTL and network reference pages in the *HP-UX 11i v3 Driver Development Reference*.

For information on developing a tightly coupled driver, see Chapter 12 (page 293).

**Loosely Coupled Drivers**

Loosely coupled drivers implement both a STREAMS driver for their DLPI implementation and a WSIO interface driver that interact with each other using an internal, proprietary interface. The loosely coupled driver must also invoke HP-DLPI infrastructure services to register and update the systemwide network interface repository maintained by HP-DLPI. Registering with the HP-DLPI infrastructure makes the interface information available to all other components of the architecture.

Figure 10-3 shows the data link layer architecture for loosely coupled drivers.
The STREAMS DLPI driver portion of a loosely coupled driver must implement the DLPI 2.0 API and certain HP extensions to this API. You determine the actual implementation of the HP extensions. For example, certain HP extensions must be implemented for HP-UX LAN commands to work with the loosely coupled driver. For more information, see Chapter 13 (page 337).

The WSIO network interface driver portion of the loosely coupled driver controls the PCI-based NICs and can be any MAC type supported by the DLPI 1.0 standard. A driver for Ethernet NICs can be implemented either as a tightly coupled or as a loosely coupled driver. Because the HP-DLPI infrastructure does not support any other MAC types, implement drivers for other MAC types (for example, FDDI, Token Ring, and ATM) as loosely coupled drivers.

For information on developing a new loosely coupled driver, see Chapter 12 (page 293).

Driver Migration from Previous Releases

If you want to migrate a loosely coupled driver to HP-UX 11i v3, see the *HP-UX 11i v2 to 11i v3 Network Driver Migration Guide*. 

Driver Migration from Previous Releases
This chapter provides guidance and information on developing LAN device drivers that use HP-DLPI. In addition, this chapter includes detailed explanation on the interaction required between non-native drivers and HP-DLPI.

NOTE: This chapter does not focus on interactions with systems other than HP-DLPI (for example, WSIO, STREAMS, and Kernel Services), though drivers interact with these systems, too.

This chapter addresses the following topics:
- “Introduction to Non-Native LAN Driver”
- “Overview of Networking Driver Structure” (page 223)
- “Data Structures and Interfaces” (page 228)
- “Property Exchange Between HP-DLPI and Drivers” (page 233)
- “Initializing Non-Native Networking Device Drivers” (page 236)
- “Driver Control Function” (page 249)
- “Inbound Frame Processing” (page 264)
- “Outbound Frame Processing” (page 268)
- “MIB Statistics” (page 277)

All examples in this chapter are code fragments from the IELAN sample driver.

Introduction to Non-Native LAN Driver

HP-UX 11i v3 exports a comprehensive interface to network driver writers. This interface, HP-DLPI, is a layer between the transport layer and the Network Interface Card (NIC) device drivers. In this chapter, these drivers are called non-native (tightly coupled) network (LAN) drivers.

A non-native LAN driver relies on the HP-DLPI services to handle certain parts of control requests and ioctlS (inbound and outbound data processing). Figure 11-1 illustrates the relationship among non-native drivers, HP-DLPI, and other systems with which the driver interacts.
Architecture Components

Figure 11-1 illustrates how a non-native driver interacts with the following HP-UX kernel components:

HP-DLPI Driver Interface

HP-DLPI provides a set of interfaces to interact with drivers. These interfaces are discussed later in this chapter.

Driver Interface

A non-native driver must provide a set of functions for HP-DLPI. For example, the driver’s control function and outbound processing function. In addition, WSIO calls the install and attach routine of a driver to register the driver and attach it to a specific driver instance. WSIO also calls the interrupt service routine of the driver.

Driver Module

These routines are internal to the driver. They are called by other driver functions to manipulate the card hardware.

WSIO Services

A non-native driver uses WSIO services, attaches the driver to NIC, and sets up a driver instance to NIC. The driver uses WSIO services to register Interrupt Service Routine (ISR), then WSIO services call the driver ISR routine.
NetTL Services

If a driver supports network tracing and logging, it interacts with NetTL services.

PCI Bus

Drivers communicate with hardware using Direct Memory Access (DMA) and by writing to card registers directly. Card interrupts are sent through the bus to the host.

HP-UX Kernel Services

These areas consist of basic functions used by a driver. For example, routines to allocate memory blocks, copy memory blocks, acquire and release spinlocks, and so on, are kernel services.

HP-UX Networking Commands and Utilities

These services include System Management Homepage (SMH), `nwmgr`, `lanadmin`, and other HP-UX commands that use HP-DLPI. For example, SMH uses transparent ioctls to get and set driver parameters. It also uses `nwmgr` and `lanadmin` to configure card parameters. Furthermore, users can use `nwmgr` and `lanadmin` to get and set card parameters. The ioctls used in `lanadmin` are either HP-DLPI primitives, which are processed by HP-DLPI and passed to the driver, or are delivered by HP-DLPI (unprocessed) to the driver.

STREAMS Services

The kernel modules for the HP-UX transports (for example, TCP/IP, UDP, and OSI) are STREAMS modules. Drivers that interface with the transport stacks must work within this environment. A non-native LAN driver accesses the upper layer’s services through the HP-DLPI to Driver Interface.

Although minimal knowledge about HP-UX STREAMS is required to construct a non-native drivers under HP-DLPI, you must do the following tasks:

- Understand the structure of a message block (MBLK).
- Create, destroy, copy, and clone an MBLK.
- Traverse through an MBLK chain.

For information related to STREAMS modules and device drivers, see the following documents. Read the information on DLPI references carefully. This document briefly discusses the STREAMS mechanisms and concentrates more on specific HP variants.

**Hewlett-Packard Documents**

- **STREAMS/UX for HP 9000 Reference Manual**
- **HP 9000 Networking DLPI Programmer’s Guide**

**Other References**

- **Data Link Provider Interface Specifications**, Unix International

Overview of Networking Driver Structure

The flowchart in Figure 11-2 illustrates a suggested development sequence for developing a non-native network driver under HP-DLPI.

Step 1 (Standard Knowledge Base) lists the necessary information. Some information, such as “Protection and Synchronization for Non-Native Drivers” and “HP-UX STREAMS” (page 228) is not described completely in this chapter, but is still a prerequisite. For additional information on writing a device driver, see Chapter 4 (page 77).

Steps 2 through 9 list the options available for increased network driver capabilities. The information following the flowchart describes each capability.
Figure 11-2 Developing a Non-Native Networking Driver — Part 1

STANDARD KNOWLEDGE BASE

1. HP-UX LAN Architecture
2. Protection and synchronization for non-native drivers
3. Using HP-UX STREAMS
4. Data structures and interfaces
5. Property exchange between HP-DLPI and drivers
6. Initializing non-native networking drivers
7. Driver control function
8. Interrupt Service Routine (ISR)
9. Inbound packet processing
10. Outbound packet processing
11. Driver event notification
12. HP-DLPI to driver event notifications

OPTIONS

1. LAN Commands Support?
2. MIB Statistics?
3. nwmgr Support?
4. OOP Support?

MIB Statistics and Non-Native Drivers
nwmgr Support
OOP Support

LAN Commands
Figure 11-3 Developing a Non-Native Networking Driver — Part 2

OPTIONS (continued)

6. Interrupt Migration Support?
   - Yes: Interrupt Migration Support
   - No: SMH Support?

7. SMH Support?
   - Yes: Configuration of NIC on Boot and SMH Support
   - No: Log and Trace Support?

8. Log and Trace Support?
   - Yes: Logging and Tracing Support
   - No: OLARD Support?

9. OLARD Support?
   - Yes: Online Addition, Replacement, and Deletion of Cards and HP-DPLI
   - No: Driver Complete
1. This step in the network driver development lists the mandatory knowledge needed to customize the basic functions of driver. The topics are as follows:

HP-UX LAN Architecture

This section includes an overview of the HP-UX LAN Architecture. You must understand the overall LAN architecture before writing a non-native driver. See Chapter 10 (page 213).

Protection and Synchronization for Non-Native Drivers

This section describes the OSF/Encore spinlock protection model.

HP-UX STREAMS

This section includes functions and macros commonly used by STREAMS networking drivers. For more information on STREAMS programming, see HP-UX STREAMS Programmer’s Guide.

Data Structures and Interfaces

This section describes the interfaces between the major driver subsystems and non-native drivers.

Data Structures and Interfaces (continued)

From HP-UX 11i v2, driver properties are kept in a data structure maintained by the HP-DLPI layer. Drivers get the set values of these properties through an HP-DLPI supplied function. This does not indicate that driver properties must be stored in HP-DLPI only. Drivers can store their own copy of the same property. However, a driver must ensure that both the properties (one in HP-DLPI and the other in the data structure of a driver) have the same values.

Initializing Non-Native Networking Device Drivers

This section describes the install and initialization routines for a non-native LAN driver. The attach routine is discussed for an HP-UX driver. It includes IELAN code fragments, which illustrate how IELAN registers its capabilities with HP-DLPI.

Driver Control Function

This section describes the driver control function and the guidelines regulating the interactions between driver control functions and HP-DLPI. HP-DLPI needs drivers to implement a number of ioctls. The “Driver Control Function” (page 249) section describes the guidelines and the required ioctls. The section also includes examples for changing MTU, enabling multicast addresses, and driver-specific ioctls.

Interrupt Service Routine (ISR)

This section describes the interrupt services routine. In addition, it provides pointers on writing an interrupt service routine for a non-native driver.

Inbound Frame Processing

This section provides detailed descriptions about how to send frames to the upper layer. It describes the operation that the driver has to perform on a frame before passing it to HP-DLPI.

Outbound Frame Processing

This section explains how HP-DLPI directs a non-native driver to send out a packet. In addition, it provides guidelines and examples on how to do this. If the driver supports OOP data, it shows how to
traverse the OOP data and how to handle checksum offloading (CKO). For more information, see “Outbound Frame Processing” (page 268).

Notification of Driver Events
This section describes the events a driver is required to notify HP-DLPI. For example, when changing driver properties, HP-DLPI requires the driver to report a link down condition. After changing the property, the driver must report a link up to HP-DLPI.

HP-DLPI to Driver Event Notifications
This section describes how to handle the events sent by HP-DLPI to the driver.

Optional Capabilities of Non-Native Drivers
The following options are available when developing a network driver:

2. HP-UX LAN Commands Support
A properly written driver under HP-DLPI is supported by the linkloop and lanscan commands.
This section does not include information on the HP-UX LAN commands. For additional information on LAN commands, see Chapter 13 (page 337).

3. MIB Statistics and Non-Native LAN Drivers
The nwmgr command can display driver statistics. A non-native driver gathers some statistics and HP-DLPI gathers other statistics. For more information on the statistics that a non-native driver needs to collect and how to pass them to HP-DLPI, see “MIB Statistics” (page 277).

4. nwmgr Support for Non-Native Drivers
This section describes how to extend the nwmgr command to support driver-specific ioctls. In addition, it describes how nwmgr can support a non-native driver. To understand the HP-DLPI primitives that a non-native driver must implement to work with nwmgr and SMH, see “Driver Control Function” (page 249) and Chapter 13 (page 337).

5. Using OOP Data in Non-Native Drivers
This section describes how to incorporate OOP data in non-native drivers. For complete information on OOP, see Chapter 15 (page 377).

6. Interrupt Migration
HP-UX 11i enables interrupt service routines to be processed on a specific set of processors. This section describes how a driver deals with interrupt migration. For complete information on interrupt migration, see Chapter 22 (page 495).

7. NIC Configuration and SMH Support
This section describes how NICs controlled by a non-native driver can be configured by SMH. In addition, it describes how startup scripts function so that a NIC can be configured when HP-UX boots. For information on how a non-native driver can provide SMH support, see “NIC Configuration and NIC Tool Support” (page 287).

8. Logging and Tracing Support
This section describes how a non-native driver supports logging and tracing. For instructions, see Chapter 14 (page 353).

9. Online Addition, Replacement, and Deletion of Cards and HP-DLPI
This section includes information on how a driver interacts with HP-DLPI for OLARD events if a non-native driver supports online addition, replacement, or deletion of cards. For complete information on PCI OLARD, see Chapter 20 (page 455).
Protection and Synchronization for Non-Native Drivers

Synchronization issues with a non-native LAN driver are similar to the synchronization issues in the native LAN drivers.

In non-native LAN drivers, use spinlocks to prevent data corruption and race conditions to protect the driver instance data structure when multiple threads on different processors access the same driver instance. Furthermore, in drivers, protect driver data structure against interrupts. For non-native drivers, follow the rule of releasing the spinlocks when the drivers call HP-DLPI functions.

**NOTE:** Drivers must not hold any lock when calling HP-DLPI functions.

For more information on spinlocks, see Chapter 3 (page 69) and spinlock(9F) in the HP-UX Device Driver Reference.

**NOTE:** Each spinlock causes a busy wait. Be aware of the impact on system performance caused by the frequency of acquiring a spinlock and the duration of holding a spinlock. For more information on spinlocks, see the HP-UX Device Driver Reference.

HP-UX STREAMS

The message block, queue functions, and macros are defined by STREAMS/UX. For more information, see STREAMS/UX or the HP 9000 Reference Manual.

From HP-UX 11i v1, the header in the message block data structure, mblk_t, is not cacheline aligned. The area in an MBLK that stores data follows the header. Because 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 for reading DMA data, it can corrupt the message block header. This happens because the data area and the header share the same cacheline. Therefore, drivers must take precautions to avoid this problem. One solution is to verify that the data area and the header are in different cachelines.

The list of commonly used message block functions is as follows:

- allocb: Allocates a message block
- freemsg: Frees an MBLK chain
- freeb: Frees a message block
- pullupmsg: Concatenates and aligns the data stored in complex messages
- adjmsg: Adjusts the length of the message
- dupmsg: Duplicates a simple or complex message

Data Structures and Interfaces

A non-native network driver must interact with other HP-UX kernel subsystems to function. “HP-UX Architecture for Non-Native Drivers” (page 222) shows the interfaces a non-native drivers has with other subsystems.

The following sections describe the interfaces used to exchange information between a non-native driver and WSIO, HP-DLPI, HP-UX kernel services, and HP-UX LAN commands.

Interface with WSIO

WSIO is an HP-UX driver infrastructure service. To use WSIO services, a driver must register with WSIO. Registration is done when the driver_install routine is called. Later, WSIO calls the driver_attach routine, thereby enabling a driver to claim a card that it can support. After claiming the card, WSIO calls the driver_init routine, then initializes the card and sets up a driver instance. After initialization, the driver makes the card operational.
In addition to claiming a card for driver, `driver_install` also registers the driver event handler routine with WSIO. This event handler is called by WSIO for online replacement and interrupt migration events.

To register with WSIO, a driver must define three data structures. Be familiar with the following data structures:

`drv_info_t`  
A data structure that holds the general information about a driver. It describes the driver name, driver class, major and minor numbers.

**NOTE:** The driver name is the one that `ioscan` displays for a card managed by the driver.

For a non-native driver under HP-DLPI, the driver class must always be `lan`, and the major and minor numbers are always `-1` to enable HP-UX to assign them.

`drv_ops_t`  
A data structure that holds the pointers to the entry points in the driver for the read, write, select, open, and close entry points. A non-native driver under HP-DLPI must set it to NULL.

**NOTE:** For open, read, write, select, and option1, use `nodev` as the entry point. For `ioctl`, use `notty`; for close, use `nulldev`. Use NULLs for the rest of the entry points.

`wsio_drv_data_t`  
A data structure required for all WSIO drivers. For more information, see Chapter 4 (page 77).

**NOTE:** For a non-native driver, the `drv_path` field must be `lan_device`, the `drv_type` field must be `T_INTERFACE`, and the `drv_flags` field must be `DRV_CONVERGED`. The `drv_minor_build`, `drv_minor_decode`, `drv_get_minor`, and `io_path_mgr` fields must be NULL. A non-native LAN driver does not create any device file.

`wsio_drv_info_t`  
A table that holds pointers from the `drv_info_t` and `drv_ops_t` data structure of a driver. The table is passed to WSIO for registration.

For more information, see Chapter 4 (page 77).

In addition to initialization, a driver uses WSIO services for DMA operation. WSIO calls the WSIO event handler of a driver to process WSIO-generated events for OLARD or interrupt migration.

**Interface with HP-DLPI**

HP-DLPI serves as an intermediary between the non-native driver and the transport layer. The non-native driver performs the following tasks with HP-DLPI:

**Property Exchange with Drivers**

HP-DLPI provides a facility to maintain basic card properties (operational parameters, promiscuous states, MAC address, and so on). Drivers get and set these properties using the `dlpi_propp` interface. This is a key interface in HP-DLPI.

**Registration**

A non-native driver must register with HP-DLPI. After registration, the non-native driver receives a handle; this handle is used to call HP-DLPI functions. The registration also uses `dlpi_propp`. The properties of the card are written to the `dl_hp_create_info_t` data structure.
The driver provides information such as card instance number, card properties (speed and MTU), and pointers to control and outbound processing functions of the card.

For more information on registration, see “Step 4: Calling driver_init()” (page 239).

**Outbound Processing**

HP-DLPI passes the outgoing packet to a non-native driver. The driver processes this packet to remove the OOP header, then the packet passes to the card for transmission. If the CKO is supported, the driver directs the card to calculate the packet checksum.

HP-DLPI passes the packet as MBLK. If the driver supports OOP, the CKO information (cko_info_t) and the format of each OOP data in an OOP header (dl_hp_oop_hdr_t) is added in MBLK. For more information, see dlpi_outboundp(9F) in the HP-UX 11i v3 Driver Development Reference.

**Inbound Processing**

To process incoming data, a non-native driver obtains the packet data from the card, then passes the data to the HP-DLPI using a dlpi_inboundp function. The inbound processing function is obtained from HP-DLPI after registration.

**Control Processing**

HP-DLPI passes IOCTLs to the driver for processing. The control requests must be passed as MBLKs. When the control processing completes, the driver calls dlpi_wakeup to inform HP-DLPI. In addition, a non-native driver must also implement processing for a set of primitives and ioctl. For more information, see “Driver Control Function” (page 249) and “Driver to HP-DLPI Events” (page 274).

**HP-UX Kernel Services**

A non-native driver uses a function call from HP-UX Kernel Services to perform tasks such as requesting spinlocks, allocating memory, setting up DMA data to card, and so on.

**HP-UX LAN Commands**

The HP-UX nwmgr command enables users to obtain NIC statistics from a driver that includes MIB statistics support. Inclusion of MIB statistics enables nwmgr to obtain statistics from the driver. The nwmgr command displays driver statistics on number of bytes transmitted or received, number of inbound packets discarded or received, number of outbound frames discarded or transmitted, number of bytes transmitted, and number of transmit and receive errors.

The driver updates some of these data and HP-DLPI updates the rest. These statistics are kept in the MIB statistics mib_xEntry data structure.

The following fields in the MIB data structures are initialized in driver_init:

- ifMtu: MTU size currently in effect
- ifIndex: Network management ID of the NIC
- ifAdmin: Administrative status of the NIC
- ifOper: Operational status of the NIC
- ifSpeed: Current speed of the NIC
- ifType: Network type of the NIC

**Driver Instance Data Structure**

A driver has to manage data for each driver instance under its control. Data for each instance is maintained in a data structure. HP recommends that a driver allocates this data structure using
HP-UX kernel services during **driver_init** time. This ensures efficient and optimum usage of system resources.

The **ielan_ift_t** Data Structure

The **ielan_ift_t** data structure holds data used for each IELAN NIC instance. This data structure maintains the information for one IELAN interface. During **driver_init**, this data structure is allocated from memory. Most IELAN driver functions use a pointer to this data structure as the first argument. The following example illustrates this data structure.

typedef struct ielan_ift {
    /***************************************************************
    * originally shared with lanc/hwift
    ***************************************************************
    uint32_t instance_num;
    uint32_t nm_id;
    uint32_t mtu;
    dl_hp_encaps_type_t encapsulation;
    uint32_t features;
    uint32_t mac_type;
    uint32_t mac_addr_len;
    uint8_t mac_addr[LAN_PHYSADDR_SIZE];
    char *drv_name;
    char hdw_path[MAX_HDW_PATH_LEN];
    dl_hp_hw_state_t hdw_state;
    uint32_t promisc_level;
    /***************************************************************
    * Interfaces DLPI exported to drivers
    ***************************************************************
    void (*dlpi_inboundp)(void *, mblk_t *, void *, void *);
    void (*dlpi_wakeupp)(void *, mblk_t *, uint32_t, void *);
    int32_t (*dlpi_eventp)(void *, dl_hp_event_type_t, void *, void *);
    /***************************************************************
    * added due to DLPI re-architecture
    ***************************************************************
    uint32_t dpi_version;
    int32_t watch_timer;
    int32_t dma_timer;
    int32_t dma_time;
    void *dlpi_hdlp; /*handle for dlpi_prop() */
    mblk_t *__strong_order rwchan; /* wait channel for reset */
    mblk_t *mwchan; /* wait channel for multicast setup */
    /***************************************************************
    * PCI Configuration information
    ***************************************************************
    ubit16 vendor_id; /* Vendor ID */
    ubit16 device_id; /* Device ID */
    ubit16 sub_id; /* Subsystem ID */
    ubit16 sub_vendor_id; /* Subsystem Vendor ID */
    ubit8 base_class;
    ubit8 subclass;
    unsigned rev:4;
    unsigned step:4;
    unsigned fill:8;
    caddr_t cbma; /* HPA, memory access only */
    /***************************************************************
    * Device Specific Section
    ***************************************************************
    struct isc_table_type *isc;
    ielan_srom_t *srom; /* Serial ROM layout */
    ielan_drv_state_t drv_state; /* Driver state */
    ielan_sub_state_t sub_state; /* Driver sub state */
} ielan_ift_t;
/* First phy address found */
phy_addr;
/* Phy Type */
phy_type;
/* connected port */
port;
/* DMA_handle */
cable_state;

* Transmit Section

ielan_tb_t *tbr; /* Transmit buffer Ring */
ielan_td_t *tdr; /* Transmit Descriptor Ring */
caddr_t tdr_raw_phys_addr; /* Physical Address of TDR */

/* Count of active TDs */
tdr_act_cnt;
/* Active TD index */
tdr_act_index;
/* First free TD */
tdr_free_index;
/* Number of free TDs */
tdr_free_cnt;

/* Count of active TBs */
tbr_act_cnt;
/* Active TB index */
tbr_act_index;
/* Number of pending TBs */
tbr_pend_cnt;
/* First pending TB */
tbr_pend_index;
/* First free TB */
tbr_free_index;
/* Number of free TBs */
tbr_free_cnt;

volatile ubit32 tbr_complete_flag; /* Tx Completion Flag */
tbr_timed_index; /* tbr timer is set for */

* Receive Section

ielan_rd_t *rdr; /* Receive Descriptor Ring */
ielan_rb_t *rbr; /* Receive Buffer Ring */
caddr_t rdr_raw_phys_addr; /* Physical Address of RDR */

/* Current RD index */
rdr_index;
/* First Empty RD */
rdr_empty_index;
/* Number of empty RDs */
rdr_empty_cnt;
/* water for refill */
rdr_empty_watermark;

* Full Duplex, speed and Transmit Threshold settings

ielan_duplex_t duplex; /* Full Duplex setting */
ielan_speed_t speed; /* Speed setting (10/100 Mb) */
ielan_ctype_t conn_type; /* Selected Connection type */
xmit_threshold; /* CSR6 xmt threshold value */

ielan_tt_t tt_display; /* xmit thrsld display val */

* Local Driver Receive Stats

rcv_stats_t rstats; /* Receive Statistics */

* Local Driver Transmit Stats

trx_stats_t tstats; /* Transmit Statistics */

* Mib Specific Section

Ext_64bit_mib_t mibp;

* Lock and Other Synchronization Section

ielan_lock; /* ielan_ift lock */
Property Exchange Between HP-DLPI and Drivers

Driver properties are kept in data structures maintained by the HP-DLPI layer. Drivers get and set values of these properties through an HP-DLPI supplied function. This does not indicate that driver properties must be stored in HP-DLPI only. Drivers can have their own copy of the same property. However, a driver must ensure that both the properties (one in HP-DLPI and the other in the data structure of a driver) have same values.

The `dlpi_propp` function retrieves and sets property values from an HP-DLPI repository. In most cases, drivers can use the `dlpi_propp` interface only after the driver is registered with HP-DLPI. Registration is performed with the same function. See “Step 4c: Registering a Driver Instance with HP-DLPI” (page 241).

The prototype of the `dlpi_propp` function is as follows:

```c
int dlpi_propp(
    void *driver_handle,
    dl_hp_opt_t cmd,
    dl_hp_prop_t prop_name,
    void *valuep,
    void *rsvd1p,
    void *rsvd2p
);
```

Where:

- `driver_handle` Handle to the HP-DLPI driver instance created during registration. See “Step 4c: Registering a Driver Instance with HP-DLPI” (page 241).
- `cmd` Describes the operations that can be applied to a property. The operations are as follows:
  - `DL_HP_OP_GET` Retrieves a value. This value is returned in a data structure for that property.
**DL_HP_OP_SET** Sets a property value in the repository.

**DL_HP_OP_CREATE** Registers a driver instance with HP-DLPI (the property is DLPI_PROP_DRV_INSTANCEP).

**DL_HP_OP_DELETE** Unregisters a driver instance. Use this only after a driver registers an instance with HP-DLPI (DL_HP_OP_CREATE) and before the driver reports its hardware state to HP-DLPI.

---

**prop_name** Name of the property. For the complete list of HP-DLPI properties, see *dl_hp_prop_t*(9S) in the HP-UX 11i v3 Driver Development Reference.

**valuep** Pointer to the data structure that holds property value.

**rsvd1p** Reserved for future use. It must be set to 0 or NULL.

**rsvd2p** Reserved for future use. It must be set to 0 or NULL.

---

**Return Values**

The return values from *dlpi_propp* are as follows:

- **0** Success
- **EINVAL** Invalid property call
- **ENOSUP** Unsupported property or operation
- **EBUSY** HP-DLPI cannot currently handle this property
- **ENOMEM** HP-DLPI ran out of resources when attempting to meet this request
- **ENOBUFS** Allocation of an MBLK failed
- **ENOENT** No entries

---

**Rules for Using dlpi_propp()**

When using *dlpi_propp*, follow these guidelines:

- Do not hold locks while calling this routine.
- A call to *dlpi_propp* does not block or sleep.
- Caller must pass the buffer for the structure associated with a DL_HP_OP_GET or DL_HP_OP_SET operation. Any additional allocation required to meet the request is allocated by HP-DLPI. The allocation must be freed by the driver using the free routine returned along with the information.
- Pass an HP-DLPI private handle for most calls. Drivers must register with HP-DLPI before performing a get or set operation.
- HP-DLPI maintains a copy of some driver properties (such as speed, MTU, ifAdmin status) for giving them to drivers. Driver must set such properties when they are changed.
- Avoid *dlpi_propp* calls in performance paths.

---

**HP-DLPI Properties and Associated Data Structures**

For each property, there is an associated data structure. *Table 11-1* lists the associations for some of the properties. This table provides information on the properties that HP-DLPI maintains. For other properties, see *dl_hp_prop_t*(9S) in the HP-UX 11i v3 Driver Development Reference.
Table 11-1 Data Structures for HP-DLPI Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Data Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DLPI_PROP_EVENTP (get only)</td>
<td>pointer to a function</td>
<td>Pointer to a function for the driver to notify HP-DLPI about an interface that is up or down.</td>
</tr>
<tr>
<td>DLPI_PROP_WAKEUP (get only)</td>
<td>pointer to a function</td>
<td>Pointer to a function for the driver to wake up HP-DLPI after completing a control request.</td>
</tr>
<tr>
<td>DLPI_PROP_HDW_STATE</td>
<td>dl_hp_hw_state_t</td>
<td>Hardware State of the driver. See dl_hp_hw_state_t(9S) in the HP-UX 11i v3 Driver Development Reference.</td>
</tr>
<tr>
<td>DLPI_PROP_MCAST_INFOP</td>
<td>dl_hp_getinfo_t</td>
<td>Multicast address is stored in the datap field of the data structure. The driver must free datap after use. See dl_hp_get_info_t(9S) in the HP-UX 11i v3 Driver Development Reference.</td>
</tr>
<tr>
<td>DLPI_PROP_MAC_ADDR</td>
<td>an array of bytes</td>
<td>Physical address of the card.</td>
</tr>
<tr>
<td>DLPI_PROP_PROMISC_INFO</td>
<td>dl_hp_promisc_state_t</td>
<td>Promiscuous mode enabled on the card. See dl_hp_promisc_state_t(9S) in the HP-UX 11i v3 Driver Development Reference.</td>
</tr>
<tr>
<td>DLPI_PROP_OP_PARAM</td>
<td>dl_hp_op_param_t</td>
<td>Operational parameters for NIC: speed, MTU, duplex, and autonegotiation state. See dl_hp_op_param_t(9S) in the HP-UX 11i v3 Driver Development Reference.</td>
</tr>
<tr>
<td>DLPI_PROP_FEATURES_ONE</td>
<td>dl_hp_drv_features_one_t</td>
<td>Features and capability that a driver supports. See dl_hp_drv_features_t(9S) in the HP-UX 11i v3 Driver Development Reference.</td>
</tr>
<tr>
<td>DLPI_PROP_DRV_INSTANCEP</td>
<td>dl_hp_create_info_t</td>
<td>Helps the driver registration and deregistration. The driver sets up the data structure and passes dlpi_propp to register driver. See “Step 4c: Registering a Driver Instance with HP-DLPI” (page 241).</td>
</tr>
</tbody>
</table>

Information for dl_hp_getinfo_t

Special attention must be paid when HP-DLPI saves data into the dl_hp_get_info_t data structure. This data structure enables dlpi_propp (in a get operation) to return a list of any data structures to the caller. The list is kept as an array in the datap field. The dhg_count field holds the number of elements stored in datap. If the calling function does not need the array, the caller must free the array using the deallocation function provided in the free_funcp field of the data structure. For more information, see dl_hp_getinfo_t(9S) in the HP-UX 11i v3 Driver Development Reference.

NOTE: The dhg_count field records the number of elements stored in the array. The free_funcp function can be called only if dhg_count is greater than 0. Do not call free_funcp if dhg_count is equal to 0.

Deallocating a Multicast Address List Example

The following example illustrates how the IELAN driver frees the list of multicast addresses. The list is obtained from dlpi_propp using a get operation on the DLPI_PROP_MCAST_INFOP property. The caller defines the dl_hp_get_info_t data structure. The dlpi_propp sets up the dhg_datap field to point to the multicast address list. After use, the caller frees the dhg_datap field.
if( dlpi_propp(ielan_iftp->dlpi_hdlp ,DL_HP_OP_GET,
    DLPI_PROP_MCAST_INFOP, &pMcast, NULL, NULL) ) {
    msg_printf("ielan_ctl_req: dlpi_prop() get DLPI multicast list failed.\n");
    status = EINVAL ;
    break;
}

if(pMcast.dhg_count > IELAN_MAX_MULTICAST) {
    /* get promiscuous level */
    (void)dlpi_propp(ielan_iftp->dlpi_hdlp ,DL_HP_OP_GET,
        DLPI_PROP_PROMISC_INFO, &promisc, NULL, NULL) ;
    if( !( promisc & (DL_HP_PHYSICAL_LEVEL | DL_HP_MULTI_LEVEL)) ) {
        pMcast.dhg_free_funcp(pMcast.dhg_datap);
        status = EINVAL ;
        break;
    }
}

Initializing Non-Native Networking Device Drivers

This section describes the initialization process for a non-native driver. Excerpts from the IELAN driver are used as examples.

Initialize a driver using three routines. The initialization process follows these steps:

1. Set up the data structures to register the driver with WSIO.
2. driver_install registers the driver with WSIO.
3. driver_attach called for each unclaimed I/O device. This routine determines whether an I/O device can be manipulated by the card.
4. After driver_attach has claimed an I/O device, call the driver_init routine to initialize the instance of the driver that manages the device. This instance is called a driver instance.
5. Create a driver metadata file.

Step 1: Setting Up the Data Structures and the WSIO Event Handler

For IELAN, WSIO requires the following data structures to be defined and initialized before calling wsio_install_driver in the ielan_install:

driver_ops_t

   Driver-specific fields that all drivers use. These fields are pointers to the entry points for the open, close, read, write, ioctl, and a few system calls. For non-native drivers, set this to NULL.
   These values are no-OP functions because each of them returns an error code to the calling function.

The nodev function returns ENODEV, and notty returns ENOTTY.

static drv_ops_t ielan_drv_ops = {
    (d_open_t) nodev,        /* open */
    (d_close_t) nulldev,      /* close */
    NULL,              /* strategy */
    NULL,              /* dump */
    NULL,              /* psize */
    NULL,              /* reserved */
    (d_read_t) nodev,        /* read */
    (d_write_t) nodev        /* write */
    (d_ioctl_t) notty,       /* ioctl */
    (d_select_t) nodev,      /* select */
    (d_option1_t) nodev,      /* option1 */
    NULL,              /* reserved1 */
    NULL,              /* reserved2 */
    NULL,              /* reserved3 */
    NULL,              /* link */
    0,                 /* device flags */
};

236 Writing a LAN Driver Under HP-DLPI
wsio_drv_data_t
Driver specific fields for WSIO drivers. The fields of this data structure have the values as explained in “Interface with WSIO” (page 228).

static wsio_drv_data_t ielan_data = {
    "lan_driver",      /* for matching probes with drivers */
    T_INTERFACE,       /* type of hardware, device or interface */
    DRV_CONVERGED,    /* driver flag */
    NULL,     /* minor number build routine */
    NULL,     /* minor number decode routine */
};

The remaining fields, not explicitly initialized here, are set to NULL.

wsio_drv_info_t
To register a driver with WSIO, the driver must pass this data structure into the registration routine. The data structure holds pointers to the three configured data structures. For example:

static wsio_drv_info_t ielan_wsio_info = {
    (drv_info_t *) &ielan_drv_info,  /* driver info */
    (drv_ops_t *) &ielan_drv_ops,    /* driver ops */
    (wsio_drv_data_t *) &ielan_wsio_drv_data, /* driver Data */
    WSIO_DRV_CURRENT_VERSION
};

The last field must always be set to WSIO_DRV_CURRENT_VERSION.

driver_wsio_event_handler
If the driver supports additional capabilities like online addition, replacement, and deletion (OLARD) of cards and transaction-based interrupt, then the driver must handle some WSIO events. During driver installation, as part of the registration with WSIO, the WSIO event handler routine of the driver must also be registered.

Step 2: Installing the Driver

When the I/O system configuration process calls driver_install, the driver_install routine calls the driver_load routine, which calls wsio_install_driver routine to register the driver with the I/O subsystem and returns errors, if any. The prototype of the driver_install and driver_load functions are as follows:

void driver_install( void );
int driver_load ( void *drvinfop );

After this operation, the driver_load routine places the driver_attach entry in an attach chain to be called at I/O bus scan time. The pci_attach global variable is the pointer to the head of the list of driver_attach routines. If the driver is installed successfully, the old head is saved by the driver and the attach routine of the driver becomes the head.

Finally, the WSIO event handler function of the driver is registered.

Following is the code for ielan_install and ielan_load. These functions illustrate all the preceding processing.

/* to attach PCI driver to system */
int (*ielan_saved_attach)();
void ielan_install()
{
    if ( ielan_load(NULL) == 0 )
    {
        ielan_saved_pci_attach = pci_attach;
        pci_attach = ielan_attach;
        return;
    }
}

.  .  .
int ielan_load(void *drvinfop)
{
    if (drvinfop) // NOT called by ielan_install()
    {
        ielan_wsio_drv_info.drv_info=(drv_info_t *)drvinfop;
        if(mod_wsio_attach_list_add(MOD_WSIO_PCI, &ielan_do_attach,
            ielan_wsio_drv_info.drv_info->name))
        {
            msg_printf("ielan_load() FAILED. drv_info->name = %s\n",
                (drv_info_t *)drvinfop)->name);
            return ENXIO;
        }
    }
    if (wsio_install_driver((void *)&ielan_wsio_drv_info) != 1)
        return ENOREG; // Need to verify if its proper error
    if (wsio_install_drv_event_handler((wsio_drv_info_t *)&ielan_wsio_drv_info,
            ielan_wsio_event_handler) != WSIO_OK)
    {
        if(drvinfop)
            msg_printf("ielan_load(): wsio_install_drv_event_handler() failed.\n");
        else
            msg_printf("ielan_install(): wsio_install_drv_event_handler() failed.\n");
        return 0;
    }
}

Step 3: Calling driver_attach()

The prototype of the driver_attach routine is as follows:

int driver_attach(
    uint_32_t product_id,
    struct isc_table_type *isc_ptr
);

Where:

product_id    Four-byte PCI product ID.
isc_ptr       Pointer to the isc_table_type structure. Each instance of the driver is
              allocated an isc_table_type.

The driver_attach routine determines whether the NIC device ID matches those supported
by the driver. If they match, the routine claims the card using wsio_claim_node. Next, the
driver_attach routine calls the next attach routine in the chain by calling the
*driver_saved_attach routine.

When the driver_attach routine recognizes the device ID, it allocates and initializes its driver
control blocks and PCI I/O registers. In addition, the driver_attach routine sets up
driver_init and calls wsio_claim_node to claim the device.

The following code illustrates the attach routine for the IELAN driver. The ielan_attach
routine calls ielan_find_NIC_index to determine whether the card can be supported by
IELAN.

static void ielan_do_attach(uint32_t id, struct isc_table_type *isc){
    ubit32 sub_id;
    ubit32 olrad_timeout;
    wsio_event_mask_t event_mask=0;
    wsio_reg_node_t node_info;
    ielan_if_t *ielan_iftp;
    int set_ret;
    if (isc->if_reg_ptr == NULL)
        return ; // ielan_saved_pci_attach(id, isc);
    pci_read_cfg_uint32_isc(isc, SSID, &sub_id);
    if (ielan_find_NIC_index(id, sub_id) == -1)
        return; // ielan_saved_pci_attach(id, isc);

238 Writing a LAN Driver Under HP-DLPI
bzero((caddr_t)isc->gfsw, sizeof(struct gfsw));
isc->gfsw->init = (int (*)(()))ielan_init;
isc->gfsw->diag = (int (*)(()))NULL;
isc->if_id = id & 0xffff;
/* Allocation of ift structure is moved here from init */
/* because ielan_wsio_get_dev_attr might be called before init. */
ielan_arena_ielan_ift = kmem_arena_create(sizeof(ielan_ift_t),
    "ielan_arena_ielan_ift", NULL, M_WAITOK);
VASSERT(ielan_arena_ielan_ift != NULL);
ielan_iftp = (ielan_ift_t *) kmem_arena_alloc(ielan_arena_ielan_ift,
    M_WAITOK);
bzero((caddr_t)ielan_iftp, sizeof(ielan_ift_t));
isc->if_drv_data = (caddr_t )ielan_iftp;

// Prepare to claim interface. Initialize required
// wsio_reg_node_t fields.
// bzero((void *)&node_info, sizeof(wsio_reg_node_t));
node_info.version = WSIO_DRV_CURRENT_VERSION;
node_info.drv_info = &ielan_wsio_drv_info;
node_info.type = T_INTERFACE;
node_info.cb_func = ielan_wsio_get_dev_attr;
node_info.cb_arg = (void *)ielan_iftp;
node_info.events_mask = CB_GET_DEV_ATTR;
node_info.isc = isc;

// Claim interface using wsio_claim_node. This new interface allows us
// to pass a function pointer to a routine that implements the
// CB_GET_DEV_ATTR callback. CB_GET_DEV_ATTR support is required if
// the P_CIM flag is set for the health property.
if (wsio_claim_node(isc->card_node, &node_info) != WSIO_OK) {
    isc->if_drv_data = (caddr_t )NULL);
kmem_free ((caddr_t)ielan_iftp, M_DYNAMIC);
return;
}

event_mask = WSIO_EVENT_SUSPEND | WSIO_EVENT_RESUME |
WSIO_EVENT_REMOVE | WSIO_EVENT_LBI_INTR_MIGR |
WSIO_EVENT_BUS_ERROR;
/* WSIO backs out of functions wsio_reg_drv_capability_mask
* and wsio_set_parm if ielan_init() returns -1
*/
if (wsio_reg_drv_capability_mask(isc, event_mask) == WSIO_OK) {
    olrad_timeout = IELAN_OLRAD_TIMEOUT;
    if (wsio_set_parm(isc, WSIO_HW_SUSPEND_TIMEOUT,
        (void *)olrad_timeout) != WSIO_OK)
        if (wsio_reg_drv_capability_mask(isc,
            wsio_event_mask_t)0) != WSIO_OK)
            msg_printf("ielan_attach(): wsio_reg_drv_capability_mask(isc, 0) failed.\n");
else if (wsio_set_parm(isc, WSIO_HW_RESUME_TIMEOUT,
    (void *)olrad_timeout) != WSIO_OK)
        if (wsio_reg_drv_capability_mask(isc, (wsio_event_mask_t)0) != WSIO_OK)
            msg_printf("ielan_attach(): wsio_reg_drv_capability_mask(isc, 0) failed.\n");
else ;
}
return ; // ielan_saved_pci_attach(id, isc);

After the driver claims the card, the system calls driver_init so that the driver can initialize
the card and have the card operational.

NOTE: The driver_attach routine can be called many times before a match is found. For a
device in the first slot, the associated driver_attach routine is called by number of devices
in the PCI backplane. For a device in the last slot of the PCI backplane, the associated
driver_attach routine is called only once.

Step 4: Calling driver_init()

After the driver has claimed the card, initialize the card and the driver instance representing
the card using the driver_init function.
The prototype of the `driver_init` function is as follows:

```c
static int driver_init(
    struct isc_table_type *isc_ptr
);
```

Where:

- `isc_ptr`: Pointer to the `isc_table_type` structure. Each instance of the driver is allocated `isc_table_type`.

After `driver_attach` claims the card, WSIO calls the `driver_init` routine to begin initializing the card and driver instance. The initialization can be divided into the following steps:

- "Step 4a. Verifying HP-DLPI Version Compatibility" (page 240)
- "Step 4b: Initializing the Card Instance Data" (page 240)
- "Step 4c: Registering a Driver Instance with HP-DLPI" (page 241)
- "Step 4d: Retrieving HP-DLPI Callback Functions and NM_ID" (page 245)
- "Step 4e: Initializing a MIB Data Structure" (page 246)
- "Step 4f. Informing HP-DLPI about the Current Administrative and Operational Status" (page 247)
- "Step 4g. Setting Up ISR for a Driver Instance" (page 248)

These steps include a description of `ielan_init`. Each step describes the action a driver performs.

**Step 4a. Verifying HP-DLPI Version Compatibility**

Before starting the initialization, a driver must verify whether the HP-DLPI version can be used with the driver. To do this, call `dlpi_propp` with a get operation on the `DLPI_PROP_VERSION` property.

If the version is incorrect, the driver can take appropriate action.

The following code snippet is from `ielan_init`. It illustrates how to retrieve the HP-DLPI version using the `dlpi_propp` interface.

```c
static int ielan_init(struct isc_table_type *isc)
{
    /*
    * Determine whether the driver supports the DLPI version
    * in the kernel.
    */
    if( dlpi_propp(NULL ,DL_HP_OP_GET, DLPI_PROP_VERSION,
        &dlpi_version, NULL, NULL) ) {
        msg_printf("ielan_init: dlpi_prop() get DLPI version failed.\n");
        return(-1);
    }
    if( dlpi_version != DL_HP_DLS_VERSION) {
        msg_printf("%s: does not support this version-%d of DLPI. ielan_init() failed.\n", 
            driver_name, dlpi_version);
        return(-1);
    }
}
```

**Step 4b: Initializing the Card Instance Data**

If the HP-DLPI version is compatible with the driver, the driver can initialize its card instance data.

In this step, data for the current driver instance is initialized. In addition, the required DMA memory area is created. PCI vendor ID, PCI device ID, PCI sub-ID, subvendor ID, and revision IDs are saved on the card for future use. Use these IDs to do the following:

- Verify that the replacement card is identical to the old card, when OLA/R is supported.
- Debug the driver.
- Configure the driver to support different variants of NIC.
In IELAN, the `ielan_find_NIC_index` function determines the variant of IELAN NIC that the driver must manage.

In IELAN, an `ielan_ift_t` data structure stores the data for a driver instance. The `ielan_ift_t` data structure is defined in the `<ielan.h>` header file.

The following code snippet is an excerpt from the `ielan_init` code routine:

```c
/* Obtain memory for the driver instance variables in the ielan_ift_t data structure */
.. size = IELAN_TD_SIZE * IELAN_MAX_TDR_ENTRIES;
.. size = roundup(size, NBPG);
err = wsio_allocate_shared_memory(isc, size, &ielan_iftp->tdr_raw_phys_addr, (caddr_t *)&ielan_iftp->tdr, 0);
..
..
.. pcireadcfguint32_isc(isc, SSID, &ssid);

// Check for PCI bus error if (wsio_check_bus_error(isc) != 0) {
    msg_printf("ielan_init(inst=%d) WSIO_CHECK: PCI config space read for SSID returns -1.\n", ielan_iftp->instance_num);
    ielan_backout(ielan_iftp, isc, IELAN_PCICFG_RD_ERROR);
    return (-1);
}

ielan_iftp->sub_id = (ubit16)(ssid >> 16) ;
ielan_iftp->sub_vendor_id = (ubit16)(ssid & 0x0000ffff) ;
// Read the Configuration ID information *
pci_read_cfg_uint32_isc(isc, PCI_CS_VENDOR_ID, &id);
// Check for PCI bus error if (wsio_check_bus_error(isc) != 0) {
    msg_printf("%s: PCI Bus Error: PCI config space read for VEN_ID returns -1.\n", driver_name);
    ielan_backout(ielan_iftp, isc, IELAN_PCICFG_RD_ERROR);
    return (-1);
}

/* Read the Configuration Revision information */
pci_read_cfg_uint32_isc(isc, PCI_CS_REV_ID, &revid);
// Check for PCI bus error if (wsio_check_bus_error(isc) != 0) {
    msg_printf("ielan_init(inst=%d): PCI Bus Error: PCI config space read for REV_ID returns -1.\n", ielan_iftp->instance_num);
    ielan_backout(ielan_iftp, isc, IELAN_PCICFG_RD_ERROR);
    return (-1);
}

pci_write_cfg_uint32_isc(isc, CFDA, 0);
/* Turn on PCI memory access and bus master capability on host */

ielan_iftp->isc = isc;
/* Initialize PCI configuration data into the * driver instance data structure. */
.. ielan_iftp->vendor_id = CFID_VENDER_ID(id);
ielan_iftp->device_id = CFID_DEVICE_ID(id);
ielan_iftp->base_class = CPFV_BASE_CLASS(revid);
ielan_iftp->subclass = CPFV_SUBCLASS(revid);
ielan_iftp->rev = CPFV_REVISION_NUMBER(revid);
ielan_iftp->step = CPFV_STEP_NUMBER(revid);
/* Determine whether the NIC can be supported by * driver based on the PDI vendor and device IDs * return right away. */
..

nic_index = ielan_find_NIC_index(ielan_iftp->vendor_id << 16 | ielan_iftp->device_id, ielan_iftp->sub_id);
..
..
.. wsio_set_attributes (isc, IO_NONINTERLEAVED_DMA);
/* Get the cards instance number */
..
..
..
..
.. wsio_set_attributes (isc, IO_NONINTERLEAVED_DMA);
/* Get the cards instance number */
..
..
..
.. msg_printf("%s: wsio_isc_to_instance failed. ielan_init() failed.\n", driver_name);
    ielan_backout(ielan_iftp, isc, IELAN_ISC_INSTANCE_F);
    return(-1);
}
```

**Step 4c: Registering a Driver Instance with HP-DLPI**

After initializing the driver instance data, the `driver_init` function can register this driver instance with HP-DLPI.
The registration requires a driver to call `dlpi_propp` using the operation `DLPI_PROP_DRV_INSTANCEP`. The `dl_hp_create_info_t` data structure containing the information for the driver instance is passed to HP-DLPI. After the call returns, the unique opaque instance handle to HP-DLPI for this driver instance is returned in the `dhc_dlpi_hdlp` field.

The fields of the `dl_hp_create_info_t` structure are listed with information about how to populate them for a non-native network driver:

- **dhc_version** `(uint32_t)` Version of the data structure, which determines the interface for this driver instance. Only drivers can set this field during interface registration. For HP-UX 11i v3, the version is 3.

- **dhc_instance_num** `(uint32_t)` An identifier that maps 1-1 with a physical interface belonging to a particular class. Only drivers can set this field during interface registration. This field is displayed by the `lanscan` command.

- **dhc_ppa** Physical Point of Attachment (PPA) of the interface.

  Valid for native drivers only. For non-native LAN drivers, the `dhc_instance_num` is used as the PPA.

  Only drivers can set this field during interface registration. The `lanscan` command displays this field.

- **dhc_mac_type** `(uint32_t)` This field describes the MAC type of the NIC, and is settable by drivers during interface registration only. The `lanscan` command displays this field. Find the complete list of MAC types supported by HP-DLPI in `dl_hp_mac_type_t` in the HP-UX 11i v3 Driver Development Reference.

- **dhc_mac_addr_len** `(uint32_t)` Length (in bytes) of the MAC address. Only physical drivers can set this field during interface registration.

- **dhc_nm_id** `(uint32_t)` The network management ID. Loosely coupled drivers can set this field during interface registration. Later, set this field using the `DLPI_PROP_NMID` property.

  Non-native drivers ignore this field.

- **dhc_mtu** `(uint32_t)` Default MTU value in bytes. The recommended minimum value is 256.

  Only drivers can set this value during interface registration. Modify this value later using the `DLPI_PROP_OP_PARAM` property.

- **dhc_major_num** This field is used for native drivers only.

- **dhc_max_mcast** `(uint32_t)` Driver-specified maximum number of unique multicast addresses supported by the driver or NIC in the range 1-0x7FFFFFFF.

  Only physical drivers can set this field during driver instance registration. This field value cannot be modified later.

- **dhc_features_one** `(type dl_hp_drv_features_one_t) Features supported by the driver. See `dl_hp_drv_features_one_t` in the HP-UX 11i v3 Driver Development Reference.

  For most LAN drivers, the features are as follows:

  ```
  DL_HP_DRV_HP_DLS | DL_HP_DRV_LAN_CLASS | DL_HP_DRV_PHYSICAL
  ```
### dhc_features_two
Reserved for future use. Set to 0.

### dhc_features_three
Reserved for future use. Set to 0.

### dhc_encaps
Specifies the encapsulation type supported by the driver. HP-DLPI supports the following types:
- DL_HP_IEEE
- DL_HP_EXT_IEEE
- DL_HP_SNAP
- DL_HP_ETHERTYPE

### dhc_mac_addr
MAC address of the card. The maximum length is DL_HP_MAX_MAC_ADDR_LEN (32 bytes).

### dhc_hdw_path
Hardware path of NIC as a null terminated string. Maximum size DL_HP_MAX_HDW_PATH_LEN (100 bytes).

### dhc_drv_name
Driver name as a null terminated string. The name must be less than DL_HP_MAX_DRIVER_NAME_LEN (64 bytes).

### dhc_outputp
Outbound process function of the driver.

Drivers can set this field only during registration of the driver instance.

### dhc_build_hdrp
Driver routine to build the MAC/LLC header in the outbound path.

Only physical drivers can set this field value during driver instance registration.

### dhc_output_hdlp
Driver-specified opaque (to DLPI) handle passed to outbound routines.

### dhc_controlp
Driver routine for all control requests.

Drivers can set this field only during driver instance registration.

### dhc_control_hdlp
Driver-specified opaque (to DLPI) handle passed to the driver control request processing routine.

### dhc_eventp
Driver event handler for events defined by the DLS provider. See “Driver to HP-DLPI Events” (page 274). A driver can set this field only during driver instance registration.

### dhc_event_hdlp
Driver-specified opaque (to DLPI) handle passed to the driver event handler.

### dhc_drv_propp
This information is optional. Set to NULL.

### dhc_mod_hdlp
Opaque handle returned by DLPI on module-level registration with DLPI. Set to NULL.

### dhc_dlpi_hdlp
Instance opaque handle that drivers must use in future calls to DLPI interfaces.

### dhc_reserved[3]
Reserved for future use. Set to NULL.
If the driver registration fails, the driver cannot continue to initialize. The driver must not proceed with this `driver_init`. This indicates that resources (memory, DMA buffers, and so on), if any, must be freed.

The following code snippet illustrates how `ielan_init` registers the IELAN driver with HP-DLPI. Because the registration requires hardware path, `ielan_init` calls the `wsio_isc_to_hwpath` WSIO function. For the same reason, `ielan_init` calls the `wsio_isc_to_instance` function to obtain the NIC instance number. This value is saved in `instance_num` of the driver instance data.

**NOTE:** The `dhc_drv_propp` field is set to NULL so that HP-DLPI can assign a handle to this driver instance.

Upon successful registration, HP-DLPI puts the unique opaque handle in the `dhc_dlpi_hdlp` field of the `dl_hp_create_info_t` data structure. After this operation, the IELAN driver records the value in its instance data structure (`ielan_iftp->dlpi_hdlp`) so that it can be used in all future calls to HP-DLPI.

```c
/* Set the encapsulation methods supported */
lolan_iftp->encapsulation = (DL_HP_IEEE | DL_HP_EXT_IEEE | DL_HP_SNAP |
  DL_HP_ETHERTYPE);
create_struct.dhc_encaps = (DL_HP_IEEE | DL_HP_EXT_IEEE |
  DL_HP_SNAP | DL_HP_ETHERTYPE);
lolan_iftp->mac_addr_len = mac_addr_len = LAN_PHYSADDR_SIZE;
lolan_iftp->hw_state = HDW_STATE_DEAD;
bcopy((caddr_t)rom->mac, (caddr_t)ielan_iftp->mac_addr, |size_t)LAN_PHYSADDR_SIZE);
bcopy((caddr_t)ielan_iftp->mac_addr, |size_t)ielan_iftp->mac_addr_len);
lolan_iftp->features = (DL_HP_DRV_HP_DLS | DL_HP_DRV_LAN_CLASS | DL_HP_DRV_PHYSICAL |
  DL_HP_DRV_64BIT_MIB);
create_struct.dhc_features_one = ielan_iftp->features;
create_struct.dhc_features_one_cap = ielan_iftp->features;
create_struct.dhc_features_two = NULL;
create_struct.dhc_features_two_cap = NULL;
create_struct.dhc_features_three = NULL;
create_struct.dhc_features_three_cap = NULL;
lolan_iftp->mtu = ETHERMTU;
lolan_iftp->mac_type = DL_HP_DEV_ETHER;
.
.
/* Get the card instance number */
if (((isc->my_address = wsio_isc_to_instance(isc, NULL)) == WSIO_ERROR) {
  msg_printf("%s: wsio_isc_to_instance failed. ielan_init() failed.\n", driver_name);
  ielan_backout(ielan_iftp, isc, IELAN_ISC_INSTANCE_F);
  return(-1);
}
/* Initialize the create_struct (dl_hp_create_info_t) */
/* structure. This structure is used to register with DLPI. */
bzero(&create_struct, sizeof(dl_hp_create_info_t));
create_struct.dhc_version = ielan_iftp->dlpi_version;
lolan_iftp->instance_num = isc->my_address;
create_struct.dhc_instance_num = ielan_iftp->instance_num;
create_struct.dhc_ctlreq = (void (*)(void)) ielan_ctl_req;
create_struct.dhc_control_hdlp = ielan_iftp;
create_struct.dhc_eventtp = (int32_t (*)(void)) ielan_event_handler;
create_struct.dhc_event_hdlp = ielan_iftp;
create_struct.dhc_max_mcast = IELAN_MAX_MULTICAST;
create_struct.dhc_max_vlan = 0;
/* Get hardware path */
```

244 Writing a LAN Driver Under HP-DLPI
if (wsio_isc_to_hwpath(isc, ielan_iftp->hdw_path) != WSIO_OK) {
    msg_printf("%s: wsio_isc_to_hwpath failed. ielan_init() failed.\n", driver_name);
    ielan_backout(ielan_iftp, isc, IELAN_HW_PATH_F);
    return(-1);
}
strcpy(create_struct.dhc_hw_path, ielan_iftp->hdw_path);
/* REQUIRED: Manufacturing Initialize Statement */
msg_printf("%s: Initializing 10/100BASE-TX card at %s....\n", driver_name, ielan_iftp->hdw_path);
/* Set the encapsulation methods supported */
ielan_iftp->encapsulation = (DL_HP_IEEE|DL_HP_EXT_IEEE|
DL_HP_SNAP|DL_HP_ETHERTYPE);
create_struct.dhc_encaps = (DL_HP_IEEE|DL_HP_EXT_IEEE|
DL_HP_SNAP|DL_HP_ETHERTYPE);
create_struct.dhc_mac_addr_len = ielan_iftp->mac_addr_len =
LAN_PHYSADDR_SIZE;
ielan_iftp->hdw_state = HDW_STATE_DEAD;
bcopy((caddr_t)srom->mac, (caddr_t)ielan_iftp->mac_addr,
(size_t)LAN_PHYSADDR_SIZE);
bcopy((caddr_t)ielan_iftp->mac_addr,(caddr_t) create_struct.dhc_mac_addr,
(size_t)ielan_iftp->mac_addr_len);
ielan_iftp->features = (DL_HP_DRV_HP_DLS|DL_HP_DRV_LAN_CLASS
| DL_HP_DRV_PHYSICAL | DL_HP_DRV_64BIT_MIB);
create_struct.dhc_features_one = ielan_iftp->features;
create_struct.dhc_features_one_cap = ielan_iftp->features;
create_struct.dhc_features_two = NULL;
create_struct.dhc_features_two_cap = NULL;
create_struct.dhc_features_three = NULL;
create_struct.dhc_features_three_cap = NULL;
create_struct.dhc_mtu = ielan_iftp->mtu = ETHERMTU;
ielan_iftp->drv_name = driver_name;
strcpy(create_struct.dhc_drv_name , ielan_iftp->drv_name);
create_struct.dhc_mac_type = ielan_iftp->mac_type = DL_HP_DEV_ETHER;
create_struct.dhc_outputp = (uint32_t (*)(*))ielan_resolved_output;
create_struct.dhc_output_hdlp = ielan_iftp; 4
create_struct.dhc_build_hdrp = NULL;
create_struct.dhc_drv_propp=NULL;
/* Register with DLPI */
if( dlpi_propp(NULL,DL_HP_OP_CREATE, DLPI_PROP_DRV_INSTANCEP,
&create_struct, NULL,NULL) ) {
    /* If registration fails, undo all previous
     * initializations and allocations here.
     */
    ielan_backout(ielan_iftp, isc, IELAN_DLPI_INSTANCE_F); 5
    return -1;
}  
ielan_iftp->dlpi_hdlp = create_struct.dhc_dlpi_hdlp;

1 By setting DL_HP_DRV_64BIT_MIB, the driver supports 64-bit MIB statistics.
2 The dhc_controlp_hdlp value is passed to the ielan_ctl_req function.
3 The dhc_event_hdlp function is passed to the ielan_event_handler function.
4 The dhc_output_hdlp value is passed to the ielan_resolved_output function.
5 When the registration fails, ielan_backout is called to unregister the driver instance with
HP-DLPI. It also undoes any memory allocations that have been made. The
IELAN_DLPI_INSTANCE_F marker informs ielan_backout about the starting point of
the backout.

Step 4d: Retrieving HP-DLPI Callback Functions and NM_ID

After completing the registration, the driver must obtain the following callback functions from
HP-DLPI:
dlpi_inboundp Pointer to HP-DLPI's inbound packet processing routine
dlpi_wakeupp HP-DLPI routine that indicates the completion of the processing of a control request
Then, the driver obtains the Network Management ID (NM_ID) from HP-DLPI.

For specific details on these routines, see `dlpi_inboundp(9F)`, `dlpi_wakeupp(9F)`, and `dlpi_eventp(9F)` in the HP-UX 11i v3 Driver Development Reference.

If any of the `dlpi_propp` calls fails to obtain these routines, the driver cannot continue initialization. In addition, the previously allocated memory and DMA buffers must be freed and the driver instance must unregister itself with HP-DLPI.

The following code fragment illustrates how `ielan_init` obtains three callback routines and NM_ID.

```c
/* Get the interfaces that DLPI exports to drivers */
if (dlpi_propp(ielan_iftp->dlpi_hdlp, DL_HP_OP_GET, DLPI_PROP_INBOUNDP,
    &ielan_iftp->dlpi_inboundp, NULL, NULL) ) {
    /* Unregister driver instance */
    dlpi_propp(ielan_iftp->dlpi_hdlp, DL_HP_OP_DELETE,
        DLPI_PROP_DRV_INSTANCEP, NULL, NULL, NULL);
    /* Back out of all other initializations and allocations. */
    ielan_backout(ielan_iftp, isc, IELAN_DLPI_PROP_F);
    return -1;
}

if (dlpi_propp(ielan_iftp->dlpi_hdlp, DL_HP_OP_GET, DLPI_PROP_WAKEUPP,
    &ielan_iftp->dlpi_wakeupp, NULL, NULL) ) {
    /* Unregister driver instance */
    dlpi_propp(ielan_iftp->dlpi_hdlp, DL_HP_OP_DELETE,
        DLPI_PROP_DRV_INSTANCEP, NULL, NULL, NULL);
    /* Back out of all other initializations and allocations. */
    ielan_backout(ielan_iftp, isc, IELAN_DLPI_PROP_F);
    return(-1);
}

if (dlpi_propp(ielan_iftp->dlpi_hdlp, DL_HP_OP_GET, DLPI_PROP_EVENTP,
    &ielan_iftp->dlpi_eventp, NULL, NULL) ) {
    /* Unregister driver instance */
    dlpi_propp(ielan_iftp->dlpi_hdlp, DL_HP_OP_DELETE,
        DLPI_PROP_DRV_INSTANCEP, NULL, NULL, NULL);
    /* Back out of all other initializations and allocations. */
    ielan_backout(ielan_iftp, isc, IELAN_DLPI_PROP_F);
    return(-1);
}

if( dlpi_propp(ielan_iftp->dlpi_hdlp, DL_HP_OP_GET, DLPI_PROP_NMID,
    &ielan_iftp->nm_id, NULL, NULL) ) {
    msg_printf("ielan_init: dlpi_prop() get DLPI nm id failed.\n");
    ielan_backout(ielan_iftp, isc, IELAN_DLPI_PROP_F);
    return(-1);
}
```

**Step 4e: Initializing a MIB Data Structure**

After obtaining HP-DLPI callback routines, the driver must now set up the MIB data structure. During initialization, some MIB statistics of the driver instance are set to default values, while others are set to zero. The `ifAdmin` member indicates whether the driver can be used for data traffic. An `ifAdmin LINK_UP` state indicates that it can be used for data traffic. The `ifOper` member indicates whether the link is established on the NIC.

At the driver initialization phase, the link is down (LINK_DOWN). All other values are set to their default.

The following code fragment is for the IELAN driver. For IELAN, the speed of the driver is set to its default value (`ETHER_BANDWIDTH`, or 10 Mb/s), the `ifType` value is set to `ETHERNET_CSMACD`, and the `ifMtu` value is set to ETHERMTU (1500). As it is an Ethernet NIC, the IELAN driver collects data on some collision statistics (`dot3_ext_coll` data structure) and other 802.3 statistics (`dot3_ext_stats` data structure).
/* Initialize the MIB objects */
mib_ptr = &ielan_iftp->mibp.mib_if;
mibX_ptr = &ielan_iftp->mibp.mib_Xif;

/* Initialize length field for the MAC address */
mib_ptr->ifPhysAddress.o_length = LAN_PHYSADDR_SIZE;
bcopy((void *)srom->mac, (void *)IELAN_PHYADDR(mib_ptr),
(size_t)LAN_PHYSADDR_SIZE);
mib_ptr->ifIndex = ielan_iftp->nm_id;
mib_ptr->ifType = IELAN_ETHERNET_CSMACD;
mib_ptr->ifMtu = ETHERMTU;
mib_ptr->ifSpeed = ETHER_BANDWIDTH;
mibX_ptr->ifHighSpeed = ETHER_BANDWIDTH / IELAN_MBPS;
mib_ptr->ifAdmin = LINK_UP;
mib_ptr->ifOper = LINK_DOWN;
mib_ptr->ifSpecific = ID_dot3;

ielan_iftp->mibp.mib_specific.dot3_Xmib.dot3Xcolls.dot3CollIndex = mib_ptr->ifIndex;
ielan_iftp->mibp.mib_specific.dot3_Xmib.dot3Xstats.dot3StatsIndex = mib_ptr->ifIndex;

Step 4f. Informing HP-DLPI about the Current Administrative and Operational Status

After establishing the MIB parameters, HP-DLPI must be informed about the current operational parameters and the current administrative status. Because the current administrative status has been set to LINK_UP, inform HP-DLPI of this status through the dlpi_propp call using a SET operation on the DLPI_PROP_IFADMIN property.

The operational parameters are the NIC's current speed, duplex, and autonegotiation status (on/off). HP-DLPI is informed about the current operational parameter by the driver calling dlpi_propp with a SET operation on the DLPI_PROP_OP_PARAM property of the dlpi_hp_op_param_t data structure. The parameter type is described by dop_param_type, speed is described by dop_speed, duplex is described by dop_duplex, and autonegotiation is described by dop_auto_neg_sense.

Many operational parameter types can be set at once. For example, in the following IELAN code, speed, duplex, and autonegotiation status are set simultaneously with one call to dlpi_propp.

For more information, see dlpi_hp_op_param_t(9S) in the HP-UX 11i v3 Driver Development Reference.

The following code fragment illustrates how ielan_init informs HP-DLPI about the operational and administrative status. During initialization, the IELAN's speed and duplex is set to 10 M/bps and half duplex, with autonegotiation turned on. For the administrative status, dlpi_propp is called to set the DLPI_PROP_IFADMIN property. The value of this property is in ifAdmin. The ifAdmin value of DL_HP_IFADMIN_UP indicates that the administrative status is up.

dl_hp_op_param_t op_param;
...

if (dlpi_propp(ielan_iftp->dlpi_hdlp,DL_HP_OP_SET, DLPI_PROP_OP_PARAM,
&op_param, NULL, NULL) ) {
    msg_printf("ielan_init: dlpi_prop() set DLPI OP_PARAM failed.
    ielan_backout(ielan_iftp, isc, IELAN_DLPI_PROP_F);
    return(-1);
}

ifAdmin = DL_HP_IFADMIN_UP;
if (dlpi_propp(ielan_iftp->dlpi_hdlp,DL_HP_OP_SET, DLPI_PROP_IFADMIN,
    &ifAdmin, NULL, NULL) ) {
    msg_printf("ielan_init: dlpi_prop() set DLPI IFADMIN failed.
    ielan_backout(ielan_iftp, isc, IELAN_DLPI_PROP_F);
    return(-1);
}
Step 4g. Setting Up ISR for a Driver Instance

In this step, the driver sets up an Interrupt Service Routine (ISR) for the driver instance. To set up an ISR, an interrupt object must be allocated from WSIO using the `wsio_intr_alloc` routine.

Following the call to `wsio_intr_alloc`, the driver must indicate whether NIC supports line-based or transaction-based interrupts. The IELAN driver uses line-based interrupts; therefore, it calls `wsio_intr_set_irq_line` to set an IRQ line. To get the line-based interrupt information, use `wsio_intr_get_irq_line`.

If a driver supports transaction-based interrupts, the driver calls `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.

After this process, the driver enables the interrupt by calling `wsio_intr_activate`.

For more information on the usage of these routines, see Chapter 4 (page 77). In addition, see the manpages for these routines in the HP-UX 11i v3 Driver Development Reference.

The following code snippet illustrates how `ielan_init` allocates and activates the `ielan_isr` interrupt routine for a particular IELAN instance. The first argument is the argument passed to `ielan_init`, a pointer to the ISC table that represents this NIC device. `ielan_isr` is the interrupt service routine. The argument passed into `ielan_isr` is a pointer to the driver instance data, `ielan_iftp`. The handle to the interrupt object is placed in `ielan_iftp->ielan_wsio_intr`.

After this operation, the `wsio_set_irq_line` routine is called to set the IRQ value for this device. In IELAN, the value is set to `WSIO_IRQ_LINE_AUTO`, which makes WSIO determine the value for the driver.

If any of the calls to WSIO fails here, the driver instance is inoperable. In this case, all actions performed until this point must be reversed using the `ielan_backout` routine.

```c
/* Set up interrupt handler. */
if(wsio_intr_alloc(isc, (wsio_drv_isr_t) ielan_isr, (uintptr_t)ielan_iftp,
    WSIO_INTR_SHARED, &(ielan_iftp->ielan_wsio_intr)) != WSIO_OK) {
    /* OLA/R ISR linking failed */
    msg_printf("ielan - interrupt object allocation failed...
    ielan_backout(ielan_iftp, isc, IELAN_INTR_ALLOC_F);
    return -1;
}
if(wsio_intr_set_irq_line(isc, ielan_iftp->ielan_wsio_intr,
    WSIO_IRQ_LINE_AUTO, WSIO_INTR_LEVEL_SENSITIVE) != WSIO_OK) {
    /* OLA/R ISR linking failed */
    msg_printf("ielan - assigning IRQ line failed...
    ielan_backout(ielan_iftp, isc, IELAN_IRQ_ASGN_F);
    return -1;
}
if(wsio_intr_activate(isc, ielan_iftp->ielan_wsio_intr) != WSIO_OK) {
    /* OLA/R ISR linking failed */
    msg_printf("ielan - activating interrupt line failed...
    ielan_backout(ielan_iftp, isc, IELAN_INTR_ACTV_F);
    return -1;
}
}

Step 5: Creating a Driver Metadata File

To configure the driver into a kernel, a metadata file must be provided. The driver modmeta file describes the driver module properties.

The IELAN’s metadata file is listed in “IELAN Metadata File Example” (page 249). Use it as a template.
IELAN Metadata File Example

```c
module ielan {
    desc    "IELAN : HP-UX 11i v3.0 (11.31) HP-DLPI based sample driver"
    type      wsio_intfc
    version  3.2
    states static
    dependency dlpi
    dependency pci
    dependency wsio
    initfunc driver_install ielan_install static
    driver  {
        char
        class  lan
        flags  mp_safe  save_conf
    }
}
```

For more information on metadata files, see Chapter 6 (page 157) or `modmeta(4)`. A non-native PCI driver that uses HP-DLPI must have the following module dependencies:

- dlpi
- liblan
- libpci
- libstream
- wsio

### Driver Control Function

The driver control function manages changes to operating properties of a NIC. HP-DLPI requires a non-native driver to handle some primitives (control requests that may have been processed by HP-DLPI before being passed to the driver control function for further processing) or transparent ioctls (control requests that are passed to the driver by HP-DLPI without being processed by HP-DLPI). A non-native driver is required to handle a list of primitives and transparent ioctls. See “Primitives Required to be Supported by Drivers” (page 252) and “Information on Transparent ioctls” (page 255), respectively.

In a non-native driver, all control processing request is initiated from HP-DLPI, which passes control requests to the driver for further processing by calling the `driver_control` function. The `driver_init` function sets up a pointer to this function when registering this driver instance with HP-DLPI.

The prototype for the driver control function is as follows:

```c
void driver_control(
    void *driver_handle,
    int32_t cmd,
    mblk_t *mblkp,
    void *rsudp
);
```

Where:

- `driver_handle` Contains the value of `dhc_control_hdlp`. This value is set during the registration of this driver instance with HP-DLPI. See the sample code in “Step 4c: Registering a Driver Instance with HP-DLPI” (page 241). In the IELAN driver, this value is a pointer to the driver instance data structure.

- `cmd` Primitive name or ioctl request. For more information and list of supported primitives and ioctls, see “Control Request Types” (page 251) and `driver_controlp(9F)` in the HP-UX 11i v3 Driver Development Reference.

Driver Control Function 249
**mblkp**  
Pointer to MBLK containing details of the command request. For more information on the data type associated with each request, see `driver_controlp(9F)` in the HP-UX 11i v3 Driver Development Reference.

---

**NOTE:** The pointer must be passed to HP-DLPI when the driver completes the processing of the request.

**rsdvp**  
Reserved for future use. HP-DLPI passes NULL.

### Control Flow Processing

Figure 11-4 (page 250) illustrates the flow of information for a control request that originated from a DLS user. In this case, the DLS user can be `nwmgr` or any other HP-UX LAN command.

The control request originates from the user and is passed to HP-DLPI, which determines whether it must process the control request. If the control request must be passed to the driver, HP-DLPI passes the control request to the `driver_control` function. The control request and associated information is passed in an MBLK. The driver then processes the routine.

After the request processing is complete, the driver must call `dlpi_wakeupp` with the status of the processing. The status is 0 if the request processed successfully. Otherwise, a non-zero value that describes the error condition is returned. In addition, the driver must pass MBLK (passed to `driver_controlp`) as an argument to `dlpi_wakeupp`. After the driver calls `dlpi_wakeupp`, HP-DLPI relays the status back to the DLS user.

---

**NOTE:** HP-DLPI does not require a control request to complete processing within the thread of execution that invoked `driver_controlp`. When a driver completes processing, it uses `dlpi_wakeupp` to notify HP-DLPI about its completion and the processing status. The MBLK passed to `dlpi_wakeupp` specifies the control request that completes the processing.

---

**Figure 11-4 Process Flow of a Control Request**

[Diagram showing the process flow of a control request]

---

250  Writing a LAN Driver Under HP-DLPI
Implementation Guidelines for driver_control()

When implementing `driver_control()`, follow these guidelines:

- After processing a control request, `driver_control` must wake up HP-DLPI using the `dlpi_wakeupp` call. A missed wake-up call by the driver can hang an application that uses these ioctlts. For more information, see “Control Request Types” (page 251) or `dlpi_wakeupp(9F)` in the HP-UX 11i v3 Driver Development Reference.
- While processing a control request, a driver can wake up HP-DLPI on a thread that is different from the thread in which HP-DLPI called `driver_control`.
- No locks must be held while calling `dlpi_wakeupp`.
- Multiple control requests can be sent to the driver simultaneously. The driver must queue and handle them. Drivers can return an error if they receive a control request while handling another control request. However, take care to retain the request MBLK to wake up HP-DLPI with error.
- If the control request changes the property of the interface (for example, MTU, speed, and MAC address), the drivers must inform HP-DLPI about the change using the `dlpi_propp` interface. In addition, it may need to generate link DOWN and UP events. For more information, see “Driver to HP-DLPI Events” (page 274) and `dlpi_eventpp(9F)` in the HP-UX 11i v3 Driver Development Reference.
- No locks are held by HP-DLPI while calling `driver_control`.
- This interface must be provided by the driver to HP-DLPI during property exchange as a `dhc_controllp` element of the `dl_hp_create_info_t` structure. To see how to provide this interface information to the HP-DLPI, see the ielan code fragment.
- Driver-specific applications and drivers can define ioctlts that are transparent to DLPI. Transparent ioctlts pass directly to the driver without any processing in DLPI. For constraints on defining transparent ioctlts, see “Information on Transparent Ioctlts” (page 255).
- Drivers must not modify the control request (first) MBLK.
- Drivers must not modify `b_rptr` (if any) of the second MBLK.

Control Request Types

There are different types of control requests that the `driver_control` function of a non-native driver handles. The different types of control functions are as follows:

- **Primitives** Controls that are processed by HP-DLPI before being passed to a driver for further processing.
- **Ioctlts** Controls in which the driver is required to check whether permission is granted to process these ioctlts.
- **Transparent ioctlts** Controls that HP-DLPI passes directly to the driver for processing.

Handling of these control requests is described in the following sections.

Handling a Primitive

A driver handles a primitive depending on the specifications for the primitive by the DLPI standard or by HP (if it is an HP-specific primitive). The driver does the following:

- Reads the request and takes appropriate action, for example, `DL_HP_HW_RESET_REQ`. After the driver receives this request, it takes appropriate action and returns the status of its actions (success/failure) to HP-DLPI through `dlpi_wakeupp`.
- If necessary, reads the request and writes the requested information to the attached MBLK.

HP-DLPI handles this type of request using the following steps:
1. Allocates an MBLK for the driver to write into. This MBLK must be large enough to contain an acknowledgement to the original request.

2. Moves b_rptr of the allocated MBLK by the size of the acknowledgement. This enables the driver to write into this MBLK in the correct location, without having to know about the internals of acknowledgement (such as size).

3. Chains this MBLK to the request MBLK. This enables the driver to identify the request primitive. The driver must neither break the chain of MBLKs sent to it, nor free any of the MBLKs.

The format of the MBLK passed to the driver for the DL_GET_STATISTICS_REQ primitive is described in “The DL_GET_STATISTICS_REQ MBLK Chain” (page 252). The b_rptr of the request (first) MBLK holds the request type dl_get_statistics_req_t. See “Request Data Structures Associated with Primitives” (page 253). This request asks the driver to report the MIB statistics it collected. HP-DLPI expects the driver to copy the statistics into the Ext_mib_t data structure in the second MBLK. The start address of the Ext_mib_t data structure is at b_rptr of the second MBLK.

Figure 11-5 The DL_GET_STATISTICS_REQ MBLK Chain

---

The start address of the request data structure is always at b_rptr of the first MBLK. If the driver must pass data to HP-DLPI, the driver writes the data in the data structure of the second MBLK.

Primitives Required to be Supported by Drivers

Table 11-2 lists the primitives that a non-native driver must support. Table 11-3 (page 253) lists the primitives and their request data structures. Table 11-4 (page 254) lists the primitives that a non-native driver must return to HP-DLPI, along with their associated response data structure.

Table 11-2 Required DLPI Primitives and Iocltls

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL_ENABMULTI_REQ</td>
<td>Enables a multicast address.</td>
</tr>
<tr>
<td>DL_DISABMULTI_REQ</td>
<td>Disables a multicast address.</td>
</tr>
<tr>
<td>DL_PHYS_ADDR_REQ</td>
<td>Retrieves the MAC address of an interface.</td>
</tr>
<tr>
<td>DL_SET_PHYS_ADDR_REQ</td>
<td>Sets the MAC address of an interface.</td>
</tr>
<tr>
<td>DL_HP_HW_RESET_REQ</td>
<td>Resets the NIC.</td>
</tr>
<tr>
<td>DL_PROMISCOFF_REQ</td>
<td>Disables the promiscuous mode on the NIC.</td>
</tr>
<tr>
<td>DL_PROMISCON_REQ</td>
<td>Turns promiscuous mode off.</td>
</tr>
<tr>
<td>DL_HP_RESET_STATS_REQ</td>
<td>Request to reset the MIB statistics of an interface (physical/logical).</td>
</tr>
</tbody>
</table>
### Table 11-2 Required DLPI Primitives and ioctl(continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL_GET_STATISTICS_REQ</td>
<td>Request to retrieve the standard and extended MIB statistics. HP-DLPI passes</td>
</tr>
<tr>
<td></td>
<td>a dl_get_statistics_req_t data structure in the MBLK.</td>
</tr>
<tr>
<td>DL_HP_GET_MIBSTATS_REQ</td>
<td>Request to retrieve the standard and extended MIB statistics applicable to all</td>
</tr>
<tr>
<td></td>
<td>interfaces. The driver passes a dl_hp_get_mibstats_req_t data structure in</td>
</tr>
<tr>
<td></td>
<td>the MBLK.</td>
</tr>
<tr>
<td>DL_HP_SET_IFADMIN_REQ</td>
<td>Sets the ifAdmin value as given in the dl_ifAdminStatus element of this</td>
</tr>
<tr>
<td></td>
<td>structure.</td>
</tr>
</tbody>
</table>

### Table 11-3 Request Data Structures Associated with Primitives

<table>
<thead>
<tr>
<th>Primitives</th>
<th>Data Structure</th>
<th>Data Structure Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL_GET_STATISTICS_REQ</td>
<td>dl_get_statistics_req_t</td>
<td>mblkp-&gt;b_rptr</td>
</tr>
<tr>
<td></td>
<td>See dl_get_statistics_req_t(9S) in HP-UX 11i v3 Driver Development Reference.</td>
<td></td>
</tr>
<tr>
<td>DL_HP_GET_MIBSTATS_REQ</td>
<td>dl_hp_get_mibstats_req_t</td>
<td>mblkp-&gt;b_rptr</td>
</tr>
<tr>
<td></td>
<td>See dl_hp_get_mibstats_req_t(9S) in HP-UX 11i v3 Driver Development Reference.</td>
<td></td>
</tr>
<tr>
<td>DL_HP_RESET_STATS_REQ</td>
<td>dl_hp_reset_stats_req_t</td>
<td>mblkp-&gt;b_rptr</td>
</tr>
<tr>
<td></td>
<td>See dl_hp_reset_stats_req_t(9S) in HP-UX 11i v3 Driver Development Reference.</td>
<td></td>
</tr>
<tr>
<td>DL_PHYS_ADDR_REQ</td>
<td>dl_phys_addr_req_t(request)</td>
<td>mblkp-&gt;b_rptr</td>
</tr>
<tr>
<td></td>
<td>See dl_phys_addr_t(9S) in HP-UX 11i v3 Driver Development Reference.</td>
<td></td>
</tr>
<tr>
<td>DL_HP_SET_IFADMIN_REQ</td>
<td>dl_hp_set_ifadmin_req_t</td>
<td>mblkp-&gt;b_rptr</td>
</tr>
<tr>
<td></td>
<td>See dl_hp_set_ifadmin_req_t(9S) in HP-UX 11i v3 Driver Development Reference.</td>
<td></td>
</tr>
<tr>
<td>DL_SET_PHYS_ADDR_REQ</td>
<td>dl_set_phys_addr_req_t</td>
<td>mblkp-&gt;b_rptr</td>
</tr>
<tr>
<td></td>
<td>See dl_set_phys_addr_req_t(9S) in HP-UX 11i v3 Driver Development Reference.</td>
<td></td>
</tr>
<tr>
<td>DL_ENABMULTI_REQ</td>
<td>dl_enabmulti_req_t</td>
<td>mblkp-&gt;b_rptr</td>
</tr>
<tr>
<td></td>
<td>See dl_enabmulti_req_t(9S) in HP-UX 11i v3 Driver Development Reference.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NOTE: To find the field holding the new ifAdmin state, see dl_hp_set_ifadmin_req_t(9S).</td>
<td></td>
</tr>
<tr>
<td>DL_DISABMULTI_REQ</td>
<td>dl_disabmulti_req_t</td>
<td>mblkp-&gt;b_rptr</td>
</tr>
<tr>
<td></td>
<td>See dl_disabmulti_req_t(9S) in HP-UX 11i v3 Driver Development Reference.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NOTE: To find the location of the new multicast address, see dl_enabmulti_req_t(9S).</td>
<td></td>
</tr>
<tr>
<td>DL_HP_HW_RESET_REQ</td>
<td>dl_hp_hw_reset_req_t</td>
<td>mblkp-&gt;b_rptr</td>
</tr>
<tr>
<td></td>
<td>See dl_hp_hw_reset_req_t(9S) in HP-UX 11i v3 Driver Development Reference.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NOTE: To find the location of the multicast address disabled on NIC, see dl_disabmulti_req_t(9S).</td>
<td></td>
</tr>
</tbody>
</table>
Table 11-3 Request Data Structures Associated with Primitives (continued)

<table>
<thead>
<tr>
<th>Primitives</th>
<th>Data Structure</th>
<th>Data Structure Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL_PROMISCOFF_REQ</td>
<td>dl_promiscoff_req_t</td>
<td>mblkp-&gt;b_rptr</td>
</tr>
<tr>
<td></td>
<td>See dl_promiscoff_req_t(9S) in HP-UX 11i v3 Driver Development Reference.</td>
<td></td>
</tr>
<tr>
<td>DL_PROMISCON_REQ</td>
<td>dl_promiscon_req_t</td>
<td>mblkp-&gt;b_rptr</td>
</tr>
<tr>
<td></td>
<td>See dl_promiscon_req_t(9S) in HP-UX 11i v3 Driver Development Reference.</td>
<td></td>
</tr>
</tbody>
</table>

**NOTE:** This primitive disables promiscuous mode on NIC.

Table 11-4 Response Data Structures of Primitives

<table>
<thead>
<tr>
<th>Primitives</th>
<th>Response Data Structure Created by HP-DLPI</th>
<th>Data Structure Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL_GET_STATISTICS_REQ</td>
<td>Ext_mib_t (See /usr/include/sys/mib.h)</td>
<td>mblkp-&gt;b_cont-&gt;b_rptr</td>
</tr>
<tr>
<td>DL_HP_GET_MIBSTATS_REQ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DL_PHYS_ADDR_REQ</td>
<td>char[]</td>
<td>mblkp-&gt;b_cont-&gt;b_rptr</td>
</tr>
</tbody>
</table>

64-Bit MIB Statistics Support

Devices that have speed 650 Mb/s or higher can support 64-bit statistics primitive. To support 64-bit statistics, the driver sets the DL_HP_DRV_64BIT_MIB feature bit. See dl_hp_drv_features_t in HP-UX 11i v3 Driver Development Reference. It also supports the DL_HP_GET_64BIT_STATS_REQ primitive. See “MIB Statistics” (page 277).

Handling an ioctl

The MBLK structure of ioctls follows the same format as primitives, see “The DL_GET_STATISTICS_REQ MBLK Chain” (page 252). The differences reside in the MBLK contents. The first MBLK contains an ioctl control block (struct iocblk) instead of a specific request type. See driver_controlp(9F). The second MBLK contains the data structure associated with the ioctl. Depending on the ioctl, the driver can write to or read from this data structure.

Figure 11-6 illustrates the MBLK chain for a DL_HP_SET_DRV_PARAM_IOCTL request. This ioctl supplies the new NIC parameters in the dl_hp_set_drv_param_ioctl_t data structure in the second MBLK. Next, the driver must set NIC to the new parameter. See “DL_HP_SET_DRV_PARAM_IOCTL ioctl” (page 258).

Figure 11-6 MBLK Chain for an ioctl
Drivers must meet the following requirements when handling an ioctl:

- Drivers are responsible for user privilege checks (the ioc_cr element of iocblk contains credential information).
- Every ioctl requires a wakeup to HP-DLPI, similar to the handling of DLPI primitives.
- HP-DLPI serializes ioctls if the DL_SERIALIZE bit is set in the dl_request field of the dl_hp_set_drv_param_ioctl_t structure associated with the IOCTL.
- Each request is associated with a data structure. For example, DL_HP_SET_DRV_PARAM_IOCTL is associated with the dl_hp_set_drv_param_ioctl_t data structure.
- Drivers must fill the ioc_count element of the ioctl MBLK if processing is successful. The ioc_count returns the size of the data portion of the MBLK.
- Drivers must allocate appropriate memory for IOCTLs, with the exception of DL_HP_SET_DRV_PARAM_IOCTL and DL_HP_GET_DRV_PARAM_IOCTL.
- Drivers can define their own IOCTLs, but it must follow the rules described in “Information on Transparent Ioctls” (page 255).

Information on Transparent Ioctls

Transparent ioctls are control requests that are not processed by HP-DLPI. HP-DLPI passes them to drivers for processing. Driver-specific ioctls are also transparent ioctls. Drivers must also service some transparent ioctls, if the driver must enable users to configure the NIC through SMH.

The transparent ioctls defined by HP required for SMH support are the following:

- **DL_HP_SET_DRV_PARAM_IOCTL** Sets a NIC parameter (MTU, speed, duplex, and so on) as specified by the relevant fields in the dl_hp_set_drv_param_ioctl_t data structure. This data structure is found in the MBLK chain. See “MBLK Chain for an Ioctl” (page 254). It is required for SMH support.
  
  For more information, see dl_hp_set_drv_param_ioctl_t(9S) in the HP-UX 11i v3 Driver Development Reference.

- **DL_HP_GET_DRV_PARAM_IOCTL** Retrieves one or more current NIC parameters. These parameters are copied to a dl_hp_get_drv_param_ioctl_t data structure in the second MBLK.

When defining transparent ioctls, drivers must use the _IOR, _IOW, and _IORW macros. In addition, drivers must ensure that the first identifier is any letter other than D, E, or S (as the first argument in _IOR, _IOW, and _IORW), because these letters are reserved by HP-DLPI.

Transparent ioctls must be handled by a driver. These ioctls also require a call back to HP-DLPI using dlpi_wakeupp. All elements of the iocblk structure must be filled by the driver, except the ioc_error and db_type of the ioctl MBLK. Drivers must set the ioc_count in the ioctl MBLK if the request is successful.

An example of a driver-specific ioctl is as follows:

```c
#define IELAN_LINK_SPEED _IOW('B',13, char)
```

The IELAN_LINK_SPEED IOCTL sets and gets data to and from card control registers.

The code fragment for ielan_ctl_req includes a detailed explanation on how the IELAN driver processes the IELAN_LINK_SPEED driver-specific ioctl.
Information on dlpi_wakeupp()

The dlpi_wakeupp function is provided by HP-DLPI. A driver retrieves the pointer to the function by calling dlpi_propp inside the driver_init call.

```c
void (*dlpi_wakeupp)(
    void *dlpi_hdlp,
    mblk_t *mblkp,
    uint_32_t status,
    void *reservedp
);
```

The dlpi_wakeupp function returns the control path processing status to the caller. Upon successful control processing, the driver control function calls dlpi_wakeupp with a status of 0. When a driver encounters an error during the execution of a request, the control function passes the appropriate error code from <sys/errno.h> to dlpi_wakeupp. If the request is invalid, or if the request cannot be handled by the driver, the status value is EINVAL.

IELAN Handling a DL_ENABMULTI_REQ Primitive

The following code fragment is from the ielan_ctl_req function of IELAN. It handles the control request to enable a multicast address (DL_ENABMULTI_REQ). The request directs the driver to enable a new multicast address for the underlying NIC.

Consider the following important points while handling this control:

1. HP-DLPI maintains a list of multicast addresses active on a NIC.
2. The driver can use dlpi_propp to obtain the list of active multicast addresses.
3. If an NIC handles only \( N \) multicast addresses, the driver must inform HP-DLPI about this limit during registration. This value is communicated to HP-DLPI during registration in the dhc_max_mcast field of the dl_hp_create_info_t data structure.

IELAN performs the following steps to enable a multicast address on a NIC.

Step 1: Obtain the List of Currently Enabled Multicast Addresses

Before passing the request to enable multicast addresses to the driver, HP-DLPI updates its own private data structure with the information.

After the IELAN driver determines that the control request is DL_ENABMULTI_REQ, it retrieves the list of currently enabled multicast addresses from HP-DLPI through the dlpi_propp interface. HP-DLPI returns the multicast address list in the datap field of the dl_hp_getinfo_t data structure as an array of multicast addresses. The list also includes the new multicast address. If the retrieval fails, then IELAN sets the status code to EINVAL.

NOTE: The driver must release the array at pMcast.dhg_datap. It must also check that the number of multicast addresses returned by HP-DLPI is greater than 0 before proceeding.

```c
int8_t *mac_addrp;
dl_hp_getinfo_t pMcast;
if( cmd == ... ) {
```
status = ...;
} else if( cmd == DL_ENABMULTI_REQ ) {
/* Get the list of multicast addresses enabled on this NIC 
* from HP-DLPI. If an error occurs, then the control 
* cannot continue for this NIC. The multicast address that 
* caused this request being called is already on the list. 
*/
pMcast.dhg_modifier = NULL;
if (ielan_iftp->dlpi_propp(ielan_iftp->dlpi_hdlp, DL_HP_OP_GET, 
   DLPI_PROP_MCAST_INFOP, &pMcast, NULL, NULL) ) {
/* Cannot get the multicast address list. This request fails. */
   status = EINVAL ;
}
... if(pMcast.dhg_count)
   pMcast.dhg_free_funcp(pMcast.dhg_datap);
}

Step 2: Update Card Hardware to Handle Multicast Addresses

After the multicast address list is obtained successfully, IELAN processes the list and sets up the card hardware with new multicast addresses.

NOTE: IELAN handles a maximum of IELAN_MAX_MULTICAST multicast addresses.

IELAN must determine that promiscuous mode is on before it can continue. It obtains this information through the dlpi_propp interface using the DL_HP_OP_GET opcode on the DLPI_PROP_PROMISC_INFO property. If it is not turned on, this indicates that the newest multicast address cannot be detected by the NIC. Therefore, the exit status is set to EINVAL. Otherwise, IELAN sets up the NIC with the multicast addresses and initiates a reset to apply the changes. This is done in the ielan_send_subseq_setup_frame routine.

... if (pMcast.dhg_count > IELAN_MAX_MULTICAST) {
   /* Get promiscuous level */
   (void)dlpi_propp(ielan_iftp->dlpi_hdlp ,DL_HP_OP_GET,
   DLPI_PROP_PROMISC_INFO, &promisc, NULL, NULL);
   if( !( promisc & (DL_HP_PHYSICAL_LEVEL | DL_HP_MULTI_LEVEL)) ) {
      pMcast.dhg_free_funcp(pMcast.dhg_datap);
      status = EINVAL ;
      break;
   }
}
...

status=ielan_send_subseq_setup_frame(ielan_iftp, &pMcast);
...

Step 3: Wake Up HP-DLPI when Processing Completes

When the driver sets up the NIC with the new multicast address, it must inform HP-DLPI about the completion of this DL_ENABMULTI_REQ control request using the dlpi_wakeupp function. In IELAN, the successful completion of this request comes after ielan_ctl_req is returned.

NOTE: The value of status and MBLK (passed into driver_controlp) to dlpi_wakeupp. In case of success, status is 0.

iftp->dlpi_wakeupp(iftp->dlhp_hdlp, mblkp, status, NULL);
If IELAN encounters errors (status is nonzero) in steps 1 or 2, then dlpi_wakeupp is called inside ielan_ctl_req.

**DL_HP_SET_DRV_PARAM_IOCTL ioctl**

Figure 11-6 shows the MBLK chain passed to a driver for a **DL_HP_SET_DRV_PARAM_IOCTL** request. This ioctl sets the speed, duplex, and MTU values. In addition, it turns on autonegotiation. The following code snippet illustrates how the IELAN driver retrieves the information from the MBLK chain and processes this ioctl.

```c
static void
ielan_ctl_req(ielan_ift_t * ielan_iftp, int32_t cmd, mblk_t *mblkp, void *rsdp) {
    .
    .
    switch (cmd) {
        case DL_HP_GET_DRV_PARAM_IOCTL: {
            .
            .
            break;
        }
        case DL_HP_SET_DRV_PARAM_IOCTL: {
            /* DL_HP_SET_DRV_PARAM_IOCTL */
            dl_hp_set_drv_param_ioctl_t *reqp;
            struct iocblk     *ioc_reqp;
            if (!mblkp || !mblkp->b_cont || !mblkp->b_cont->b_rptr) {
                status = EINVAL;
                break;
            }
            ioc_reqp=(struct iocblk*)mblkp->b_rptr;
            if(drv_priv(ioc_reqp->ioc_cr) != 0) {
                /* not superuser */
                status = EPERM;
                break;
            }
            reqp = (dl_hp_set_drv_param_ioctl_t *) ( mblkp->b_cont->b_rptr);
            switch((reqp->dl_request & ~DL_HP_SERIALIZE)) {
                /* sub_request */
                case (DL_HP_DRV_SPEED | DL_HP_DRV_DUPLEX) :
                {
                    int speed_duplex = -1 ;
                    IELAN_LOCK;
                    if( ielan_iftp->rwchan) {
                        IELAN_UNLOCK;
                        status = EBUSY;
                        break;
                    }
                    if( reqp->dl_speed == 10 )
                        speed_duplex =
                            (reqp->dl_duplex == DL_HP_FULL_DUPLEX ) ? IELAN_10FD : IELAN_10HD ;
                    else if (reqp->dl_speed == IELAN_SPEED_100 )
                        speed_duplex = (reqp->dl_duplex == DL_HP_FULL_DUPLEX ) ? IELAN_100FD : IELAN_100HD; ;
                    else {
                        IELAN_UNLOCK;
                        status = EINVAL;
                        break;
                    }
                    switch(speed_duplex) {
                        case IELAN_10HD:
                            ielan_iftp->conn_type = MII_10BASET_HD;
                            break;
                        case IELAN_AUTONEG_ON:
                            ielan_iftp->conn_type = MII_AUTOSENSE;
                            break;
                        case IELAN_100FD:
                            ielan_iftp->conn_type = MII_100BASSETX_FD;
                            break;
                        default:
                            status=EINVAL;
                            IELAN_UNLOCK;
                            break;
                    }
                }
            }
            break;
        }
    }
}
```

258 Writing a LAN Driver Under HP-DLPI
/* Do a reset to make the duplex setting * take affect. Ignore the return value. */
if( ! status ) {
    ielan_iftp->rwchan = mblkp;
    IELAN_UNLOCK;
    link_cause = DL_HP_EVENT_OP_PARAM_CHANGE;
    ielan_iftp->dlpi_eventp(ielan_iftp->dlpi_hdlp,
        DL_HP_EVENT_INTERFACE_DOWN, (void *) &link_cause, NULL);
    (void) ielan_reset(ielan_iftp);
    return;
}
break;
case DL_HP_DRV_AUTONEG:
    ...;
    ...
    return;
case DL_HP_DRV_RESET_MTU:
    case DL_HP_DRV_MTU:
        dl_hp_op_param_t op_param;
        if( (reqp->dl_request & ~DL_HP_SERIALIZE) == DL_HP_DRV_RESET_MTU )
            reqp->dl_mtu = 0;
        if ( !(reqp->dl_mtu) && (reqp->dl_mtu <= 256 )
            || (reqp->dl_mtu > ETHERMTU ) ) {
            status = EINVAL;
            break;
        }
        IELAN_LOCK;
/* A set MTU request with size 0 resets the MTU. */
        if( reqp->dl_mtu) {
            mib_ptr->ifMtu = ielan_iftp->mtu = reqp->dl_mtu;
        } else {
            mib_ptr->ifMtu = ielan_iftp->mtu = ETHERMTU;
            op_param.dop_mtu = ielan_iftp->mtu;
            IELAN_UNLOCK;
            link_cause = DL_HP_EVENT_OP_PARAM_CHANGE;
            op_param.dop_param_type = DL_HP_MTU_SET;
            if( mib_ptr->ifOper == LINK_UP)
                ielan_iftp->dlpi_eventp(ielan_iftp->dlpi_hdlp,
                    DL_HP_EVENT_INTERFACE_DOWN, (void *) &link_cause, NULL);
            (void) dlpi_propp(ielan_iftp->dlpi_hdlp, DL_HP_OP_SET,
                DLPI_PROP_OP_PARAM, &op_param, NULL, NULL);
            if( mib_ptr->ifOper == LINK_UP)
                ielan_iftp->dlpi_eventp(ielan_iftp->dlpi_hdlp,
                    DL_HP_EVENT_INTERFACE_UP, (void *) &link_cause, NULL);
            break;
        }
        default:
            status = EINVAL;
            break;
        }
        ielan_iftp->dlpi_wakeupp(ielan_iftp->dlpi_hdlp, mblkp,
            (uint32_t) status,NULL);
        return;
    }

1. If users are allowed to change a NIC's parameters with superuser privilege only, the users must use the `drv_priv` function to check the user's privilege. Users credentials are kept in the `ioc_cr` field of the `iocblk`.

2. A driver can fail a second control request if it is busy processing (`status = EBUSY`) the first control request.

3. The IELAN driver must reset the NIC so that the new speed and duplex setting can take effect. After the NIC is reset, the IELAN driver calls `dlpi_wakeupp` with the status of the speed and duplex change. The MBLK chain passed back to the `ielan_ctl_req` is passed to `dlpi_wakeupp`.

IELAN permits the `DL_HP_DRV_SPEED`, `DL_HP_DRV_DUPLEX`, `DL_HP_DRV_MTU`, `DL_HP_DRV_AUTONEG`, and `DL_HP_DRV_RESET_MTU` parameters to be set. For their definitions and for other parameters that a driver can get or set, see `dl_hp_drv_param_req_type_t(9S)` in the `HP-UX 11i v3 Driver Development Reference`. 

Driver Control Function 259
NOTE: The code for handling DL_HP_DRV_MTU (changing MTU value) also illustrates the steps to change driver parameters. HP-DLPI must be notified that the link is down before changing parameters. After the changes are made, HP-DLPI must be informed that the link is up. This process is described in “Driver to HP-DLPI Events” (page 274).

Driver-Specific I/OCTLS — IELAN_LINK_SPEED

In the following IELAN example, there is one driver-specific ioctl called IELAN_LINK_SPEED. This ioctl is an entry point to all IELAN driver-specific requests. One of these requests is IELAN_READ_REG (reading the card register value at the card address reg_offset specified by nwmgr).

These requests are handled in the ielan_private_ioctl function. To change the speed and duplex, the user must be a superuser.

NOTE: The IELAN_LINK_SPEED ioctl is not used to get or set a NIC speed setting. The IELAN NIC speed or duplex is set using the DL_HP_SET_DRV_PARAM_IOCTL control request. Use DL_HP_SET_DRV_PARAM_IOCTLs when setting driver properties because it enables the request to be sent serially to the driver.

In case of driver-specific ioctls, the driver receives an MBLK with its data portion being a struct fis data structure. The user of the ioctl is the nwmgr shared library of IELAN. See Chapter 13 (page 337).

Both the requests (in reqtype) and its associated values are passed in this data structure, and they must be integers. If the ioctl returns a value to nwmgr, the value must also be an integer.

IELAN notifies modules upstream about a link down status, similar to other ioctls and control requests that change a NIC’s parameter. After the parameter changes are complete, a link up is reported upstream. In case of changing NIC speed, IELAN must reset NIC. A link up event is reported upstream when the reset is done.

```c
static void
ielan_ctl_req(ielan_ift_t * ielan_iftp, int32_t cmd,mblk_t *mblkp, void *rsdp)
{
    intptr_t status = 0;
    switch (cmd) {
    case DL_HP_GET_DRV_PARAM_IOCTL: {
        ...
        break;
        ...
    }
    default:
    if (cmd == IELAN_LINK_SPEED ) {
        if (!mblkp || !mblkp->b_cont || !mblkp->b_cont->b_rptr) {
            status = EINVAL;
            break;
        }
        IELAN_LOCK;
        status = ielan_private_ioctl(ielan_iftp,
            (struct fis *) (mblkp->b_cont->b_rptr), mblkp);
        IELAN_UNLOCK;
        /* let reset wake up whether successful or not*/
        if( status < 0 )
            return;
        /* RESET_IN_PROGRESS do wakeup later */
    } else
        status = EINVAL;
        break;
    }
    ielan_iftp->dlpi_wakeupp(ielan_iftp->dlpi_hdlp, mblkp,
```
Interrupt Service Routine

A non-native LAN driver's Interrupt Service Routine (ISR) can handle transmit completion, receive interrupts, and any other events that cause an NIC to interrupt the driver. The driver can process inbound packets when servicing receive interrupts inside the `driver_isr` routine.

Spinlocks must be used to protect driver instance data. In addition, `driver_isr` must not sleep.

The `driver_isr` function is as follows:

```c
int driver_isr(
    void *arg
);
```

The `arg` argument is a value that a driver writer wants to pass to ISR. Usually, it is a pointer to the driver instance data. In the IELAN example, it is a pointer to the `ielan_ift_t` data structure.

ISR is set up in the `driver_init` function. For more information on how to setup an ISR in a non-native driver, see “Step 4g. Setting Up ISR for a Driver Instance” (page 248).

Rules for ISR

An ISR must perform the following actions:

- Determine a reason for the interrupt (if appropriate) and take appropriate action, such as cleanup or retry.
- Acquire spinlocks to protect transmit, control, or any types of events that can happen outside the interrupt context.
• Based on the reason, process the interrupt. If necessary, give the NIC an acknowledgement that the interrupt is or has been serviced.
• If the ISR services an interrupt, it returns 1; otherwise, it returns 0.

The driver ISR might be called even though NIC did not generate the interrupt. Therefore, before processing the interrupt, `driver_isr` must check whether the NIC generated the interrupt. If the NIC did not generate the interrupt, ISR returns 0.

Spinlocks must be used to protect driver instance data that can be accessed both in and out of the interrupt context.

An ISR executes in an interrupt context, not in a kernel thread context. Therefore, an ISR must not call `sleep` or any function that can block execution.

IELAN ISR

In the IELAN driver, ISR handles transmit completion interrupts (the NIC sends frames to the network), receive interrupts (the NIC receives frames from the network), and abnormal interrupts (for example, bus error, link fail, and transmit underflow).

In IELAN, processing of transmit and receive interrupts by the ISR is not protected by spinlock. This enables other parts of the IELAN code to run at the same time as the ISR to improve performance.

The `servicing_interrupts` flag is used as a flag to tell non-ISR IELAN functions to spin until ISR completes, especially in situations when the parallel execution of ISR and the `ielan` function causes problems. An example for this is the simultaneous execution of ISR with the driver reset operation.

The code for receiving interrupts and transmitting complete interrupts is discussed as part of inbound frame processing and outbound frame processing. See “Inbound Frame Processing” (page 264) and “Outbound Frame Processing” (page 268), respectively.

**NOTE:** The `servicing_interrupts` flag is declared as a volatile variable.

The sample code for `ielan_isr` is as follows:
static int ielan_isr(ielan_ift_t *ielan_iftp) {
    IELAN_LOCK;
    if (ielan_iftp->drv_state == IELAN_ONLINE) {
        ielan_iftp->servicing_interrupts = 1;
        IELAN_UNLOCK;
        ielan_process_interrupts(ielan_iftp);
        /* Returns with servicing_interrupts reset */
    } else {
        ielan_iftp->servicing_interrupts = 1;
        /* When the NIC is DOWN, interrupts are not processed.
        * If the NIC is being reset, or if the card
        * is currently offline, then interrupts are serviced.
        */
        return ielan_offline_isr(ielan_iftp);
        /* servicing_interrupts is reset upon
        * return from ielan_offline_isr
        */
    }
}

int ielan_process_interrupts(ielan_ift_t *ielan_iftp) {
    ubit32 csr5, dummy_csr5, csr5_ints;
    int i = 0;
    ielan_reg_read(ielan_iftp, CSR5, &csr5);
    csr5_ints = CSR5_ACK_INTS (csr5);
    .
    .
    do {
        /* Acknowledge interrupts */
        ielan_reg_write(ielan_iftp, CSR5, csr5_ints);
        /* Read the interrupt register again to flush the write */
        ielan_reg_read(ielan_iftp, CSR5, &dummy_csr5);
        /* Process any abnormal conditions first */
        if (CSR5_AIS(csr5)) {
            ielan_abnormal_isr(ielan_iftp, csr5);
        }
        /* Process receive interrupts */
    }
    /* Call receive_frame unconditionally. The following scenario
    causes a receive hang:
    - The 21143 chip will not generate an interrupt for a
    condition if the driver acknowledges that condition
    without doing the processing.
    - If this happens with the last frame in the descriptor,
    then since the ring is full, the card never
    generates another received frame interrupt and keeps
    generating only rx desc unavailable interrupts.
    - So the solution is to call receive_frame without checking
    CSR5. This will prevent the above problem.
    */
    ielan_receive_frame(ielan_iftp);
    /* Process transmit interrupts */
    ielan_transmit_complete(ielan_iftp);
    /* Any new interrupts while we were handling those? */
    ielan_reg_read(ielan_iftp, CSR5, &csr5);
    csr5_ints = CSR5_ACK_INTS (csr5);
}
while ((csr5_ints != 0) && (++i < IELAN_ISR_LIMIT));
ielan_iftp->servicing_interrupts = 0;
/* No locking required */
return SERVICED_INTERRUPT;
}

NOTE: It is possible for a NIC driver to continue to receive data until it exhausts all the available receive buffers. The driver must be written to handle this situation. IELAN handles the situation by replenishing receive buffers in the ISR.

Inbound Frame Processing

This section describes how a non-native driver handles incoming frames. A non-native driver obtains the frame data produced by the NIC, processes it, and then passes the frame upstream to HP-DLPI. HP-DLPI processes the frame data and passes it to upper layers.

A frame is passed to HP-DLPI using the driver_inboundp interface. This is an HP-DLPI function. A non-native driver obtains this interface pointer after the driver has registered with HP-DLPI. Call the dlpi_propp interface with the DLPI_PROP_INBOUNDP property, see “Step 4d: Retrieving HP-DLPI Callback Functions and NM_ID” (page 245).

Where:

```c
uint32_t driver_inboundp(
    void *dlpi_handle,
    mblk_t *mblkp,
    void *rsvd1p,
    void *rsvd2p
);
```

The parameters are as follows:

- `dlpi_handle` Handle to the DLPI instance for this driver instance. This value is obtained from dhc_dlpi_hdlp after a driver instance successfully registers with HP-DLPI. See “Step 4c: Registering a Driver Instance with HP-DLPI” (page 241).
- `mblkp` Pointer to MBLK containing the packets to be sent out. The MBLK chain can contain more than one packet. The `b_flag` in MBLK containing the start of the packet contains bits that influence processing of the packet.
- `rsvd1p` Reserved for future use.
- `rsvd2p` Reserved for future use.

Inbound Path Processing Rules

HP-DLPI requires the following from the inbound processing function of a driver:

- The out-of-packet data template created by the DLPI must be used to pass any Out-Of-Packet (OOP) information upstream. For more information on OOP, see “OOP and Transport Ioctls” (page 377).
- If OOP is present, the `b_flag` of MBLK containing OOP must set the DL_HP_OOP_PRESENT flag.
- The `b_rptr` member of a MBLK must either point to OOP (if present) or to the MAC header (Destination Address for Ethernet, Frame Control byte for FDDI, and Access Control for Token Ring).
- The `b_wptr` member of an MBLK must point to the end of the data in the MBLK.
- The beginning of the MBLK data (`b_rptr`) must be 4-byte aligned.
- When the driver is looping back an outbound packet, the driver must also pass the OOP information sent down by HP-DLPI.
• Drivers that handle packet trains must copy the cko_offset element of the cko_info_t structure from the first fragment of the packet train to the subsequent fragment being looped back. The fragments must be separated and sent as individual packets to HP-DLPI. For more information on packet trains, see Chapter 15 (page 377).

• For 802.3 packet trains, drivers must also calculate the Logical Link Control (LLC) length of each of the packet fragments and place it in the length part of LLC header before sending it to HP-DLPI.

• HP-DLPI handles automatic response for SAP 0 TEST packets.

• The inbound processing routine of a driver must check the validity of a packet. If the packet data is received with internal errors (packet too long, timeout during receive, and so on), the driver must discard it. Otherwise, the packet is passed to HP-DLPI for processing by calling the driver_inboundp function.

**NOTE:** No locks can be held when calling dlpi_inboundp.

### Updating Inbound MIB Statistics

If a driver collects MIB statistics, the inbound path processing of the driver must perform the following actions:

- When a driver finds an error in a frame during inbound processing causing it to discard the frame, it must update the following counters:
  - The number of receive errors found when processing incoming frames (ifInErrors of the mib_ifEntry data structure).
  - The number of frames discarded (ifInDiscards of the mib_ifEntry data structure).

- If a driver discards a frame for any other reason, it must update the following counter:
  - The number of frames discarded (ifInDiscards of the mib_ifEntry data structure).

- After it processes a valid packet, the inbound packet processing routine increments the following MIB counters:
  - The number of unicast frames received (ifInUcastPkts of the mib_ifEntry data structure) if a packet is a unicast packet.
  - The number of nonunicast frames received (ifInNUcastPkts of the mib_ifEntry data structure), if a packet is a broadcast or multicast packet.
  - The number of bytes received (ifInOctets of the mib_ifEntry data structure).

After the driver passes a packet using dlpi_inboundp, HP-DLPI updates the other inbound related counters.

For more information on MIB Statistics, see “MIB Statistics” (page 277).

### IELAN Driver Inbound and Receive Routine

The following code fragments are snippets from the inbound packet processing routine of IELAN. The routine is called by the ielan_isr ISR function when the NIC generates a receive interrupt. Therefore, the routine is running in an interrupt context.

The routine uses the following steps to process packets:

1. Receive the packet from the NIC and determine the validity of the packet. Update counters, as described in “Updating Inbound MIB Statistics” (page 265), except the counters for inbound unicast and nonunicast packets. These counters are initially kept in local variables, and the total counter value, kept in the driver instance, is updated in step 3.

2. Send the frame to HP-DLPI through the IELAN_PROCESS_FRAME macro. Update the counters for inbound, unicast, and nonunicast packets. Call the IELAN_PROCESS_FRAME macro to pass the frame upstream using the dlpi_inboundp function.
3. Update the driver instance MIB counters by incrementing each counter with its corresponding local variable counter.

4. Process more packets, if any. If the NIC has more incoming packets that needs to be processed, repeat steps 1 to 3. Otherwise, inbound packet processing is complete.

The detailed description of steps 1 to 4, along with snippets from the ielan_receive_frame routine, follows.

### Processing Received Frames

Information on a data packet, along with data, is retrieved from memory using a Direct Memory Access (DMA) operation.

The `tot_count` indicates the number of packets yet to be processed. At this step, the routine can encounter processing errors, or find frame errors. Therefore, the temporary counters for receive errors and inbound packet discarded are updated.

```c
static void ielan_receive_frame(ielan_ift_t * ielan_iftp)
{
    #define MAX_PKT_CT 16
    ielan_rd_t *desc_elem = (void *) 0;
    ielan_rb_t *buf_elem;
    caddr_t bufp, datap;
    ubit32 rdr_index, csr6, rdes0, frame_len, ether_frame=0,
    purge_len;
    int tot_count = IELAN_RBR_ENTRIES - ielan_iftp->rdr_empty_cnt;
    int count, exit_flag=0;
    .
    .
    while (tot_count) {
        .
        .
        int empty_count  = 0;
        int good_packets  = 0;
        int mib_ifInOctets = 0;
        int mib_ifInErrors = 0;
        int mib_ifInDiscards = 0 ;
        int mib_ifInUcastPkts = 0 ;
        int mib_ifInNUcastPkts = 0;
        /*------------------------ Part 1 -----------------------*/
        /* NO LOCK NECESSARY. Without a lock, process all received
         * packets whether good or bad (errored). Keep local counts
         * of the number of packets processed in the loop and limit
         * the processing to ielan_max_rx_frames. After the limited
         * number of received packets are processed, LOCK is then
         * necessary to process the global count corresponding to the
         * local counts, AND also to refill the descriptors with
         * buffers and posting them to the card. Local counters are
         * empty_count, good_packets, and mib_ifInOctets.
         */
        /* Loop through the RBR and corresponding RDR entries that
         * are posted to the card.
         */
        ielan_iftp->time_set =0;
        rdr_index = ielan_iftp->rdr_index;
        while (count--) {
            /* Get Receive Descriptor and associated buffers */
            desc_elem = &ielan_iftp->rdr [rdr_index];
            buf_elem = &ielan_iftp->rbr [rdr_index];
            if (buf_elem->in_use != 0) {
                .
            }
```

266 Writing a LAN Driver Under HP-DLPI
Sending the Frame to HP-DLPI

At this point, the frame obtained from the NIC is determined to be a good frame. If IELAN supports OOP, the OOP header must be prepended to the packet. Ensure that the transport layer header starts on a 4-byte aligned boundary whenever possible.

After this operation, the `ielan_receive_frame` routine updates the temporary inbound octet count, in addition to the inbound unicast and nonunicast packet counts (in the `IELAN_PROCESS_FRAME` macro).

```c
if (ielan_iftp->drv_state == IELAN_ONLINE) {
    mib_ifInOctets += frame_len;
    IELAN_PROCESS_FRAME(ielan_iftp, bufp);
}
```

/* while more packets to receive */
/*------------------------- Part 1 Completed --------------------------*/

The content of the `IELAN_PROCESS_FRAME` macro is as follows:

```c
NOTE: The mib_ifInUcastPkts and mib_ifInNUcastPkts variables are declared in ielan_receive_frame, from which this macro is called.

#define IELAN_PROCESS_FRAME(ielan_iftp, bufp) \
{ \
    u_char * addr; \
    \
    addr =((u_char *)((mblk_t *)bufp)->b_rptr); \
    if (IELAN_INDIVIDUAL_ADDRESS(addr) ) \
        mib_ifInUcastPkts++ ; \
    else \ 
        mib_ifInNUcastPkts++;

    ielan_iftp->dlpi_inboundp(ielan_iftp->dlpi_hdlp,(mblk_t *)bufp,NULL,NULL);
}
```

Updating MIB Statistics

In this step, the MIB counters of the driver instance are updated to reflect the counter values collected in steps 1 and 2 in “IELAN Driver Inbound and Receive Routine” (page 265).

```c
ielan_iftp->mib_xstats.mib_stats.ifInOctets += mib_ifInOctets;
ielan_iftp->mib_xstats.mib_stats.ifInUcastPkts += mib_ifInUcastPkts;
ielan_iftp->mib_xstats.mib_stats.ifInNUcastPkts += mib_ifInNUcastPkts;

/* ------------------------- Part 2 Completed --------------------------*/
} /* while tot_count */
```
Processing More Packets, If Any

If there are no packets for processing, inbound processing is complete. Otherwise, return to step 1 in “IELAN Driver Inbound and Receive Routine” (page 265).

Outbound Frame Processing

When HP-DLPI needs a non-native driver to send out a frame, it calls the \texttt{driver_outputp} function. This interface is established during property exchange as the \texttt{dhc_outputp} element of the \texttt{dl_hp_create_info_t} data structure in \texttt{driver_init}.

The prototype for the function is as follows:

\begin{verbatim}
uint32_t driver_output(
    void *driver_hdlp,
    mblk_t *mblkp,
    dl_hp_pkt_type_t pkt_type,
    void *rsvdp
);
\end{verbatim}

Where:

- \texttt{driver_hdlp} Value of \texttt{dhc_output_hdlp}. This value is set during the registration of this driver instance with HP-DLPI.
- \texttt{mblkp} Pointer to MBLK containing the packets to be sent out. The MBLK chain can contain one or more IP fragments. The \texttt{b_flag} of the first MBLK influences the processing of the frame.
- \texttt{pkt_type} A packet type, as defined by \texttt{dl_hp_pkt_type_t}. For more information on types, see \texttt{dl_hp_pkt_type_t} (9S) in the HP-UX 11i v3 Driver Development Reference.
- \texttt{rsvdp} Reserved for future use. Set to NULL

HP-DLPI expects one of the following return values from the \texttt{driver_outputp} function:

- 0 The frame has been successfully queued to the hardware transmit queue.
- ENXIO Incorrect device
- ENOMEM Insufficient kernel memory
- EINVAL One of the parameters has an invalid value
- ENOBFS Insufficient buffer space (in message block)
- ENOLINK Interface is not connected

Outbound Path Processing Rules

Outbound path processing must take care of the following items:

- If the NIC experiences a link down condition, the \texttt{driver_output} function does not process the passed MBLK, and \texttt{ENOLINK} must be returned to HP-DLPI.
- Multicast and broadcast frames must be looped back if the MSGNOLOOP flag is not set in \texttt{b_flag} of an MBLK. Packets that must be looped back to HP-DLPI must be cloned by calling \texttt{copymsg}. For more information on \texttt{copymsg}, see the STREAMS User's Guide.
- Self-addressed frames are returned to HP-DLPI only if the MSGNOLOOP flag is not set in \texttt{b_flag} of MBLK.
- If promiscuous mode is enabled, frames must be looped back, irrespective of the MSGNOLOOP flag value.
- If the driver must loop back frames, it must call the \texttt{dlpi_inboundp} function to do so. All MIB statistics updated during the inbound path processing must be updated for loopback frames as well. For information on updating inbound MIB statistics, see “Inbound Frame Processing” (page 264).
• The driver must not change the structure of the OOP data in packets when the packets are looped back.
• The beginning of a new fragment in a packet train is specified by setting the DL_HP_OOP_PRESENT flag in b_flag of MBLK. For more information on packet trains, see “OOP and Transport Ioctls” (page 377).
• Drivers must not modify frames received with pkt_type of DL_HP_RAW_PKT. For information on supported packet types, see dl_hp_pkt_type_t (9S).
• Drivers are not allowed to block (sleep) during this call.
• HP-DLPI handles MAC and multicast addresses in canonical format for Ethernet and FDDI drivers, and in wire format for Token Ring drivers.
• Drivers must not change the Source Address (SA) of the frame.
• HP-DLPI does not send frames to driver with the size greater than the MTU value.

**Updating Outbound MIB Statistics**

If a function rejects a frame, it must increment the MIB statistics for outbound packet discard count (ifOutDiscards). See the sample IELAN code.

If the function passes the frame to the queue of the NIC for transmission, the MIB statistics for outbound byte count (ifOutOctets), outbound packet count (ifOutUcastPkts for unicast packets, and ifOutNUcastPkts for broadcast and multicast packets), and transmission queue length (ifOutQlen) must be updated.

**IELAN Outbound Routine**

After an MBLK is received by HP-DLPI, IELAN takes care of the self-addressed frames first. Next, IELAN sets up the DMA buffers, verifies frame validity, and sends the frame to the NIC. If the frame is determined to be a loopback frame or IELAN is in promiscuous mode, the frame is also sent upstream to HP-DLPI.

In addition to sending the frame to the NIC, the driver must also handle the transmit complete interrupts. MBLK data structure, DMA buffer, and transmit descriptor cleanup activities must also be performed as part of transmit complete interrupt processing.

The following code snippet illustrates outbound packet processing for the IELAN driver. IELAN registers the ielan_resolved_output function as the outbound packet processing function with HP-DLPI.

```c
static uint32_t ielan_resolved_output(caddr_t *ifp, mblk_t *mblkp, dl_hp_pkt_type_t pkt_type,void *rsvdp)
{
    return ielan_hw_req((ielan_ift_t *)ifp, IELAN_REQ_WRITE, (caddr_t)mblkp);
}
```

**Handling Self-Addressed Frames**

The driver passes only the packet backup to HP-DLPI if the MSGNOLOOP bit is set in the b_flag field of the MBLK. In the IELAN driver, ielan_process_frames_toself handles self-addressed packets.

In this function, the IELAN_BUF_NOLOOPBACK macro determines whether the MSGNOLOOP flag is set.

If the packet is sent back to HP-DLPI, the counters for the number of inbound octets (ifInOctets) and the number of inbound unicast packets (ifInUcastPkts) are updated.

```c
static void
ielan_process_frames_toself(ielan_ift_t * ielan_iftp, caddr_t bufp)
{
    caddr_t  m_temp;
    uint32_t  frame_len=0;
    u_char  *addr;
```
if (IELAN_BUF_NOLOOPBACK(bufp)) {
    IELAN_FREE_BUF(bufp);
    return;
}
m_temp = bufp;
do {
    frame_len += (uint32_t)IELAN_DATA_LEN(m_temp);
} while (m_temp=IELAN_NEXT_PTR(m_temp));

ielan_iftp->dlpi_inboundp(ielan_iftp->dlpi_hdlp, (mblk_t *)bufp, NULL, NULL);
IELAN_R_LOCK;
ielan_iftp->mib_xstats.mib_stats.ifInOctets += frame_len;
ielan_iftp->mib_xstats.mib_stats.ifInUcastPkts++;
IELAN_R_UNLOCK;
}

Send Frame to NIC

Because self-addressed frames have been processed, the remaining frames must be multicast frames, broadcast frames, or unicast frames with different source and destination addresses. All three types of frames must be passed to NIC for transmission. In IELAN, frame transmission occurs as described in the following sections.

DMA the Frame to NIC

The ielan_slow_hw_req routine maps frame data to DMA buffers using the ielan_map_data routine. This function sets up the mapping information in the quad_ptr table. If it encounters an error during DMA mapping (indicated by invalid values in the quad_ptr table), then it discards the frame and updates the outbound discard count (ifOutDiscards). It is also updated when the frame is less than 14 bytes.

If the mapping succeeds, then the frame is DMAed to the NIC.

static uint32_t
ielan_slow_hw_req(ielan_ift_t *ielan_iftp, int32_t req_code, caddr_t bufp, u_long buf_cnt)
{
    uint32_t  frame_len=0;
    u_long    buf_len, map_buf;
    ubit32    td_cnt;
    caddr_t   datap, temp_bufp, mac_addr;
    space_t   sid;
    u_long    *quad_ptr;
    u_long    start_page, end_page;
    datap = IELAN_DATA_PTR(bufp);
    mac_addr = (caddr_t)ielan_iftp->mac_addr;
    map_buf = (caddr_t)ielan_iftp->mac_addr;
    temp_bufp = bufp;
do {
    buf_len = IELAN_DATA_LEN(temp_bufp);
    frame_len += (uint32_t)buf_len;
    if (!map_buf) {
        quad_ptr = IELAN_QUAD_PTR(temp_bufp);
        datap = IELAN_DATA_PTR(temp_bufp);
        ielan_map_data(ielan_iftp, temp_bufp);
        if (((quad_ptr[0] == (u_long)-1)) || (quad_ptr[1] == (u_long)-1)) {
            if(temp_bufp = IELAN_NEXT_PTR(temp_bufp)) {
                quad_ptr = IELAN_QUAD_PTR(temp_bufp);
            }
        }
        ielan_iftp->mib_xstats.mib_stats.ifInOctets += frame_len;
        ielan_iftp->mib_xstats.mib_stats.ifInUcastPkts++;
        IELAN_R_UNLOCK;
        return;
    }
    while (map_buf = IELAN_NEXT_PTR(map_buf)) {
        quad_ptr = IELAN_QUAD_PTR(map_buf);
        if (quad_ptr[0] == (u_long)-1) {
            if (map_buf = IELAN_NEXT_PTR(map_buf)) {
                quad_ptr = IELAN_QUAD_PTR(map_buf);
            }
        }
    }
    ielan_map_data(ielan_iftp, temp_bufp);
    } while (map_buf = IELAN_NEXT_PTR(map_buf));
    datap = IELAN_DATA_PTR(bufp);
    mac_addr = (caddr_t)ielan_iftp->mac_addr;
    map_buf = buf_cnt;
    temp_bufp = bufp;
    do {
    buf_len = IELAN_DATA_LEN(temp_bufp);
    frame_len += (uint32_t)buf_len;
    if (!map_buf) {
        quad_ptr = IELAN_QUAD_PTR(temp_bufp);
        datap = IELAN_DATA_PTR(temp_bufp);
        ielan_map_data(ielan_iftp, temp_bufp);
        if (((quad_ptr[0] == (u_long)-1)) || (quad_ptr[1] == (u_long)-1)) {
            if(temp_bufp = IELAN_NEXT_PTR(temp_bufp)) {
                quad_ptr = IELAN_QUAD_PTR(temp_bufp);
            }
        }
    }
    IELAN_R_UNLOCK;
    return;
}

270 Writing a LAN Driver Under HP-DLPI
}
  ielan_unmap(ielan_iftp->isc, bufp, ielan_iftp->DMA_handle);
  IELAN_FREE_BUF (bufp);
  ielan_iftp->mib_xstats.mib_stats.ifOutDiscards++;
  ielan_iftp->tstats.trx_errs++;
  ielan_log (ielan_iftp, ielan_emsg_map_failed, 0, 0, 0, "");
  return(ENOBUFS);
}
sid = ldsid(datap);
dma_sync(sid,(caddr_t)datap, IELAN_DATA_LEN(temp_bufp),
  IO_WRITE | IO_NO_SYNC);
}
while (temp_bufp = IELAN_NEXT_PTR(temp_bufp));
if (frame_len < 14) {
  ielan_iftp->mib_xstats.mib_stats.ifOutDiscards++;
  ielan_iftp->tstats.trx_errs++;
  ielan_unmap(ielan_iftp->isc, bufp, ielan_iftp->DMA_handle);
  IELAN_FREE_BUF(bufp);
  return(EINVAL);
}

return ielan_transmit_frame(ielan_iftp, bufp, buf_cnt,
  req_code,frame_len, quad_ptr);

Setup Transmit Descriptors

In this step, NIC must be prepared to pass the frame to the wire. Before this can be done, the
driver must set up the transmit descriptors so that NIC can transmit the data over the wire using
ielan_transmit_frame. It uses the IELAN_SETUP_TRANSMIT_Descriptors macro to set
up and DMA the transmit descriptors to NIC. After this operation, the ielan_transmit_frame
routine updates the MIB statistics for the outbound unicast packet count (ifOutUcastPkts),
outbound nonunicast packet count (ifOutNUcastPkts), and the number of packets waiting
to be transmitted (ifOutQlen).

static intptr_t
ielan_transmit_frame(ielan_ift_t *ielan_iftp,caddr_t bufp,
  u_long buf_cnt,int32_t req_code,
  uint32_t frame_len, u_long* quad_ptr)
{
  ubit32   tbfc, tbfii, tbac;
  ubit32   tbpc, tbpi;
  ubit32   tdfc, td_cnt, tdi;
  ielan_tb_t *tb;  caddr_t   addr;
  int loopback_frame_len = 0;
  if(req_code==IELAN_REQ_WRITE) {
    IELAN_FILTER_PKTS;
  }
  /* WARNING: Hold the lock until this command is completed
     * and posted for usage. Otherwise, a timing window opens,
     * which causes commands to get out of sync on MP systems.
     */
  IELAN_LOCK;
  
  if (tbfc > 0) {
/* Transmit buffer entry available to process request */
  tbfi = ielan_iftp->tbr_free_index & (IELAN_TBR_ENTRIES - 1);
  tb  = &ielan_iftp->tbr[tbfi];
if ((tbpc == 0) && (tdfc >= td_cnt)) {
  /* Transmit descriptors are available to build the descriptor list */
  IELAN_SETUP_TRANSMIT_DESCRIPTORS(ielan_iftp, bufp, tdfc, tb);
  ...
  /* Issue Transmit Poll Request */
  ielan_reg_write(ielan_iftp, CSR1, 1);
  addr = (caddr_t) IELAN_DATA_PTR(bufp);
  if(ielan_iftp->drv_state == IELAN_ONLINE) {
    if (IELAN_INDIVIDUAL_ADDRESS(addr))
      ielan_iftp->mib_xstats.mib_stats.ifOutUcastPkts++;
    else
      ielan_iftp->mib_xstats.mib_stats.ifOutNUcastPkts++;
  }
  ielan_iftp->mib_xstats.mib_stats.ifOutQlen++;
  if(req_code==IELAN_REQ_WRITE_L) {
    ielan_iftp->dma_timer = 0;
    IELAN_UNLOCK;
    return(0);
  } else if (ielan_iftp->dma_timer != 0 ) {
    IELAN_UNLOCK;
    return(0);
  } else {
    IELAN_T_LOCK;
    ielan_iftp->tbr_timed_index = tbfi;
    ielan_iftp->dma_time = IELAN_DMA_TIMEOUT;
    ielan_iftp->dma_timer = 1;
    IELAN_T_UNLOCK;
    IELAN_UNLOCK;
    return(0);
  } else {
    /* Out of transmit buffers drop request */
    ...
  }
return(ENOBUSP);
}

Handle Loopback and Promiscuous Mode Cases
In this step, IELAN must decide whether the frame must also be passed back to HP-DLPI.
If the promiscuous mode is enabled, then the frame must be cloned and passed back to HP-DLPI.
A nonunicast (broadcast or multicast) frame must also be cloned and returned to HP-DLPI when
the MSGNOLOOP flag is not set.
The OPROMISC_CLONE_LOOPBACK_PKT macro clones using copymsg and passes a frame back
to HP-DLPI.

NOTE: The cloned MBLK b_flag and the source MBLK must have the same value.

The IELAN_PROCESS_FRAME macro sends the packet to HP-DLPI and updates the inbound
unicast and inbound nonunicast packet counts. For example:
#define OPROMISC_CLONE_LOOPBACK_PKT(bufp, clone)\
{\
  clone = (caddr_t) copymsg((MBLKP)bufp);
  if (clone) {\
    (MBLKP)clone)->b_flag = ((MBLKP)bufp)->b_flag;
    loopback_frame_len = frame_len;
    IELAN_PROCESS_FRAME(ielan_iftp, clone);\
  }\
}
Handle Transmit Complete Interrupts

When NIC completes the transmission, NIC generates a transmit complete interrupt to notify the driver of the transmissions status of the frame. The ielan_transmit_complete routine handles this interrupt. The routine unmaps the DMA buffers and frees the MBLK. It also decrements the outbound queue length by the number of packets transmitted.

```c
static void
ielan_transmit_complete(ielan_ift_t * ielan_iftp)
{
    ielan_td_t *td;
    ielan_tb_t *tb;
    caddr_t   bufp;
    ubit32   tdes0;
    ubit32   tdes1;
    ubit32   tbi, tdi;
    ubit32   tbac, tdac, tdfc, tdfi;
    ubit32   td_cnt = 0;
    ubit32   tb_cnt = 0;
    dl_hp_event_link_cause_t link_cause;
    mib_ifEntry     *mib_ptr;
    IELAN_LOCK;
    /* Make sure the driver is not currently completing any * transmit commands. */
    if (ielan_iftp->tbr_complete_flag != 0) {
        IELAN_UNLOCK;
        return;
    }

    ielan_iftp->tbr_complete_flag = 1;
    /* Process any completed transactions. */
    tbac = ielan_iftp->tbr_act_cnt;
    tbi = ielan_iftp->tbr_act_index;
    IELAN_UNLOCK;
    while ((tbac-tb_cnt) > 0) {
        /* There are more active requests. */
        tb = &ielan_iftp->tbr[(tbi & (IELAN_TBR_ENTRIES - 1))];
        /* Check for the command status: FREE means that the * transmitting side has not set this to INUSE. However,
         * this occurs because during transmission, the buffer element
         * after the one being currently transmitted was concurrently
         * prepared and issued out to the card for which the active
         * count is updated and reflected in the current value.
         */
        if (tb->in_use != IELAN_CMD_FREE) {
            /* Buffer element is in use */
            .
        } /* Buffer element is in use with an active transmit request */
        else {
            /* A transmit request is not yet issued to the card. */
            break;
        }
    }
}
```
IELAN_FREE_BUF(bufp);
}
} /* The driver owns this descriptor now. */
else {
    /* Command not yet complete Issue Transmit Poll Request */
    ielan_reg_write(ielan_iftp, CSR1, 1);
    break;
}
} /* Buffer element is in use with an active transmit request. */
else {
    /* A transmit request is not yet issued to the card. */
    break;
}
} /* While there are more active requests, */
/* update MIB statistics. */
ielan_iftp->mib_xstats.mib_stats.ifOutQlen -= tb_cnt;
.
. 
IELAN_UNLOCK;

Driver to HP-DLPI Events

Each time the link state changes, HP-DLPI must be informed about the new link state using the 
dlpi_eventp interface. In addition, when values of certain properties in a driver are changed,
HP-DLPI must be notified through the same interface. These properties are hardware state,
speed, duplex, and so on. Before changing the value of a property, the driver must send HP-DLPI
a link down event. After changing the value of a property, the driver must send HP-DLPI a
link up event, only if the link is still operational after the property change.

The link state indicates whether a NIC is currently experiencing a link up or link down condition.
The hardware state (HP-DLPI DLPI_PROP_HDW_STATE property) indicates whether a card is
currently responding to driver commands.

For example, when the cable is disconnected from a NIC, the driver can still manage the NIC.
Therefore, its link state is down, and its hardware state is up. However, when a card is no longer
responding to the driver, its hardware state is declared as down, along with the link state.

Link state change notification is done using the dlpi_eventp function. A non-native driver’s
driver_init function requests a pointer to the driver event function using dlpi_propp during
driver initialization.

The dlpi_eventp function has the following prototype:

```c
void dlpi_eventp(
    void *driver_handle,
    dl_hp_event_type_t event,
    void *event_infop,
    void *rsvdp
);
```

Where:

driver_handle The value, dlpi_hdlp, obtained from HP-DLPI after a driver registers
successfullly with HP-DLPI.

event An enumerated type describing the event of type dlpi_event_t. See
dlpi_eventp(9S) in the HP-UX 11i v3 Driver Development Reference.

event_infop Associated information for an event. This parameter is used to pass the
hint for link down and link up events. See dlpi_eventp(9F) in the HP-UX
11i v3 Driver Development Reference.

274 Writing a LAN Driver Under HP-DLPI
Reserved for future use. Must be NULL.

For events such as `link_down` or `link_up`, drivers must also pass hints (cause of the link state change). For example, when driver sends a `link_down` event during the MAC address change, it stores this information in the `event_infop` parameter. For `link_up` events, drivers can use `DL_HP_EVENT_DEFAULT` if the cause of `link_up` event is unknown.

Like other calls to HP-DLPI routines, locks must not be held while calling this routine. The following are the possible events that can be reported through `dlpi_eventp`. Furthermore, driver must pass a hint to HP-DLPI for each event. See `dl_hp_event_link_cause_t` (9S) in the HP-UX 11i v3 Driver Development Reference.

**DL_HP_EVENT_INTERFACE_DOWN/UP Events**

These events must be sent whenever following events occur:

- Cable State Changes
- Link State Changes
- Hardware State Changes, including NIC Resets
- Speed Change
- Duplex Change
- Features Change
- MAC Addresses Change
- MTU Change
- `ifAdmin` Status Change

**Driver Property Changes**

When any of the following driver properties changes, either internally or through a user command, the driver must send the `link_down` event while the parameter is being updated. After the parameter has been updated, the driver must then send a `link_up` event to HP-DLPI.

- MTU
- MAC Address
- Speed
- `ifAdmin`
- Duplex Mode
- Hardware State
- Features

After a parameter is updated, HP-DLPI must be notified about the new value of the parameter, following these steps:

1. Send a `DL_HP_EVENT_INTERFACE_DOWN` event using `dlpi_eventp` to HP-DLPI with the cause that generated this event.
2. Change the parameter within the NIC.
3. Call `dlpi_propp` to set the new property value within the HP-DLPI repository.
4. Send a `DL_HP_EVENT_INTERFACE_UP` event using `dlpi_eventp` to HP-DLPI with the cause that generated this event. The cause is the same as in step 1.

**Link State Changes**

When the driver detects link state change (from up to down or down to up), HP-DLPI must be notified through the `dlpi_eventp` interface about the new link state. At this point, the driver MIB statistics `ifOper` field must be set to `LINK_UP` or `LINK_DOWN`. 
Resetting the NIC

When the driver is resetting a card either through internal action or through a user command, drivers must notify HP-DLPI that the link is down when the reset is in progress. This notification must be done as close as possible to the start of the reset process. When the reset completes successfully and the link comes up, HP-DLPI must be notified that the link is up.

Updating the Hardware State

Drivers must notify HP-DLPI about hardware state changes. For example, during a reset, the hardware state of the NIC is down. When the reset completes, the hardware state changes to up. Unlike in case of driver link state changes, drivers notify HP-DLPI about hardware state changes through the dlpi_propp function with a DL_HP_OP_SET operation on the DLPI_PROP_HDW_STATE property. The value of the DLPI_PROP_HDW_STATE property is of enumerated type dl_hp_hw_state_t. It has two possible values: DL_HP_HW_UP and DL_HP_HW_DOWN. For more information on this type, see dl_hp_hw_state_t(9S) in the HP-UX 11i v3 Driver Development Reference.

This code snippet illustrates how the IELAN driver informs HP-DLPI that the IELAN NIC is dead (HDW_STATE_DEAD has the same value as that for DL_HP_HW_DOWN). Therefore, IELAN does not have to inform HP-DLPI about the link down event.

ielan_iftp->hdw_state = HDW_STATE_DEAD;
IELAN_UNLOCK;
(void)dlpi_propp(ielan_iftp->dlpi_hdlp,DL_HP_OP_SET,
    DLPI_PROP_HDW_STATE, &ielan_iftp->hdw_state,NULL,NULL);
ielan_log(ielan_iftp, ielan_dmsg_setup_frame_failed, 0, 0, 0, "");

HP-DLPI to Driver Events

This section describes the events sent by HP-DLPI to a non-native driver. HP-DLPI uses a driver provided function, driver_event_handler, to inform the driver about the events. This function is exchanged during the driver registration with HP-DLPI (specified by dhc_eventp field of the dl_hp_create_info_t data structure).

The prototype of the driver_event_handler function is as follows:

int32_t driver_event_handler(
    void *driver_handle,
    dl_hp_event_type_t event_type,
    void *infop,
    void *rsvdp
);

Where:

driver_handle Value of dhc_event_hdlp. This value is set during the registration of this driver instance with HP-DLPI.

event_type Name of the event. See dl_hp_event_type_t(9S) in the HP-UX 11i v3 Driver Development Reference.

infop A pointer to the information specific to the event_type.

rsvdp Reserved for future use. HP-DLPI passes NULL to the driver.

HP-DLPI expects one of the following return values when functions return:

0 Success

ENXIO Incorrect device

ENOMEM Insufficient kernel memory

EINVAL One of the parameters has an invalid value

ENOBUFS Insufficient buffer space

276 Writing a LAN Driver Under HP-DLPI
Driver Event Handler Rules

This function must follow the following rules:

• HP-DLPI does not hold any spinlock when calling this function.
• Driver is not allowed to block (sleep) inside this function.
• HP-DLPI informs a non-native driver about the following event.

Table 11-5 lists the events that HP-DLPI can send to a driver.

<table>
<thead>
<tr>
<th>Event</th>
<th>Action Taken By Driver</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL_HP_DRV_EVENT_CHANGE_NMID</td>
<td>Driver changes the driver instance’s NMID value to that supplied by HP-DLPI. The new value is passed in infop.</td>
</tr>
</tbody>
</table>

Sample IELAN Driver HP-DLPI Event Handler

The following code snippet shows IELAN’s driver event handler:

```c
static int32_t
ielan_event_handler(ielan_ift_t *ielan_iftp,
                        dl_hp_drv_event_type_t event_type,
                        void *event_infop, void *rsvdp)
{
    int32_t status = 0;
    mib_ifEntry *mib_ptr;
    mib_ptr = &ielan_iftp->mib_xstats.mib_stats;
    switch(event_type) {
        case DL_HP_DRV_EVENT_CHANGE_NMID:
            IELAN_LOCK;
            ielan_iftp->nm_id = *((uint32_t *)event_infop);
            mib_ptr->ifIndex  = (int)ielan_iftp->nm_id;
            IELAN_UNLOCK;
            break;
        default:
            status= EINVAL;
    }
    return status;
}
```

MIB Statistics

This section addresses issues relating to MIB statistics for a non-native LAN driver.

The `nwmgr` command displays some statistics on driver performance. The following example displays statistics for an HP 100BT LAN interface (lan2):

```
# nwmgr --st -c lan2
*** lan2 MIB statistics:
PPA Number = 2
Description = lan2 HP A5506B PCI 10/100Base-TX 4 Port [100BASE-TX,FD,AUTO,TT=
Interface Type = 100Base-TX
MTU Size = 1500
Speed = 100 Mbps
Station Address = 0x00306E5FF045
Administration Status = UP
Operation Status = UP
Last Change = Tue Nov 14 15:03:39 2006
Inbound Octets = 22379534
Inbound Unicast Packets = 130
Inbound Non-Unicast Packets = 363845
```
Inbound Discards          = 0
Inbound Errors            = 0
Inbound Unknown Protocols = 28640
Outbound Octets           = 10058
Outbound Unicast Packets  = 52
Outbound Non-Unicast Packets = 3
Outbound Discards         = 0
Outbound Errors           = 0
Specific                   = 655367
Index                      = 3
Alignment Errors           = 0
FCS Errors                 = 0
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 preceding statistics such as Ethernet-like Statistics Group, Inbound Unicast Packets (number of unicast packets received), or Outbound Queue Length (number of packets waiting to be transmitted) are updated by the driver. To have nwmgr report these statistics for a non-native driver, you must incorporate the MIB statistics data structure into driver instance data.

The MIB statistics data structure is defined in the /usr/include/sys/mib.h header file.

The nwmgr output in this example is generated from the fields in the mib_ifEntry and mib_Dot3StatsEntry data structures. The latter structure maintains statistics for the data grouped under Ethernet-like Statistics Group. The first data structure stores the statistics that are applicable to all NICs.

The prototype of the mib_ifEntry data structure is as follows:

typedef struct {
    int   ifIndex;
    char  ifDescr[64];
    int   ifType;
    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 ifOutUcastPkts;
    counter ifOutNUcastPkts;
    counter ifOutDiscards;
    counter ifOutErrors;
    gauge ifOutQlen;
    int   ifSpecific;
}

Table 11-6 lists the field names and descriptions for the mib_ifEntry structure.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ifIndex</td>
<td>Network Management ID of the LAN interface.</td>
</tr>
<tr>
<td>ifDescr</td>
<td>A brief description of the LAN interface.</td>
</tr>
<tr>
<td>Field Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ifType</td>
<td>LAN type of the interface.</td>
</tr>
<tr>
<td>ifMTU</td>
<td>Maximum Transport Unit.</td>
</tr>
<tr>
<td>ifSpeed</td>
<td>Current speed setting of the card.</td>
</tr>
<tr>
<td>ifPhysAddress</td>
<td>MAC address information.</td>
</tr>
<tr>
<td>ifAdmin</td>
<td>Administrative status of the card.</td>
</tr>
<tr>
<td>ifOper</td>
<td>Operational status of the card. For an example of use, see “Driver to HP-DLPI Events” (page 274).</td>
</tr>
<tr>
<td>ifLastChange</td>
<td>Last time (in ticks) the ifOper field value was changed.</td>
</tr>
<tr>
<td>ifInUcastPkts</td>
<td>Number of unicast packets (good) received.</td>
</tr>
<tr>
<td>ifInNUcastPkts</td>
<td>Number of broadcast and multicast packets (good) received or looped back.</td>
</tr>
<tr>
<td>ifInOctets</td>
<td>Number of octets received from good packets.</td>
</tr>
<tr>
<td>ifInDiscards</td>
<td>Number of packets discarded.</td>
</tr>
<tr>
<td>ifInErrors</td>
<td>Number of errors encountered on the receive path.</td>
</tr>
<tr>
<td>ifOutOctets</td>
<td>Number of octets transmitted over the wire.</td>
</tr>
<tr>
<td>ifOutNUcastPkts</td>
<td>Number of broadcast or multicast packets transmitted over the wire.</td>
</tr>
<tr>
<td>ifOutUcastPkts</td>
<td>Number of unicast packets transmitted over the wire.</td>
</tr>
<tr>
<td>ifOutDiscards</td>
<td>Number of packets discarded during outbound packet processing.</td>
</tr>
<tr>
<td>ifOutErrors</td>
<td>Number of transmit errors.</td>
</tr>
<tr>
<td>ifOutQlen</td>
<td>Number of packets waiting for the NIC to be sent over the wire.</td>
</tr>
<tr>
<td>ifSpecific</td>
<td>Driver-specific counter or flag.</td>
</tr>
</tbody>
</table>

**NOTE:** For detailed explanation of the fields in Table 11-6, see RFC 2863.

The Ethernet-specific statistics are available in the mib_Dot3StatsEntry data structure. The prototype of the mib_Dot3StatsEntry data structure is as follows:

```c
typedef struct {
    int   dot3StatsIndex;
    counter dot3StatsAlignmentErrors;
    counter dot3StatsFCSErrors;
    counter dot3StatsSingleCollisionFrames;
    counter dot3StatsMultipleCollisionFrames;
    counter dot3StatsSQETestErrors;
    counter dot3StatsDeferredTransmissions;
    counter dot3StatsLateCollisions;
    counter dot3StatsExcessiveCollisions;
    counter dot3StatsInternalMacTransmitErrors;
    counter dot3StatsCarrierSenseErrors;
    counter dot3StatsFrameTooLongs;
    counter dot3StatsInternalMacReceiveErrors;
    counter dot3StatsExcessCollisions;
    counter dot3StatsControlFieldErrors;
    counter dot3StatsMulticastsAccepted;
} mib_Dot3StatsEntry;
```

The 64-bit MIB statistics can be supported by drivers with speeds greater than or equal to 650 Mb/s. Following is sample output:
# nwmgr --st -c lan0

*** lan0 64 bit MIB statistics:

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Name</td>
<td>lan0</td>
</tr>
<tr>
<td>PPA Number</td>
<td>0</td>
</tr>
<tr>
<td>Description</td>
<td>lan0 HP PCI Core I/O 1000Base-T Release B.11.31.0</td>
</tr>
<tr>
<td>Interface Type</td>
<td>1000Base-T</td>
</tr>
<tr>
<td>MTU Size</td>
<td>1500</td>
</tr>
<tr>
<td>Speed</td>
<td>100 Mbps</td>
</tr>
<tr>
<td>Station Address</td>
<td>0x00306E4B08F9</td>
</tr>
<tr>
<td>Administration Status</td>
<td>UP</td>
</tr>
<tr>
<td>Operation Status</td>
<td>UP</td>
</tr>
<tr>
<td>Last Change</td>
<td>Tue Nov 14 15:03:34 2006</td>
</tr>
<tr>
<td>Inbound Octets</td>
<td>201712110</td>
</tr>
<tr>
<td>Inbound Unicast Packets</td>
<td>24675</td>
</tr>
<tr>
<td>Inbound Multicast Packets</td>
<td>4458</td>
</tr>
<tr>
<td>Inbound Broadcast Packets</td>
<td>3103248</td>
</tr>
<tr>
<td>Inbound Discards</td>
<td>1</td>
</tr>
<tr>
<td>Inbound Errors</td>
<td>0</td>
</tr>
<tr>
<td>Inbound Unknown Protocols</td>
<td>2240</td>
</tr>
<tr>
<td>Outbound Octets</td>
<td>3065005</td>
</tr>
<tr>
<td>Outbound Unicast Packets</td>
<td>23850</td>
</tr>
<tr>
<td>Outbound Multicast Packets</td>
<td>0</td>
</tr>
<tr>
<td>Outbound Broadcast Packets</td>
<td>350</td>
</tr>
<tr>
<td>Outbound Discards</td>
<td>0</td>
</tr>
<tr>
<td>Outbound Errors</td>
<td>0</td>
</tr>
<tr>
<td>Counter Discontinuity Time</td>
<td>Tue Nov 14 15:03:34 2006</td>
</tr>
<tr>
<td>Physical Promiscuous Mode</td>
<td>FALSE</td>
</tr>
<tr>
<td>Physical Connector Present</td>
<td>TRUE</td>
</tr>
<tr>
<td>Interface Alias</td>
<td></td>
</tr>
<tr>
<td>Link Up/Down Trap Enable</td>
<td>Enabled</td>
</tr>
<tr>
<td>Index</td>
<td>1</td>
</tr>
<tr>
<td>Alignment Errors</td>
<td>0</td>
</tr>
<tr>
<td>FCS Errors</td>
<td>0</td>
</tr>
<tr>
<td>Internal MAC Transmit Errors</td>
<td>0</td>
</tr>
<tr>
<td>Frame Too Long Errors</td>
<td>0</td>
</tr>
<tr>
<td>Internal MAC Receive Errors</td>
<td>0</td>
</tr>
<tr>
<td>Symbol Errors</td>
<td>0</td>
</tr>
<tr>
<td>Single Collision Frames</td>
<td>0</td>
</tr>
<tr>
<td>Multiple Collision Frames</td>
<td>0</td>
</tr>
<tr>
<td>SQE Test Errors</td>
<td>0</td>
</tr>
<tr>
<td>Deferred Transmissions</td>
<td>0</td>
</tr>
<tr>
<td>Late Collisions</td>
<td>0</td>
</tr>
<tr>
<td>Excessive Collisions</td>
<td>0</td>
</tr>
<tr>
<td>Carrier Sense Errors</td>
<td>0</td>
</tr>
<tr>
<td>Control Field Errors</td>
<td>0</td>
</tr>
<tr>
<td>Multicasts Accepted</td>
<td>0</td>
</tr>
<tr>
<td>Duplex Status</td>
<td>Full Duplex</td>
</tr>
<tr>
<td>Rate Control Ability</td>
<td>FALSE</td>
</tr>
<tr>
<td>Rate Control Status</td>
<td>TRUE</td>
</tr>
<tr>
<td>Collision Count</td>
<td>0</td>
</tr>
<tr>
<td>Collision Frequency</td>
<td>0</td>
</tr>
</tbody>
</table>

To report these statistics using nwmgr, the driver must include the 64-bit MIB structure into its driver instance data. The MIB structure is defined in /usr/include/sys/mib.h. When DLPI sends the mblk with the 64-bit statistics primitive, it preallocates the memory required to fill the statistics. After filling the statistics, driver must move the w_ptr of the data mblk to point to the end of data. Drivers that do not support link-specific statistics must set the values to zero and set the w_ptr to end of the structure Ext_64bit_mib_t.

Following are code snippets for 64-bit support by the driver.

**Driver Instance Data:**

```c
typedef struct ielan_if_t{
    uint32_t instance_num;
    .
    .
    Ext_64bit_mib_t mibp;
    .
```

280  Writing a LAN Driver Under HP-DLPI
Driver registration with DLPI:

\texttt{ielan_iftp->features = (DL_HP_DRV_HP_DLS|DL_HP_DRV_LAN_CLASS|DL_HP_DRV_PHYSICAL|DL_HP_DRV_64BIT_MIB);}

Control request processing by driver:

\begin{verbatim}
switch (cmd) {
  .
  .
  .
  case DL_HP_GET_64BIT_STATS_REQ:
    pMcast.dhg_modifier = NULL;
    if( dlpi_prop(ielan_iftp->dlpi_hdlp, DL_HP_OP_GET, DLPI_PROP_MCAST_INFOPE, &pMcast, NULL, NULL) ) {
      msg_printf("ielan_ctl_req: dlpi_prop() get DLPI multicast list failed.
      status = EINVAL;
      break;
    }
    ielan_ext_mibstats(ielan_iftp, mblkp->b_cont, (size_t) 0, pMcast.dhg_count);
    mblkp->b_cont->b_wptr = mblkp->b_cont->b_rptr + sizeof(Ext_64bit_mib_t);
    if(pMcast.dhg_count)
      pMcast.dhg_free_funcp(pMcast.dhgdatap);
    break;
  }
  /* end of case */
  case DL_HP_RESET_STATS_REQ:
    ielan_ext_cleanmib(ielan_iftp, NULL);
    break;
  .
  .
  .
}

static void
ielan_ext_mibstats(ielan_ift_t * ielan_iftp, mblk_t * datap, size_t offset, uint32_t count){
  /* put mibstats in data pointer */
  ielan_get_mibstats(ielan_iftp, datap, (size_t)0);
  /* put statistics in data pointer + sizeof(mibp) */
  ielan_ext_dot3stats(ielan_iftp, datap, sizeof(mib_ifEntry) + sizeof(mib_ifXEntry), count);
  /* put collisions in data pointer + sizeof(mibp + statistics) */
  ielan_ext_dot3coll(ielan_iftp, datap, sizeof(mib_ifEntry) + sizeof(mib_ifXEntry) +
    sizeof(mib_Dot3XStatsEntry));
}

static void
ielan_get_mibstats(ielan_ift_t * ielan_iftp, mblk_t * datap, size_t offset){
  Ext_64bit_mib_t *mib_ptr, *rp.ptr;
  mib_ifEntry *rmib_ptr;
  mib_ifXEntry *rmibX_ptr;
  mib_ptr = &ielan_iftp->mibp;
  uint32_t promisc;
  char *szDuplex, *szConn, *szSpeed, *szType;
  char *mibif_descr = "lan%d %s [ %s,%s,%s,TT=%d \n"
  rp.ptr = (Ext_64bit_mib_t *)(datap->b_rptr);
  rp.ptr = &rp.ptr->mibp;
  rmibX_ptr = &rp.ptr->mibXif;
  if (ielan_iftp->conn_type == MII_AUTOSENSE)
    szConn = "AUTO";
  else
    szConn = "MANUAL";
  if (ielan_iftp->speed == IELAN_SPEED_100)
    szSpeed="100";
  else
    szSpeed="10";
  if (ielan_iftp->duplex) szDuplex = "FD";
  else
    szDuplex = "HD";
  if(ielan_iftp->port == IELAN_BNC_PORT) {
    szType="BASE2";
    szConn="";
  }
  else if(ielan_iftp->port == IELAN_RJ45_PORT) {
    if(ielan_iftp->speed == IELAN_SPEED_10)
      szType="BASE-T";
    else
      szType="BASE-TX";
  }
  else {
    szType="NO LINK";
    szSpeed="";
    szDuplex="";
  }
  sprintf(mib_ptr->mib_if.ifDescr, 64, mibif_descr,
\end{verbatim}
Inbound Processing Statistics

During inbound processing, the driver must update the number of inbound broadcast or multicast packets (ifInNUcastPkts), inbound unicast packets (ifInUcastPkts), and number of octets received (ifInOctets).

The three values are updated only for good packets received. In addition, if there is an error in a packet received by an NIC, and if the driver encounters errors during the processing of a packet,
the driver must increment the number of inbound packets discarded \((ifInDiscards)\) and inbound errors \((ifInErrors)\).

In addition, during an ISR, a routine that handles transmit completion must subtract the number of packets transmitted successfully from the \(ifOutQlen\) count (the number of packets waiting to be sent over the wire).

For examples of the usage of MIB statistics during inbound processing, see “Inbound Frame Processing” (page 264).

### Outbound Processing Statistics

As in the inbound processing routine, a driver's outbound processing routines must update counters for the number of nonunicast packets sent \((ifOutNUcastPkts)\), the number of unicast packets sent \((ifOutUcastPkts)\), and the number of octets \((ifOutOctets)\). Those three statistics are counted for good packets only. If transmit routines find errors in a packet or encounters errors during processing, the number of outbound packets are discarded \((ifOutDiscards)\) and the outbound error count \((ifOutErrors)\) must be updated.

In addition, each time a packet is successfully passed to a NIC to be processed, \(ifOutQlen\) (number of packets waiting to be sent over the wire) must be updated.

For examples of the usage of MIB statistics during outbound processing, see “Outbound Frame Processing” (page 268).

### DL_HP_GET_MIBSTATS_REQ Control Request

HP-DLPI requests the driver to return the MIB statistics collected. The driver copies the MIB statistics to an MBLK.

The IELAN driver maintains three MIB statistics areas. In addition to the statistics in the \(mib\_ifEntry\) data structure and the \(mib\_Dot3StatsEntry\), it keeps \(mib\_Dot3CollEntry\).

```c
typedef struct {
    int dot3CollIndex;
    int dot3CollCount;
    counter dot3CollFrequencies;
} mib_Dot3CollEntry;
```

The driver copies the statistics to MBLK in the following order:

1. \(mib\_ifEntry\)
2. \(mib\_Dot3StatsEntry\)
3. \(mib\_Dot3CollEntry\)

The following routine copies the MIB statistics to an MBLK at \(datap\). The \(ielan\_get\_mibstats\) routine copies the statistics in \(mib\_ifEntry\) to \(datap\rightarrow rptr\). After this operation, \(ielan\_ext\_dot3\_stats\) copies the \(mib\_Dot3StatsEntry\) data structure to \(datap\rightarrow r\_ptr + sizeof(mib\_IfEntry)\).

```c
static void
ielan_ext_mibstats(ielan_ift_t * ielan_iftp, mblk_t * datap, size_t offset, uint32_t count) {
    /* put mibstats in data pointer */
    ielan_get_mibstats(ielan_iftp, datap, (size_t)0);

    /* put statistics in data pointer + sizeof(mibstats) */
    ielan_ext_dot3stats(ielan_iftp, datap, sizeof(mib_ifEntry), count);

    /* put collisions in data pointer + sizeof(mibstats + statistics) */
    ielan_ext_dot3coll(ielan_iftp, datap,
```
static void ielan_get_mibstats(ielan_ift_t * ielan_iftp, mblk_t * datap, size_t offset)
{
    mib_ifEntry *mib_ptr = &ielan_iftp->mib_xstats.mib_stats;
    caddr_t rptr;

    rptr = (caddr_t)(datap->b_rptr);

    bcopy((void *) mib_ptr, (void *) (rptr + offset), sizeof(mib_ifEntry));
    datap->b_wptr += sizeof(mib_ifEntry);
}

static void ielan_ext_dot3stats(ielan_ift_t * ielan_iftp, mblk_t * datap, size_t offset,
        uint32_t count)
{
    caddr_t rptr;
    rptr = (caddr_t)(datap->b_rptr);

    bcopy((caddr_t)(&ielan_iftp->dot3_ext_stats), rptr + offset,
            sizeof(mib_Dot3StatsEntry));
    datap->b_wptr += sizeof(mib_Dot3StatsEntry);
}

DL_HP_RESET_STATS_REQ Control Request

This control request comes from nwmgr. It directs the driver to set the statistics to zero on the NIC. The driver sets the counters to zero (not the entire mib_ifEntry) for the maintained statistics.

The following code snippet is an example of the ielan_ext_clearmib function, which executes when HP-DLPI passes a DL_HP_RESET_STATS_REQ control request to IELAN. This request clears only the traffic related counters.

static void ielan_ext_clearmib(ielan_ift_t * ielan_iftp, caddr_t datap)
{
    ielan_reset_stats(ielan_iftp);
    /* zero out the if_entry */
    bzero((char *)(&ielan_iftp->mib_xstats.mib_stats.ifInOctets), (size_t)44);
    bzero((char *)(&ielan_iftp->mib_xstats.if_DInDiscards), (size_t)48);

    /* zero out the 802.3 stats, but not the index */
    bzero((char *)(&ielan_iftp->dot3_ext_stats.dot3StatsAlignmentErrors),
            sizeof(mib_Dot3StatsEntry)-4));
    /* ielan_iftp->dot3_ext_coll not used */
    ielan_iftp->dlpi_eventp(ielan_iftp->dlpi_hdlp,
        DL_HP_EVENT_CLEAR_STATS,
        NULL, NULL);
NOTE: The IELAN mib_ifEntry data structure is in mib_xstats.mib_stats. 11 counters of type unsigned int must be cleared. Hence, the number 44 (11 counters X 4 bytes per unsigned int) is present in the first memset call.

Online Addition, Replacement, and Deletion of Cards and HP-DLPI

This section describes what a non-native driver under HP-DLPI must do if it supports Online Addition, Replacement, and Deletion (OLARD) of the NIC cards.

For information on implementing OLARD in a device driver, see Chapter 20 (page 455).

To enable OLARD support, a driver must register the OLARD capability with WSIO. During driver_attach, in addition to registering the WSIO event handler function, the driver must inform WSIO that it handles the WSIO_EVENT_SUSPEND, WSIO_EVENT_RESUME, and WSIO_EVENT_REMOVE events by using the wsio_reg_drv_capability_mask function. The ielan_attach code in “Step 3: Calling driver_attach()” (page 238) illustrates how the WSIO event handler, capabilities, and timeout values are registered with WSIO.

Online Addition

An online addition of an I/O card does not generate any WSIO events for a driver.

To support online addition, WSIO calls driver_attach and driver_init to set up the card. Ensure that both these functions run correctly after the system boots. Some resources available during boot time might not be available for long after the kernel is running. You must handle memory allocation errors. Furthermore, a driver in a running system cannot wait indefinitely for memory to become available. Finally, both these routines must be MP-safe.

Any global resource shared by all driver instances must be constructed during driver_install, not during driver_attach or driver_init.

During the process of initializing a new card, HP-DLPI must be informed about the card’s current hardware and administrative states.

Online Replacement

When a user directs HP-UX to replace a card, WSIO calls the driver’s WSIO event handler driver_wsio_event_handler with the WSIO_EVENT_SUSPEND event to suspend card operation. The driver’s WSIO_EVENT_SUSPEND event handler shuts down the card. Later, it invokes a callback function supplied by WSIO to inform WSIO that the suspend operation is complete. When the card is inserted into the PCI slot, WSIO calls the driver_wsio_event_handler function with a WSIO_EVENT_RESUME operation. If the resume operation times out, WSIO assumes the card is in the SUSPENDED state.

In response to a WSIO_EVENT_SUSPEND event, the NIC driver must do the following:

1. Notify HP-DLPI that its administrative status (DLPI_PROP_IFADMIN) is down.
2. Set the ifadmin status in the mib_ifEntry data structure to DOWN, if the driver collects MIB statistics.
3. Wait for the ISR to finish its current execution on the card. Ensure that the ISR does not service further interrupts coming from the card. Disable any interrupt generation on the card.
4. Cancel active timers, if any, started by the current driver instance.
5. Finally, notify WSIO through a WSIO supplied callback function. The callback passes to the driver when WSIO calls driver_wsio_event_handler with the WSIO_EVENT_SUSPEND event.

All activities on the card to be replaced stop. The suspend action must not fail.
The relevant parts of the driver suspend operation are included in the following code example. The \texttt{ielan\_suspend} function is called from \texttt{ielan\_wsio\_event\_handler} in response to a \texttt{WSIO\_EVENT\_SUSPEND} event. When the function is called, \texttt{ielan\_wsio\_event\_handler} has already saved a pointer to the callback function to \texttt{ielan\_iftp->wsioeventp->completecb}.

The \texttt{ielan\_halt\_card} function performs actions that halt all the driver activities on the card. For example:

```c
static void
ielan_suspend(ielan_ift_t *ielan_iftp)
{
    mib_ifEntry *mib_ptr = &ielan_iftp->mib_xstats.mib_stats;
    dl_hp_ifadmin_state_t ifAdmin;
    IELAN_LOCK;

    mib_ptr->ifAdmin = LINK_DOWN;
    IELAN_UNLOCK;
    (void)dlpi_propp(ielan_iftp->dlpi_hdlp,DL_HP_OP_SET,
                     DLPI_PROP_IFADMIN ,&ifAdmin, NULL, NULL);
    ielan_halt_card(ielan_iftp);
    ielan_iftp->wsioeventp->complete_cb(ielan_iftp->isc,
                                         ielan_iftp->wsioeventp->event_id,(void *)WSIO_OK);
}
```

In response to a \texttt{WSIO\_EVENT\_REPLACE} event, the driver must do the following:

1. Verify that the new card is same as the old card.
2. Reinitialize the card.
3. Set up the card to the state before \texttt{WSIO\_EVENT\_SUSPEND} event was processed. As part of this step, HP-DLPI must be notified (using \texttt{dlpi\_eventp}) that the operation status of the card is \texttt{DOWN} (through \texttt{dlpi\_eventp}), with the \texttt{DL\_HP\_EVENT\_HA} hint.
4. Put the card back to its state prior to replacement. HP-DLPI is informed through \texttt{dlpi\_eventp} of the \texttt{INTERFACE\_UP} event. If the link is down, nothing is sent to HP-DLPI. Finally, HP-DLPI should be informed that the administrative status has changed from \texttt{DOWN} to \texttt{UP}.
5. Call the WSIO supplied callback function to inform WSIO that the resume operation is complete.

After a card is replaced, WSIO sends a \texttt{WSIO\_EVENT\_RESUME} event through \texttt{driver\_wsio\_event\_handler}.

In the IELAN code, the \texttt{WSIO\_EVENT\_RESUME} event handler employs the reset routine (\texttt{ielan\_reset}) to reinitialize the card and bring the card back to its original state.

Before the resume operation can continue, you must verify that the new card is the same as the suspended old card.

**Online Deletion**

When a user directs HP-UX to remove a card, WSIO calls the driver's WSIO event handler \texttt{driver\_wsio\_event\_handler} with the \texttt{WSIO\_EVENT\_SUSPEND} event (to suspend card operation). The driver's \texttt{WSIO\_EVENT\_SUSPEND} event handler shuts down the card. WSIO then calls the driver's WSIO event handler \texttt{driver\_wsio\_event\_handler} with the \texttt{WSIO\_EVENT\_REMOVE} event. Later, it invokes a callback function (supplied by WSIO) to inform WSIO that the remove operation is complete.
NIC Configuration and NIC Tool Support

During boot, NICs can be initialized using a script called `driver` (called the driver startup script) installed in the `/sbin/init.d/` directory. A symbolic link `SXXXdriver` (XXX = a 3-digit number less than or equal to 337) in `/sbin/rc2.d` (directory where all symbolic links to driver startup scripts that must be run in Run Level 2) points to the script. The script uses `nwmgr` to configure the NIC. To have `nwmgr` change parameter for the driver, a shared library that makes ioctl calls into the driver is required. To write this shared library, see “Supporting the HP SMH NIC Tool in LAN Drivers” (page 387).

The following section describes the driver startup script and the configuration file. Next, the snippets from the IELAN driver startup script and configuration file are included to illustrate how to write a driver startup script and configuration file.

Driver Startup Script and Configuration File

The driver startup script is called from HP-UX boot routines. To configure NICs, the script locates the `/etc/rc.config.d/hpdriverconf` configuration file, which defines the parameters of NICs. The driver startup script accepts one of the following arguments:

```
start_msg     Prints the message, "Configure Driver Interfaces," then exits.
stop_msg      Prints the message, "Unconfigure Driver Interfaces," then exits.
start         Configures NICs.
stop          (Optional) Unconfigure NICs.
```

Exit Values Expected by HP-UX

The driver startup script returns the following errors to HP-UX:

2 (WARNING) Returned when the driver startup script encounters a warning situation.

1 (ERROR) Returned in the following situations:
- When the driver startup script encounters an error during configuration of a NIC, regardless of the successful configuration of other NICs.
- When commands used by the driver startup script are not found.
- When the configuration file is missing.

0 Returned when no errors are encountered during configuration of NICs.

Driver Startup Script Guidelines

When writing a driver startup script, follow these guidelines:

1. Use only the HP-UX commands from the `/sbin`, `/usr/sbin`, and `/usr/bin` directories. Set the PATH to `/sbin:/usr/sbin:/usr/bin`.

2. Verify that driver interfaces are present in the system. If there are no such interfaces, report that no driver supported interfaces are present. Then exit with 0.

3. Verify that the `/etc/rc.config.d/driverconf` configuration file is present. If the file is absent, exit with an ERROR.

4. Verify that the commands used by the driver startup script are present. If they are absent, exit with an ERROR.

5. Locate the configuration file. This reads all the parameters of the NICs to be configured.

6. Verify that NICs specified in the configuration file are present. If they are found, configure the NICs. If the NICs are absent, continue to process other NICs specified in the configuration file.

7. The exit code is 1 (ERROR) if the configuration of a NIC fails, regardless of the successful configuration of other NICs.

8. The exit code is 0 if all NICs are successfully configured.
The configuration for a NIC in an IELAN configuration file is shown in the following example. The IELAN driver enables the MAC address (HP_IELAN_STATION_ADDRESS), speed and duplex (HP_IELAN_SPEED), and MTU (HP_IELAN_MTU) to be configured. The instance of the configured NIC is specified by HP_IELAN_INTERFACE_NAME. The first configured NIC is at index 0, the next one is at index 1, and so on.

### IELAN Configuration File

```
# hpbase100conf: contains configuration values for HP PCI 100BASE-T
# interfaces
#
# HP_IELAN_INTERFACE_NAME  Name of interface (lan0, lan1...)  
# HP_IELAN_STATION_ADDRESS Station address of interface 
# HP_IELAN_SPEED           Speed and duplex mode 
#                          Can be one of : 10HD, 10FD, 100HD, 100FD and 
#                          AUTO_ON. 
# HP_IELAN_MTU             Maximum Transmission Unit (MTU) 
#                          Integer value between 257 and 1500, inclusive. 
#
# The interface name, major number, card instance and PPA can be 
# obtained from the lanscan(1m) command. 
# The station address and duplex are set through the lanadmin(1m) command. 
#
# HP_IELAN_INTERFACE_NAME[0]=
# HP_IELAN_STATION_ADDRESS[0]= 
# HP_IELAN_SPEED[0]=
# HP_IELAN_MTU[0]=
# HP_IELAN_INIT_ARGS="HP_IELAN_STATION_ADDRESS HP_IELAN_SPEED HP_IELAN_MTU"

# End of hpielanconf configuration file
```

### IELAN Startup Script

The following example shows the IELAN driver startup script. This script does not unconfigure NICs because it returns 0 when it encounters the stop argument.

The first step is to process the incoming argument. When the argument is start, verify that IELAN interfaces are present by running the check_install function. The function exits with a 2 (WARNING) if the IELAN interfaces are absent.

For example:

```
#!/sbin/sh
#
# /etc/rc.config.d/hpielanconf defines the configuration parameters:
#
# HP_IELAN_INTERFACE_NAME  Name of interface (lan0, lan1...) 
# HP_IELAN_STATION_ADDRESS Station address of interface 
# HP_IELAN_SPEED           Speed and mode 
# HP_IELAN_MTU             MTU Value 
#
# Note : this is an algorithmic shell only..... 
# status on exit 0 = okay 
#   1 = error 
#   2 = warning (N/A)
export PATH=/sbin:/usr/sbin:/usr/bin
```

288  Writing a LAN Driver Under HP-DLPI
check_install() {
    # This routine checks if any PCI 100BT card is installed.
    # If not, the init script exits with a warning (2).
    # This appears as a N/A on the boot screen.

    IELAN=0
    IELAN_DRIVER=ielan
    for LINE in `ioscan -kFC lan | grep $IELAN_DRIVER | sed 's/ /~/g'` ; do
        DRIVER_NAME='echo $LINE | awk -F: '{print $10}`
        if [ "$DRIVER_NAME" = $IELAN_DRIVER ]; then
            IELAN=1
            break
        fi
    done
    if [ $IELAN -eq 0 ] ; then
        echo "No 100BASE-T device configured in system (ielan)"
        exit 0
    fi
}

The next step verifies that the configuration file and the commands used by this script are present. If any of the files or commands are not found, the script exits with a 1 (ERROR). A driver startup script can use the commands found in /sbin, /usr/bin, and /usr/sbin directories only. If the IELAN configuration file (/etc/rc.config.d/ielanconf) is present, locate it. For example:

```
Nieelanconf
Nieelanconf

check_install
```

# Get actual configuration

```
# Get actual configuration

if [ -f "$HPIELAN" ]; then
    . $HPIELAN        # display any errors
    if [ $? -ne 0 ]; then
        # NB: this is not working as expected!
        echo "ERROR: incorrect data in file '$HPIELAN'" >&2
        exit 1
    fi
```
After the configuration file is located, the driver startup script processes each NIC. Before it starts configuring a NIC, it verifies that the NIC is present. If the PPA of the NIC is absent in the system, the script continues to process the next NIC. If the configuration of a NIC encounters an error, the exit code is set to 1 (ERROR). If all the NICs in the configuration file are configured successfully, the script exits with the code 0. In the IELAN script, the exit code is stored in `rval`.

```bash
rval=0

# Check for lanadmin, lanscan commands

HPLANADMIN=/usr/sbin/lanadmin
if [ ! -x "$HPLANADMIN" ]; then
    echo "ERROR: missing file '$HPLANADMIN'" >&2
    exit 1
fi

HPLANSCAN=/usr/sbin/lanscan
if [ ! -x "$HPLANSCAN" ]; then
    echo "ERROR: missing file '$HPLANSCAN'" >&2
    exit 1
fi
```

PROD_NAME=ielan
n=${#HP_IELAN_INTERFACE_NAME[*]}
i=0
while ((n>0)); do
    CHECK=${HP_IELAN_INTERFACE_NAME[i]-NOTSET}
    if [ "$CHECK" = "NOTSET" ]; then
        let i=i+1
        continue
    fi
    # The "set --" construct removes whitespace
    set -- ${HP_IELAN_INTERFACE_NAME[i]}; NAME=$*
    PATH_NAME='"$HPLANSCAN" | grep -F "$NAME" | awk '{print $1}"
    if [ -z "$PATH_NAME" ]
    then
        echo "ERROR: '$NAME', the value of INTERFACE_NAME at index $i, is not found or invalid"
        let i=i+1 n=n-1
        continue
    fi
    ...
    fi
    let n=n-1 i=i+1
done
```
NOTE: If the right hand side of a driver parameter assignment is left blank in the configuration file (for example, \texttt{HP\_driver\_MTU[4]=}) , the driver startup script must use the default value for that parameter. This rule does not apply to interface name (that is, \texttt{HP\_driver\_INTERFACE\_NAME}).

NOTE: In HP-UX, initialize NIC drivers at run level 2, before the driver modules that use NICs. At run level 2, symbolic links with numbers 337 or greater point to startup scripts of kernel modules that use NICs (APA, VLAN, IP).

NIC Tool Support

To be configurable from the Network Interfaces Configuration tool, a non-native driver must provide the following capabilities:

1. Because changes made in the NIC tool can be persistent, provide a configuration file, \texttt{/etc/rc.config.d/driverconf}. In this file, define the \texttt{HP\_DRIVER\_INIT\_ARGS} variable as a string of parameter names, which the driver startup script can set during boot. The parameter names are as follows:

   \begin{itemize}
   \item \texttt{HP\_DRIVER\_STATION\_ADDRESS} \quad \text{For MAC address}
   \item \texttt{HP\_DRIVER\_SPEED} \quad \text{NIC speed}
   \item \texttt{HP\_DRIVER\_MTU} \quad \text{Driver MTU}
   \end{itemize}

   The NIC tool accepts the parameter names in the \texttt{HP\_DRIVER\_INIT\_ARGS} list and creates a dialog box that enables users to change these values.

   \textbf{NOTE:} For a driver, the NIC tool currently supports changes to the station address, speed, and MTU parameters only. The parameter names must be prepended with \texttt{HP\_DRIVER\_}.

   For example, the IELAN driver enables users to set the MAC address, driver speed, and MTU during boot. The \texttt{/etc/rc.config.d/hpielanconf} has the following as its \texttt{HP\_IELAN\_INIT\_ARGS}:

   \texttt{HP\_IELAN\_INIT\_ARGS="HP\_IELAN\_STATION\_ADDRESS\ HP\_IELAN\_SPEED\ HP\_IELAN\_MTU"}

   This parameter informs the NIC tool that the user can define station address, speed, and MTU for each instance (each instance’s name is given in the \texttt{HP\_IELAN\_INTERFACE\_NAME} parameter) of the IELAN network interface. The parameter setting for each NIC instance has the following form:

   \begin{verbatim}
   HP\_IELAN\_INTERFACE\_NAME[0]=lan9
   HP\_IELAN\_STATION\_ADDRESS[0]=
   HP\_IELAN\_SPEED[0]=
   HP\_IELAN\_MTU[0]=1400
   HP\_IELAN\_INTERFACE\_NAME[1]=lan7
   HP\_IELAN\_STATION\_ADDRESS[1]=
   HP\_IELAN\_SPEED[1]=10HD
   HP\_IELAN\_MTU[1]=
   \end{verbatim}

   The first set at subscript 0 is the parameter set for \texttt{lan9}, and the second set at subscript 1 is the parameter set for \texttt{lan7}. Any blank assignment indicates that the default value is assigned. With this capability, parameter changes made by the user are retained through the next reboot.

2. Provide support in the non-native driver for the \texttt{DL\_HP\_SET\_DRV\_PARAM\_IOCTL} and \texttt{DL\_HP\_GET\_DRV\_PARAM\_IOCTL} control requests. These control requests must support getting and setting of all the parameters that are defined in the
/etc/rc.config.d/driverconf file. In the IELAN driver, they can be MAC address, speed, and MTU.

3. Provide a /usr/sbin/hpdriver_init script. The NIC tool calls this script to configure a NIC. This script calls nwmgr to set NIC parameters. It is called by the driver init script.

Non-Native Drivers and nwmgr Command Support

For non-native drivers, HP-UX enables users to get and set NIC parameters using the nwmgr command. Some get and set commands are identical for all drivers, for example, changing the MTU size or MAC address. The common get and set commands are carried out using a few mandatory ioctl control requests, which a non-native driver must implement.

A shared library must be supplied. The functions of this shared library utilize these ioctls. These requests are sent to HP-DLPI, which then passes them to the driver without processing.

The nwmgr command is programmed to dynamically load the functions in the shared library.

In IELAN, the IELAN_LINK_SPEED ioctl serves as the entry point for all driver-specific requests.
12 Writing a Native DLPI LAN Driver

A network interface driver in HP-UX 11i v3 can be a native STREAMS DLPI or a non-native HP-DLPI based network interface driver. This chapter provides information on how to write a native STREAMS DLPI network interface driver. This chapter does not cover non-native HP-DLPI based network interface drivers. See Chapter 11 (page 221) for information on how to write a non-Native HP-DLPI based network interface driver.

You must consider writing a non-native HP-DLPI based network interface driver instead of a Native STREAMS DLPI network interface driver.

A Native STREAMS DLPI driver interface has its own implementation of the DLPI layer and has the following logical components:

- **STREAMS DLPI layer**: Processes the standard DLPI primitives, HP-UX extensions to DLPI primitives, and ioctls.
- **WSIO Interface driver**: Controls the physical NIC.

**NOTE:** In this chapter, unless explicitly stated, the terms network interface driver, Native STREAMS DLPI driver, and NATIVE STREAMS DLPI driver are used interchangeably and refer to a Native STREAMS DLPI network interface driver.

The information in this chapter is intended for developers with experience in designing and developing networking device drivers. Knowledge of STREAMS and DLPI standard is useful in understanding concepts discussed in this chapter.

Basic STREAMS module/driver development and general networking concepts and RFCs are not included in this chapter. The HP-UX networking stack does not support BSD style drivers.

For a description of how a Native STREAMS DLPI network interface driver fits in the HP-UX network subsystem and how it interacts with the other components in the HP-UX network subsystem, see Chapter 10 (page 213).

All code snippets included in this chapter are taken from the ENET sample driver. ENET is a Native STREAMS DLPI network interface driver. ENET driver presents a sample implementation of a Native STREAMS DLPI driver for an Intel 21143 based 100BaseT Ethernet PCI NIC. The ENET driver sources are available as part of the *HP-UX 11i v3 Driver Development Kit*.

**Overview**

This section presents an overview of all the steps involved in developing a Native STREAMS DLPI network interface driver for HP-UX 11i v3.

**Figure 12-1** shows the flow of developing the driver.
1. This step lists the mandatory knowledge base needed to customize the driver basic functions. The topics are as follows:

- HP-UX LAN Architecture
- Protection Synchronization for Native Drivers
- Using STREAMS/UX
- DLPI
- Native Driver Initialization
- ISR
- Driver Control Functions
- Data Processing

2. SMH Support?
   - Yes: SMH Support
   - No

3. LAN Commands?
   - Yes: LAN Commands Support
   - No

4. NetTL?
   - Yes: NetTL Support
   - No

5. LAN Statistics?
   - Yes: LAN Driver Statistics
   - No

6. OOP/CKO?
   - Yes: OOP/CKO Support
   - No

7. DKLM?
   - Yes: DKLM
   - No

Finish
HP-UX LAN Architecture

For an introduction to 11i v3 LAN Architecture and how a Native STREAMS DLPI network interface driver fits in, see Chapter 10 (page 213).

Native STREAMS DLPI Network Driver Initialization

Describes network driver initialization during system boot. It addresses `driver_install`, `driver_attach`, and `driver_init` routines. In addition, it addresses driver registration with HP-DLPI, which is required for a driver to work with other HP-UX LAN components.

STREAMS, and Driver

Addresses topics related to the STREAMS/UX subsystem, the layer, and a Native STREAMS DLPI LAN driver.

Protocol Configuration, Binding, and Demultiplexing

Describes the routines from the ENET driver that bind and demultiplex upper layer protocols to the underlying network interface card.

Inbound and Outbound Promiscuous

Presents the inbound and outbound promiscuous matrix.

ENET — A sample Network Driver

Describes the implementation of the ENET sample driver. It addresses the upper and lower parts separately. Major driver functions are also explained.

The options are available when developing a network interface driver to work with HP-UX LAN commands and other manageability utilities.

NOTE: HP strongly recommends implementing these features in network interface drivers.

2. SMH Support

Describes the changes required in the driver for the driver to work with System Management Homepage (SMH). In addition, any additional files that the driver developer must provide are also listed.

3. HP-UX LAN Commands Support

Describes the requirements to work with HP-UX LAN commands.

4. Network Tracing and Logging

Discusses Network Tracing and Logging (NetTL) tool.

5. LAN Driver Statistics

Discusses LAN driver statistics, including 64-bit MIB statistics.

6. Ioctls and OOP

Discusses the interface between the transport layer and the a DLS provider, including OOP and packet trains.

7. DLKM

Discusses how to make a network driver dynamically loadable through DLKM.

Network Driver Initialization

This section discusses all the steps to initialize a Native STREAMS Network driver on HP-UX 11i v3, specifically the static version of the ENET driver. For information on creating a dynamically loadable version of your driver, see “DLKM” (page 331).

A Native STREAMS DLPI network interface driver is part of STREAMS/UX and is used by the file system to open and close devices. For this reason, a Native STREAMS DLPI network interface
driver is both a STREAMS and WSIO-CDIO driver. Therefore, the driver registers with both STREAMS/UX and WSIO. In addition, it must also register itself with HP-DLPI.

For a detailed description of generic STREAMS driver development, see the STREAMS/UX for HP 9000 Reference Manual.

Driver Install Routine

The driver install routine must be named `driver_install`. For example, the ENET sample driver name is `enet_install`. When `enet_install` is called by the HP-UX I/O subsystem configuration process, it performs the following actions:

- Registers with WSIO.
- Registers with STREAMS/UX.
- Attaches `enet_attach`, the driver attach routine, to the PCI attach chain.
- Registers any HA event handler.
- Allocates any global locks.

The `enet_install` routine calls `wsio_install_driver` to register the ENET driver with the HP-UX I/O subsystem. The install routine of a Native STREAMS DLPI network interface driver must also call the STREAMS/UX install interface, `str_install`. The driver does not register with HP-DLPI in the `driver_install` routine; this is done in the `driver_init` routine.

The driver must define and initialize the following configuration data structures, which are passed to `wsio_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,   /* option1 */
    NULL,   /* reserved1 */
    NULL,   /* reserved2 */
    NULL,   /* reserved3 */
    NULL,   /* link */
    0,       /* device flags */
};
static wsio_drv_data_t enet_data = {
    "enet",      /* driver path */
    T_INTERFACE, /* driver type */
    DRV_CONVERGED,/* driver flag */
    0, 0,        /* minor build, minor decode */
};
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
};
```

The driver must define and initialize the following configuration data structures, which are passed to `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},    /*streamtab*/
    STR_IS_DEVICE|MGR_IS_MP|STR_SYSV4_OPEN, /*stream flags*/
    SQLVL_QUEUE,/*sync level*/
    
};

The following is the call to enet_install:

/* To save the system PCI attach chain pointer */
int (*enet_saved_attach)();
.
.
.
void enet_install (void)
{
    /* Zero the function pointers because this driver
     * does not support LLA.
     */
bzero ((caddr_t) & enet_drv_ops, sizeof (drv_ops_t));
.

    /* Register driver with WSIO. */
    if (wsio_install_driver (&enet_wsio_drv_info))
    {
        /* Register with STREAMS subsystem */
        if (str_install (&enet_str_info))
        {
            wsio_uninstall_driver(&enet_wsio_drv_info);
            return;
        }
        /* Put the driver attach function on the WSIO attach chain */
        enet_saved_pci_attach = pci_attach;
        pci_attach = (int (*)(void)) enet_linked_pci_attach;
    }
.
.

    /* Register an OLA/R event handler with the WSIO */
    wsio_install_drv_event_handler (&enet_wsio_drv_info,
         enet_wsio_event_handler);
.
.
}

Driver Attach Routine

The HP-UX I/O subsystem calls the driver_attach routine during system configuration. Each
driver_attach routine can be called multiple times and with different PCI vendor and device
IDs. The driver_attach routine must check that the passed PCI vendor ID and device ID
match the PCI vendor ID and device ID of the devices supported by the driver. If they do not
match, the driver_attach routine must call the next attach routine in the attach chain. Only
when the driver_attach routine finds a match can it proceed with claiming and initialization.
The driver calls isc_claim in driver_attach to claim a device. The driver calls
CONNECT_INIT_ROUTINE in driver_attach to arrange for the driver init routine to be called
at a later time by the HP-UX I/O subsystem. All of the driver and device initialization must be
done in the driver init routine. For information on the `driver_attach` routine, see Chapter 4
(page 77).

```c
/* This is the routine that is linked to the system PCI attach chain. */
int enet_linked_pci_attach(uint32_t id, struct isc_table_type *isc)
{
    enet_attach (id, isc);
    return enet_saved_pci_attach (id, isc);
}
/* This is the ENET PCI attach routine. */
int enet_attach(uint32_t id, struct isc_table_type *isc)
{
    if( id != DEC21143_ID)
        return WSIO_OK;
    
    CONNECT_INIT_ROUTINE (isc, enet_init);
    isc_claim(isc, &enet_wsio_drv_info);
    if(wsio_reg_drv_capability_mask(isc, event_mask) == WSIO_OK) {
    
    }
    return WSIO_OK;
} /*enet_attach*/
```

**Driver Init Routine**

The `driver_init` routine is called by the HP-UX I/O subsystem to begin driver initialization.
The driver init routine allocates the driver control block and driver data structures, sets PCI
configuration information, links the driver Interrupt Service Routine (ISR) to the PCI interrupt,
and initializes and resets the controller hardware. In the driver init routine, the driver instance
must also register itself with HP-DLPI by calling the `dlpi_propp` interface. 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;
    
    /* Create private arenas */
    
    /* Initialize the SAP list */
    
    /* Set the private data field of the isc */
    isc->if_drv_data = (caddr_t )enet_iftp;
    
    /* Set MIB statistics pointer */
    enet_iftp->lancift.mib_xstats_ptr = &enet_iftp->mib_xstats;
    /* Set up PCI configuration registers. */
    
    /* Obtain memory for Transmit Buffer Ring.
     * Note that kmalloc always gives CACHE line aligned buffers.
     */
    
    
}
```
/* Obtain memory for Transmit Descriptor Ring.
* Note that kmalloc always give CACHE line aligned buffers.
* Currently asking for 1 page of entries for Transmit Descriptor Ring.
*/
.
.
.
/* Allocate DMA handle for Tx-descr ring. */
.
.
.
/* Allocate memory for Tx-descriptor ring. */
.
.
.
/* Obtain memory for Receive Buffer Ring.
* Note that kmalloc always gives CACHE line aligned buffers.
*/
.
.
.
/* Allocate DMA handle for Rx-descr ring. */
.
.
.
/* Allocate memory for Rx-descriptor ring. */
.
.
.
/* OLA/R: Obtain memory for wsioeventp. */
.
.
.
/* Allocate locks. */
.
.
.
/* Set the driver state to DOWN. */
.
.
.
/* Reset the 2114X to a known state. */
RESET_AND_INIT_CHIP;
/* Default setting for connection type is autosense.
* Only the Core 100BT LAN can change the connection type setting.
*/
enet_iftp->conn_type = MII_AUTOSENSE;
.
.
.
/* Initialize Transmit Descriptor List. */
enet_init_tdr(enet_iftp);
/* Initialize Receive Descriptor List. */
enet_init_rdr(enet_iftp);
.
.
.
/* Initialize Statement */
msg_printf ("%s: Initializing 10/100BASE-TX card at %s....\n",
driver_name, enet_iftp->lancift.hwift.hdw_path);
/* Register with HP-DLPI */
if ( dlpi_propp(NULL, DL_HP_OP_CREATE, DLPI_PROP_DRV_INSTANCEP,
&(enet_iftp->lancift.ci), NULL, NULL) ) {
msg_printf ("enet_init: dlpi_prop() create instance failed.\n");
return -1;
}
/* Register an interrupt:
* - allocate interrupt object
* - assign IRQ line
* - activate the interrupt
*/
.
Network Driver Initialization

299


/* Do not turn on DEC 21140A interrupts yet. The reset code will
* enable interrupts.
*/

/* OLA/R
 * enet_reset() return values are checked for any errors.
 * Return value of:
 *   > 0 - Failure                    - enet_init() returns -1 to WSIO
 *   = 0 - Never Returned
 *   < 0 - loopback test in progress. - enet_init() returns 0 to WSIO
 */
if(enet_reset(enet_iftp) > 0) {
    msg_printf("%s: failed sending first setup frame. enet_init() failed\n", driver_name);
    enet_backout(enet_iftp, isc, ENET_RESET_F);
    return (-1);
}
return 0;

Registering with HP-DLPI

As a part of driver initialization, the driver init routine must register the driver instance with HP-DLPI. The driver passes information to HP-DLPI by filling out the dl_hp_create_info_t structure. For more information on this structure and its fields, see dl_hp_create_info_t(9S) in the HP-UX 11i v3 Driver Development Reference. After filling out the dl_hp_create_info_t structure, the driver calls dlpi_propp to register with HP-DLPI. See dlpi_propp(9F) in the HP-UX 11i v3 Driver Development Reference.

NOTE: In HP-UX 11i v3, the HP-DLPI version has changed to 3.

Consider the following when using dlpi_propp:

- No locks must be held while calling into this routine.
- Calls to dlpi_propp do not block or sleep.
- When a driver registers with HP-DLPI (also known as the DL_HP_OP_CREATE operation), it receives an opaque handle. This opaque handle must be saved to be used in future calls to the dlpi_propp interface.
- HP-DLPI maintains a copy of some of the driver parameters (for example, speed and MTU). However, a driver must maintain current and final values for these parameters.
- A caller must pass the buffer for the structure associated with any Get property.

The dlpi_propp routine is exported through the <sio/dlpi_drv.h> header file, and is prototyped as follows:

int32_t (*dlpi_propp)(
    void *dlpi_hdlp,
    dl_hp_op_t opcode,
    dl_hp_prop_t prop_name,
    void *valuep,
    void *rsvd1p,
    void *rsvd2p
);

When registering with HP-DLPI by calling dlpi_propp, the prop_name argument must be set to DLPI_PROP_DRV_INSTANCEP and the opcode argument set to DL_HP_OP_CREATE. Because this is the first call to HP-DLPI, the driver does not as yet have the opaque handle, dlpi_hdlp must be set to NULL. The valuep argument must point to a filled out dl_hp_create_info_t structure. Both rsvd1p and rsvd2p must be set to NULL.
After a successful registration, HP-DLPI returns a unique opaque handle to the driver in the `dhc_dlpi_hdlp` field of `dl_hp_create_info_t` structure. The driver must save this handle; it is used in subsequent calls to HP-DLPI.

If the native STREAMS driver is also a DLKM driver, it must unregister itself before unloading. This is also done by calling `dlpi_prop` with the `prop_name` argument set to `DLPI_PROP_DRV_INSTANCEP`, `opcode` set to `DL_HP_OP_DELETE`, and `dlpi_hdlp` set to the handle of the driver to unregister.

**STREAMS Transport and Driver**

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 enables 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 10.30, the HP-UX transport stack is STREAMS-based. Beginning with HP-UX 11i v2, Independent Hardware Vendors (IHVs) must develop either a Native STREAMS DLPI driver conforming to DLPI version 2.0 or develop a non-native driver that works with HP-DLPI.

This section describes how an IHV driver can integrate itself into the STREAMS/UX framework in HP-UX 11i v3. Two styles of DLPI providers are defined by the DLPI document, distinguished by the way they enable a DLPI user to select a particular Physical Point of Attachment (PPA). The Style 1 provider assigns a PPA based on the major or minor device the DLPI user opened. The Style 2 provider requires a DLPI user to explicitly identify the desired PPA using a special attach service primitive. This section illustrates the development of a Style 2 DLPI connectionless driver.

For more information regarding DLPI, see the DLPI version 2.0 standard. For information about HP extensions to DLPI, see the *HP-UX DLPI Programmers Guide*.

For information on option negotiations, packet header template, and CKO, see Chapter 15 (page 377).

**Protection and Synchronization for Network Drivers**

The major synchronization issue with networking device drivers is to avoid data corruption and race conditions when shared structures are accessed by multiple threads in multiprocessor 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 global network locking. For more information on spinlocks, see Chapter 3 (page 69).

---

**NOTE:** Each spinlock causes a busy-wait condition. Be aware of the impact on system performance caused by the frequency of acquiring a spinlock and the duration of holding a spinlock.

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 nondirect code paths are executed due to faults, traps, or interrupts.

Drivers can increase concurrency with finer granularity locks. For example, a network interface driver can use one lock for transmit path and another for receive path data structures. This enables the driver to receive and transmit concurrently.

A list of the relative predefined lock orders for spinlocks defined by HP-UX and used by the ENET driver follows:
LAN_LANX_LOCK_ORDER Lock order for a spinlock used by HP-UX LAN device drivers, such as ENET, to protect local data structures. This lock order is used by all third party networking device drivers during initialization of a spinlock used to protect device driver structures.

STREAMS_USR1_LOCK_ORDER Lock order for a spinlock used by STREAMS drivers to protect their data structures.

STREAMS Synchronization Level

HP-UX STREAMS supports Multiprocessor (MP) scalable drivers and modules. STREAMS/UX provides the following levels of parallelism:

- Queue
- Queue pair
- Module
- Elsewhere
- Global
- NOSYNC

The queue synchronization level provides the most concurrency. For more information, see the STREAMS/UX for HP 9000 Reference Manual. The amount of parallelism for modules and drivers can be configured by specifying the synchronization level in streams_info_t during str_install. The ENET Native STREAMS DLPI network interface driver uses the queue synchronization level.

Entering STREAMS from ICS

When the driver is in interrupt context, it is not in the STREAMS context. To enter the STREAMS framework correctly from non-STREAMS/UX code, 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. For more information, see the STREAMS/UX for HP 9000 Reference Manual.

Message Block Structure and Support Functions

The Message Block (MBLK) structure and functions are defined by STREAMS/UX. For more information, see the STREAMS/UX for HP 9000 Reference Manual.

Starting with HP-UX 11i v1, 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. Because 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 to read DMA data, it can corrupt the message block header. Therefore, drivers must take precautions to avoid message corruption. One solution is to verify that the data area and the header are in different cachelines.

Commonly used message block functions are as follows:

- allocb Allocates a message block.
- freemsg Frees a message block.
- pullupmsg Concatenates and aligns the data stored in complex messages.
- adjmsg Adjusts the length of the message.
- dupmsg Duplicates a simple or complex message.
STREAMs Functions and Macros

Following is a subset of STREAMS/UX functions and macros available to a Native STREAMS driver. For more information, see the STREAMS/UX for HP 9000 Reference Manual.

- **putq**: Queues a message to be processed by queue service procedure.
- **putnext**: Calls a queue put procedure.
- **canput**: Tests whether a queue can receive messages.
- **qreply**: Sends a message back upstream.
- **OTHERQ**: Other queue in the queue pair.
- **streams_put, streams_put_release**: Enables non-STREAMS/UX (for example, driver ICS) to put in a queue.

Device File, Interface Name, and PPA Number

DLPI users can access DLPI providers through generic DLPI device files, for example, a device file corresponding to a DLPI STREAMS driver. A DLPI device file can be created by `mknod` or `insf` using device driver information from `lsdev`. The following example shows the device file for the ENET sample driver:

```bash
$ lsdev | grep enet
239      -1 enet lan
$ 11 /dev/enet
* crw-rw-rw- 1 root sys 72 0x0000ef Apr 12 18:46 /dev/enet
```

Use the HP-UX `nwmgr` command to list all the LAN interfaces present in a system. Following is an output of the `nwmgr` command. It shows the ENET NIC with Station Address 0x001083F74059.

```bash
$ nwmgr
Name/ClassInstance Interface Station Address Sub-system Type Interface Related Interface
============== ========= ============== ============= ============== ========= ============== =========
lan0           UP        0x00306EEAE22E intl100  100Base-TX
lan1           DOWN      0x00306EEA52A3 igelan  1000Base-T
enet2          DOWN      0x001083F74059 enet     enet
```

IP and ARP Configuration

After the interface name and the PPA number are known, use `ifconfig` to configure IP and ARP STREAM modules on top of the ENET driver. When `ifconfig` is done for `enet2` listed by the previous `nwmgr` command, the IP and ARP streams are set up as listed in the following steps:

1. `ifconfig` opens the `/dev/enet` device file and senses that PPA configured is 0.
2. `ifconfig` issues an `ioctl` to push the IP STREAMS module to the top of ENET.
3. `ifconfig` issues another `ioctl` to issue attach and bind requests for PPA 0.
4. `ifconfig` opens the `/dev/enet` device file and issues `ioctl` to push ARP to the top of ENET.
5. `ifconfig` again performs step 3 for ARP.
6. `ifconfig` opens `/dev/ip` and uses it as dummy multiplexer, IP/enet and ARP/enet streams are linked under a dummy multiplexer.

Protocol Configuration, Binding, and Demultiplexing

This section explains the mechanisms used by an upper layer protocol (ULP) to attach and bind to a PPA. 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 do the following:

- Obtain protocol-specific information during protocol binding.
- Obtain packet-specific information.
- Process packets and information sent by the upper level protocols.

Table 12-1 summarizes the information a networking driver requires to demultiplex inbound packets for corresponding upper layer protocols.

**Table 12-1 Packet Type, Protocol Kind, and Protocol Value**

<table>
<thead>
<tr>
<th>Packet Type</th>
<th>Protocol Kind</th>
<th>Protocol Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ethernet</td>
<td>LAN_TYPE</td>
<td>TYPE value</td>
</tr>
<tr>
<td>IEEE 802.2 LLC</td>
<td>LAN_SAP</td>
<td>SAP value</td>
</tr>
<tr>
<td>SNAP</td>
<td>LAN_SNAP</td>
<td>OID + extended SNAP information</td>
</tr>
</tbody>
</table>

Protocol kind is the type of protocol to bind. Interpretation of the protocol value field depends on the protocol kind.

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 bound protocols by calling the associated STREAMS queue. The driver calls `putnext`. For information about the device driver's interrupt service routine, see the *STREAMS/UX for HP 9000 Reference Manual*. 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.

**Protocol Binding and Unbinding**

Each upper layer protocol issues a bind request to the networking driver to affect binding. The driver must keep track of all upper layer protocols currently bound to it. The networking driver must also have a way to unbind a protocol upon request.

**Protocol Demultiplexing**

One of the main functions of the device driver's ISR is to dispatch inbound packets to the appropriate upper layer protocol. To achieve that, the interrupt service routine in the driver must do the following:

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.

**Distinguishing Packet Protocol Format**

The following information can be used to determine the protocol format and type.

To determine whether the packet is an Ethernet packet or a SNAP type IEEE 802.2 LLC packet:

- If the value of the `TYPE` field of an inbound packet is equal to or greater than 0x600, the packet is an Ethernet 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 is not an Ethernet packet.
• If both the DSAP and the SSAP values are 0xAA, the packet is a SNAP packet. The protocol kind of the packet is \texttt{LAN\_SNAP}, the protocol value is 0xAA, and the extended protocol value is the five-byte SNAP protocol data specified in the SNAP header.

• In all other cases, the packet is an IEEE 802.2 LLC non-SNAP type packet. The protocol kind is \texttt{LAN\_SAP} and the protocol value is the DSAP field specified in the packet.

The relationships of protocol kind, protocol value, and protocol processing for different types of packets are shown in Table 12-1 (page 304).

After the device driver finds 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 LLC header before passing the inbound packet to the upstream queue.

### Inbound and Outbound Promiscuous

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 12-2 shows promiscuous levels.

#### Table 12-2 Promiscuous Level and Mode

<table>
<thead>
<tr>
<th>Bound/Unbound</th>
<th>PROMISC_PHY</th>
<th>PROMISC_MULTI</th>
<th>PROMISC_SAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>An unbound promiscuous stream monitors outbound traffic.</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 and broadcast packets transmitted on the interface. No outbound unicast packets are 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>An unbound promiscuous stream monitors inbound traffic.</td>
<td>The stream gets all packets on the wire regardless of SAP or address.</td>
<td>The promiscuous stream gets all multicast and broadcast packets on the wire regardless of SAP or SNAP. No unicast packets are seen in inbound traffic.</td>
<td>The promiscuous stream gets all packets that 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>A bound promiscuous stream monitors the outbound traffic.</td>
<td>The stream gets all outbound packets that match the SAP protocols that the user has bound to.</td>
<td>The stream gets all outbound multicast and broadcast packets that match the SAP protocol the user has bound to on the promiscuous stream. No unicast packets are seen.</td>
<td>This primitive has no effect on the interface.</td>
</tr>
<tr>
<td>A bound promiscuous stream monitors inbound traffic.</td>
<td>The promiscuous stream gets all packets on the wire that match the SAP protocols that the user has bound to.</td>
<td>The promiscuous stream gets all multicast, broadcast, and unicast packets that match the SAP protocol the user has bound to.</td>
<td>This primitive has no effect on the interface.</td>
</tr>
</tbody>
</table>

### Sample ENET Network Driver

This section describes the ENET driver in detail. Both the DLPI implementation and the hardware device driver portions of the ENET sample driver are discussed. In addition, this section lists the DLPI primitives and ioctl's a HP-UX Native STREAMS DLPI driver must implement.
A Native STREAMS DLPI network interface driver is two drivers in one. The upper Native STREAMS DLPI implements the DLPI layer and exists in the STREAMS context. The lower, network interface card (NIC) specific part, implements the hardware-specific functionality and exists in the WSIO-CDIO context. The DLPI Sequence in Figure 12-2 shows the basic structure of the ENET driver. There are two main data structures, enet_if_t and enet_dlpi_data_t. They establish a linkage between the DLPI-specific portion and the NIC-specific portion of the ENET driver.

Figure 12-2 shows the flow of data from user space to the low level routines.

Native STREAMS DLPI Network Driver

STREAMS/DLPI buffers and messages are processed in the upper part of the Native STREAMS DLPI network driver. In addition, the upper part of the driver handles device special file interface to the driver, DLPI primitives, and STREAMS ioctls. This section discusses the following topics:

- Requirements for Native STREAMS DLPI network driver
- Driver synchronization in STREAMS context
- Major data structures
- Open and close routines
- Control functions
- Outbound and inbound data path
- DLPI primitives and ioctls
- DLPI primitives and ioctls required to work with HP-UX LAN commands and tools
Requirements for an HP-UX Native STREAMS DLPI Network Driver

For a Native STREAM DLPI network driver to function seamlessly in an HP-UX LAN infrastructure, it must do the following:

1. Declare itself of class lan during driver registration with WSIO. Failure to do this results in the driver not being recognized by legacy HP-UX LAN commands.

   **NOTE:** The nwmgr command does not require the driver to be of class lan.

2. Successfully register each driver instance with HP-DLPI during its initialization. A driver instance is not visible to any HP-UX LAN tool unless it has been successfully registered with HP-DLPI.

3. Update driver instance status with HP-DLPI whenever the hardware state changes. For more information, see “Notifying HP-DLPI of Hardware State Changes” (page 318).

4. Update driver instance MAC address with HP-DLPI when ever a MAC address changes. For more information, see “Notifying HP-DLPI of MAC Address Change” (page 318).

5. Send DL_HP_NOTIFY_EVENT_REQ, DL_LINK_UP_IND, and DL_LINK_DOWN_IND events whenever driver instance hardware state changes occur. For more information on these primitives, see the DLPI Programmers Guide and “Notifying HP-DLPI of Hardware State Changes” (page 318).

6. Support standard DLPI primitive, certain ioctls, and certain HP extensions of DLPI primitives. This ensures that the driver works with standard HP-UX LAN commands. This also greatly enhances user experience on HP-UX because users do not need to learn different commands and tools for different network drivers.

   **NOTE:** A fully functional driver must do many other tasks besides those listed, such as DMA buffer management.

Driver Synchronization Under STREAMS Context

This section provides an overview of the synchronization of the upper and lower parts of the driver. For a non-STREAMS character I/O mechanism, synchronization between device driver and device can be accomplished by putting the device driver to sleep with the sleep kernel call on a unique number, typically an object address, while waiting for the request to complete.

Upon receipt of the request for completion information from the device, the device driver resumes the process with the wakeup kernel call. However, for a Native STREAMS DLPI driver this kind of sleep-wakeup synchronization mechanism is not permitted because STREAMS can run on either the ICS or the STREAMS scheduler context stack. Following is an example implementation in the ENET sample driver that addresses the problem.

The ENET sample driver has an enet_dlpi_wakeup routine to support the necessary synchronization between the DLPI part and the NIC-specific part. This enet_dlpi_wakeup routine simulates the STREAMS environment wakeup kernel call.

```c
void enet_dlpi_wakeup(
    caddr_t addr_ptr
);
```

Where:

- `addr_ptr` Address of the object to wake up. See the ENET sample code in the DDK for more information.

The ENET sample driver implements an enet_dlpi_stream_ioctl routine to process ioctls. Certain actions are required of the network device driver when device control requests passed through the enet_dlpi_stream_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. If the control request does one of the following:
   • If the control request completes immediately with no error, the `enet_dlpi_stream_ioctl` routine immediately returns zero to the DLPI part of the driver.
   • 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, the driver blocks the request and puts it on the ENET sleep/wakeup queue by calling `enet_dlpi_sleep`. This situation typically happens when the driver must make a request to the NIC hardware.

2. When an interrupt or timeout occurs, the driver ISR determines that the interrupt is for a previously blocked and waiting request.

3. The driver completes the previous `enet_dlpi_stream_ioctl` by placing the results in the appropriate location for that ioctl.

4. The driver calls the `enet_dlpi_wakeup` routine with the address of the sleep object that the `enet_dlpi_stream_ioctl` routine previously returned to the DLPI part of the driver.

**Major Data Structures**

This section lists the major ENET driver data structures that constitute the DLPI portion of the driver.

`enet_dlpi_data_t`

This is a per-stream data structure. Each instance of this structure represents an open stream. This data structure is defined in `enet_dlpi.h` header file.

```c
typedef struct {
    struct enet_ift *enetiftp;
    char            dl_cmd_name[MAXCOMLEN + 1];
    uint32_t        dl_index;
    cred_t         *cred;
    queue_t        *queue_ptr;
    dev_t           enet_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;
    enet_dlsap_t   *dlsap_ptr;
    uint8_t         ssap;
    uint16_t        sxsap;
    enet_mcast_list_t *enet_mcast_list;
    int             retry_cnt;
    int             promiscuous_flg;
    int             promisc_filter;
    int             ok_to_log_snap;
    uint32_t        noloopback_flg;
    uint32_t        arp_stream;
    uint32_t        ip_stream;
    uint32_t        dl_notify_events;
    int             fast_path;
    int             fast_path_pkt_type;
    int             fast_path_llc_length;
    mblk_t         *ics_mblk_list_head;
    mblk_t         *ics_mblk_list_tail;
    mblk_t         *ics_mb;
    lock_t         *ics_lock;
    mblk_t         *ioctl_mblk;
    int             prev_state;
} enet_dlpi_data_t;
```
Table 12-3 describes the fields in the enet_dlpi_data_t structure.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>enetiftp</td>
<td>Pointer to the structure for the interface associated with this open stream.</td>
</tr>
<tr>
<td>dl_cmd_name[]</td>
<td>The name of the command that opened this stream.</td>
</tr>
<tr>
<td>dl_index</td>
<td>The index assigned in the enet_dlpi_data_ptr_array[]. Also, the minor number returned for clone opens.</td>
</tr>
<tr>
<td>cred</td>
<td>Credential structure of the user who opened the 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>The enet device number for the stream.</td>
</tr>
<tr>
<td>dlsap_addr_length</td>
<td>Length of the DLSAP address.</td>
</tr>
<tr>
<td>dlsap_addr[]</td>
<td>MAC address and 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>Current DLPI state of the driver.</td>
</tr>
<tr>
<td>xidtest_flag</td>
<td>The dl_xidtest_flg from DL_BIND_REQ indicates to the driver that XID and 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>The dlsap_t structure list of logged SAPs.</td>
</tr>
<tr>
<td>ssap</td>
<td>First SAP logged on the stream.</td>
</tr>
<tr>
<td>sxssap</td>
<td>First extended SAP logged on the stream.</td>
</tr>
<tr>
<td>enet_mcast_list</td>
<td>List of multicast addresses on this stream.</td>
</tr>
<tr>
<td>retry_cnt</td>
<td>Transmit retry counter.</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 is bound with any SAP.</td>
</tr>
<tr>
<td>ok_to_log_snap</td>
<td>Special flag to permit logging of the stream.</td>
</tr>
<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 MSGNLOOP flag in the mblk message on every outbound message so the driver does not loop the packet back.</td>
</tr>
<tr>
<td>arp_stream</td>
<td>Set if this is an ARP stream.</td>
</tr>
<tr>
<td>ip_stream</td>
<td>Set if this is an IP stream.</td>
</tr>
<tr>
<td>dl_notify_events</td>
<td>Set on or off depending whether this stream wants to receive link up/down event notifications.</td>
</tr>
<tr>
<td>fast_path</td>
<td>Set if application requests to set up a 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>ics_mblk_list_head</td>
<td>Pointer to the head of the ICS mblk list.</td>
</tr>
<tr>
<td>ics_mblk_list_tail</td>
<td>Pointer to the tail of the ICS mblk list.</td>
</tr>
<tr>
<td>ics_mb</td>
<td>Pointer to the message for streams_put in ICS.</td>
</tr>
</tbody>
</table>
Table 12-3 enet_dlpi_data_t Structure (continued)

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ics_lock</td>
<td>Pointer to a lock for ics_list and ics_mb.</td>
</tr>
<tr>
<td>ioctl_mblk</td>
<td>Pointer to the mblk for hp_dlpi_ioctl.</td>
</tr>
<tr>
<td>prev_state</td>
<td>Retains the state before a pending ioctl or control request with the driver. When the request is complete, the streams are set to the correct state.</td>
</tr>
</tbody>
</table>

enet_dlpi_data_ptr_arr[]

This array holds enet_dlpi_data_t pointers to keep track of all the open streams for the driver.

Open and Close Routines

The DLPI driver can be accessed from either a regular device file 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) and the minor number must be set to the major number of the cloned driver. The clone open is useful because the application does not need to keep track of which minor number is available and does not have to use multiple device files.

In the following example, /dev/enet is a clone device file for the ENET driver:

$ ls /dev/enet*
crw-rw-rw-   1 root       sys 72 0x0000ef Apr 12 18:46 /dev/enet

The major number of the ENET driver is decimal 239:

$ lsdev | grep enet
239     -1 enet lan

Create a clone device special file for the ENET driver as follows:

$ mknod /dev/enet c 72 239

When a clone device special file is opened, the clone driver invokes the DLPI driver's open routine with the CLONEOPEN flag set. For ENET driver, this open function is enet_open. When enet_open is called, it allocates an enet_dlpi_data_t structure for the stream being created and initializes it. Also, for clone opens the driver updates the dev_t passed to the enet_open with the assigned minor number. 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 0 to MAXNOCLOSEOPEN are reserved for regular opens. For clone opens, enet_open searches for an unused minor number by calling enet_find_minor_num. For clone opens, minor numbers are reserved from MAXNOCLOSEOPEN to MAXOPENS. Both MAXNOCLOSEOPEN and MAXOPENS are defined in enet.h header file.

Control Functions

The STREAMS driver's enet_wput function, 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 the dl_primitive message. Table 12-4 lists each control function and associated functionality.
Table 12-4 Message Service Functions

<table>
<thead>
<tr>
<th>Function Name Prefixed by enet_dlpi</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>_attach</td>
<td>The information for PPA to be attached is in the ENET driver's own structures. The enet_dlpi_data_t per-stream data structure for this stream is updated with network interface information. The stream's DLPI state is changed to DL_UNBOUND.</td>
</tr>
<tr>
<td>_bind</td>
<td>The DL_BIND_REQ primitive is sent 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 enet_log_sap_value function is called. Once the driver bind is successful, dlsap_t is allocated and initialized with the protocol type and value of SAP. The enet_dlpi_data_t per-stream data structure for this stream is updated with these bind details and the stream state is changed to DL_IDLE.</td>
</tr>
</tbody>
</table>
| _control                            | Services the following primitives:  
|                                     | • DL_ENABMULTI_REQ, DL_DISABMULTI_REQ  
|                                     | • DL_SET_PHYS_ADDR_REQ  
|                                     | • DL_PROMISCON_REQ, DL_PROMISCOFF_REQ  
|                                     | • DL_HP_HW_RESET_REQ  
|                                     | The respective ioctl commands are issued to the hardware interface portion of the ENET driver from the enet_dlpi_control function. |
| _detach                             | Disables all multicasts enabled through this stream by calling the hardware interface portion of the ENET driver with the enet_dlpi_control function. If promiscuous mode was enabled by this stream, it is disabled. The clean_str_spu_sw_q routine is called to clean up any requests in the STREAMS/UX. Finally, the stream state is updated in the per-stream data structure enet_dlpi_data_t to DL_UNATTACHED. |
| _get_mib_req                        | Services MC_GET_MIB_REQ (sys/mci.h). The DL_GET_STATISTICS driver ioctl is issued to get current MIB statistics. |
| _get_mibstats                       | Calls the enet_hw_req function to get the standard MIB statistics from the driver structures. |
| _getphyaddr                         | Returns the factory permanent address or the current address, depending on the dl_addr_type value. If the address requested is the factory permanent address, the enet_hw_req function is called to get the address from the ROM located on the NIC. If the current address is requested, it is copied from ENET data structures. |
| _info                               | A service function for DL_INFO_REQ. The information is returned upstream in the dl_info_ack_t structure. If the PPA is not attached yet, MAC type and MTU are set to DL_CSMACD and IEEE8023_MTU. |
| _multicast_list                     | This function is called to service the DL_HP_MULTICAST_LIST_REQ primitive. |
| _ppa_req                            | Receipt of DL_HP_PPA_REQ results in this function being called. The driver searches ENET internal structures for the PPA requested. The requested information is returned in the dl_hp_ppa_ack_t and dl_hp_ppa_info_t structures. |
| _set_mib_req                        | This function services MC_SET_MIB_REQ. The driver DL_HP_RESET_STATS ioctl is issued to reset the MIB statistics. |
| _status                             | This function sends the current hardware state upstream (which is stored in hdw_state) in response to the DL_HP_HW_STATUS_REQ request. |
| _subs_bind                          | 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_HIERARCHICAL_BIND the dlsap_addr information in the per-stream data structure, enet_dlpi_data_t is updated with bind details. |
| _subs_unbind                        | For each dlsap_t bound, compares the unbind request SAP. If there is a match, this function calls the enet_unlog_sap_value routine. |
Table 12-4 Message Service Functions (continued)

<table>
<thead>
<tr>
<th>Function Name Prefixed by enet_dlp</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>_unbind</td>
<td>The enet_unlog_sap_value function is called to release the DLSAP bound to the stream. The dlsap_t data structure is deallocated and the bind information in the enet_dlp_data_t per-stream data structure 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.</td>
</tr>
</tbody>
</table>

Outbound Data Path

The outbound data path from a STREAMS DLPI network driver consists of sending message blocks to the lower half of the driver. The message block transmission has two paths in the sample implementation. The regular data path, which uses the DL_UNITDATA_REQ primitive, and the Fastpath. The regular path is defined in the DLPI standards. The Fastpath uses the DLPI_IOC_HDR_INFO ioctl to set up a header with the transport layer, and is an HP extension to the DLPI standard.

Regular Data Path

The message blocks transmitted on the regular data path work as follows. The stream’s put function enet_wput receives the DL_UNITDATA_REQ primitive request from the application to send a message block to a destination specified in the unitdata message. The enet_wput function calls the enet_dlp_unitdata_out function to service the request. The enet_dlp_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 the driver’s hardware output routine enet_hw_req is called.

Fastpath

For better performance, use Fastpath to transmit and receive data. The DLS user sends the DLPI_IOC_HDR_INFO DLPI ioctl to set up the Fastpath on the stream when the stream opens. The DLS provider (in this case the ENET driver) builds an LLC header template and sends it back to the DLS user. For an outbound packet, the DLS 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 lower half of the ENET 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 DLS user without building and prepending the DL_UNITDATA_IND primitive to the data. For negotiations, header info, and CKO, see Chapter 15 (page 377).

Inbound Data Path

The message that was passed to the driver along with the stream queue pointer is received by the enet_dlp_unitdata_in function. 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 DL_UNBOUND state, discard unicast packets.
- If DL_PROMISC_SAP, discard packets not destined for the stream’s network interface.

This function calls enet_dlp_unitdata_in or enet_dlp_fast_in, based on whether Fastpath is enabled or not.

The enet_dlp_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 the unitdata message and sent to the application.
STREAMS/UX enables 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 stream’s put function calls enet_wput, which calls enet_dlpi_stream_ioctl to service M_IOCTL message types. The enet_dlpi_stream_ioctl function consists of a switch block that services various M_IOCTL messages. The ioctl commands are defined in `<sys/dlpi_ext.h>`.

### DLPI_IOC_HDR_INFO

This ioctl is sent by the HP-UX IP module to a DLPI provider to establish a Fastpath stream. On a regular stream, the DLS provider expects to receive datagrams in the form of a DL_UNITDATA_REQ (M_PROTO) message block. The DL_UNITDATA_REQ primitive contains the hardware source and destination addresses from which the DLS provider can then construct an LLC header and place it in front of the IP packet before transmitting the packet. The creation of the LLC header is not performance tuned; it must be created again for every packet that is sent out. Once the connection is established, many connection-oriented protocols like TCP have identical information in every DL_UNITDATA_REQ. To avoid constructing the LLC header every time, a DLS provider can establish a Fastpath stream with the transport layer. To establish a Fastpath stream, create a packet header template and return it to the transport layer during stream initialization. A DLS provider creates an LLC header and returns it to the transport layer when it receives a DL_IOC_HDR_INFO ioctl. This mechanism is called establishing a Fastpath stream.

In the ENET sample driver, the DLPI_IOC_HDR_INFO ioctl is handled by the enet_dlpi_fast_path_ack function.

The negotiation of Fastpath happens when the transport layer sends an M_IOCTL type message with ioc_cmd set to DL_IOC_HDR_INFO. The b_cont of the M_IOCTL message block links an M_DATA type message block that contains a dl_unitdata_req_t structure, which is used to create a DL_UNITDATA_REQ primitive.

For a Native STREAMS DLPI driver that does not support Fastpath capability, the driver can return an M_IOCNAK message, in which case the transport layer uses only the DL_UNITDATA_REQ primitive for data transfer.

Figure 12-3 shows the Fastpath negotiation message formats.

**Figure 12-3 Fastpath Negotiation Message Formats**

a) M_IOCTL message from transport layer to request fastpath header

```
M_IOCTL
  DL_IOC_HDR_INFO
    b_cont
  M_DATA
    DL_UNITDATA_REQ
```

b) Fastpath M_IOCACK response from DLPI with a link level header appended

```
M_IOCACK M_IOCTL
  DL_IOC_HDR_INFO
    b_cont
  M_DATA
    DL_UNITDATA_REQ
    b_cont
    Link Layer Header
  M_DATA
```

In response to the DL_IOC_HDR_INFO ioctl, a Native STREAMS DLPI driver returns the LLC header template containing the link level header template with information filled in. If the driver also supports the Checksum Offload (CKO) feature or Out-of-Packet (OOP) data, it must return...
the CKO header, OOP, and link level header template with information filled in. For more information on CKO and OOP, see Chapter 15 (page 377).

The OOP data template part of the packet header template can be used by the transport layers to communicate additional information used during packet processing. This information is not sent out on the wire as is, but is used by a DLPI provider or the driver to improve the overall performance of the stack. For more information on OOP data, see Chapter 15 (page 377).

**DLPI_IOC_DRIVER_OPTIONS**

The `DLPI_IOC_DRIVER_OPTIONS` ioctl is used by IP (or any other in-kernel DLS user) to obtain additional information from a DLS provider regarding the drivers features. The TCP/IP module is used as an example of how this ioctl works.

While configuring an interface for IP, IP sends to the DLS provider an `M_IOCTL` message. The `M_IOCTL` message block contains an iocblk structure with `ioc_cmd` set to `DLPI_IOC_DRIVER_OPTIONS` and `ioc_countset` to `sizeof(driver_ops_t)`. Following the `M_IOCTL` message block (pointed by `b_cont` of the `M_IOCTL` message block) is an `M_DATA` message block containing a `driver_ops_t` structure.

Table 12-5 defines the `driver_ops_t` structure.

**Table 12-5 Options Negotiations Structure (driver_ops_t)**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>driver_ops_t</code></td>
<td>DLPI/transport options negotiations structure. This structure will be passed as part of the <code>DLPI_IOC_DRIVER_OPTIONS</code> ioctl from the transport.</td>
</tr>
</tbody>
</table>

**Members**

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>driver_ops_type</code></td>
<td><code>uint32_t</code></td>
<td>Features supported by the transport. The features are bit-wise flags.</td>
</tr>
<tr>
<td><code>driver_ops_type_1</code></td>
<td><code>uint32_t</code></td>
<td>Reserved. Must be set to 0.</td>
</tr>
<tr>
<td><code>driver_ops_type_2</code></td>
<td><code>uint32_t</code></td>
<td>Reserved. Must be set to 0.</td>
</tr>
</tbody>
</table>

Table 12-6 lists the features supported by HP-UX transport stack.

**Table 12-6 Enumerated Type**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dl_hp_neg_ops_one_t</code></td>
<td>Features supported by the DLS user. These bits are set in the <code>driver_ops_type</code> element of <code>driver_ops_t</code> structure.</td>
</tr>
</tbody>
</table>
### Table 12-6 Enumerated Type (continued)

<table>
<thead>
<tr>
<th>Members</th>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRIVER_CKO</td>
<td>0x1</td>
<td>DLS user asks if the driver supports outbound IPv4 or IPv6 CKO.</td>
<td></td>
</tr>
<tr>
<td>DRIVER_COW</td>
<td>0x2</td>
<td>Not supported.</td>
<td></td>
</tr>
<tr>
<td>DRIVER_LFP</td>
<td>0x4</td>
<td>DLS user asks if the driver supports a long fat pipe.</td>
<td></td>
</tr>
<tr>
<td>DRIVER_LNP</td>
<td>0x8</td>
<td>DLS user inquires if the driver supports a long narrow pipe.</td>
<td></td>
</tr>
<tr>
<td>DRIVER_PRI</td>
<td>0x20</td>
<td>DLS user asks if the driver supports VLAN tagging either in hardware (NIC) or driver software.</td>
<td></td>
</tr>
<tr>
<td>DRIVER_CKO_OUT</td>
<td>DRIVER_CKO</td>
<td>DLS user asks if the driver supports outbound IPv4 or IPv6 CKO.</td>
<td></td>
</tr>
<tr>
<td>DRIVER_CKO_IN</td>
<td>0x100</td>
<td>DLS user asks if the driver supports inbound IPv4 or IPv6 CKO.</td>
<td></td>
</tr>
<tr>
<td>DRIVER_CKO_OUT_DF</td>
<td>0x200</td>
<td>DLS user asks if the driver does not support multifragmented CKO.</td>
<td></td>
</tr>
<tr>
<td>DRIVER_TRAIN</td>
<td>0x400</td>
<td>DLS user asks if the driver supports packet trains.</td>
<td></td>
</tr>
<tr>
<td>DRIVER_FASTPATH_IN</td>
<td>0x800</td>
<td>Not supported in HP-UX 11i v3. Native DLPI must ensure that this bit is not turned on.</td>
<td></td>
</tr>
<tr>
<td>DRIVER_LLCC</td>
<td>0x1000</td>
<td>Not supported in HP-UX 11i v3. Native DLPI must ensure that this bit is not turned on.</td>
<td></td>
</tr>
<tr>
<td>DRIVER_LSO</td>
<td>0x2000</td>
<td>Not supported in HP-UX 11i v3. Native DLPI must ensure that this bit is not turned on.</td>
<td></td>
</tr>
<tr>
<td>DRIVER_PORT</td>
<td>0x4000</td>
<td>Not supported in HP-UX 11i v3. Native DLPI must ensure that this bit is not turned on.</td>
<td></td>
</tr>
</tbody>
</table>

The transport layer uses the `driver_ops_type` field to pass a bit mask of flags indicating the driver capabilities that it is inquiring about. A DLS provider should look at the bit mask passed by the transport layer and compare them with the driver features it supports. The negotiated features are set in `driver_ops_type` element of the `driver_ops_t` structure originally sent by the IP module. For example, `DRIVER_CKO_IN` might be set by the DLS provider to say that the driver supports inbound CKO feature. The flags of all the features that are not supported or understood by the DLS provider must be disabled in the `driver_ops_type` element. It is worth mentioning that the transport turns on all the supported feature bits by default when it sends this ioctl to the DLS provider. It is the DLS providers responsibility to clear all the bits that it does not support. The following is an excerpt from the ENET sample driver:

```c
void enet_dlpi_stream_ioctl(enet_dlpi_data_t *hp_dlpi_datap, mblk_t *mblkp)
{
    hw_ift_t *hwiftp = (hw_ift_t *)hp_dlpi_datap->enetiftp;
    .
    .
    ioctl_mblkp = (struct iocblk *)mblkp->b_rptr;
    cmd = ioctl_mblkp->ioc_cmd;
    .
    .
    switch (cmd) {
        .
        .
        case DLPI_IOC_DRIVER_OPTIONS:
            .
            .
```
```c
{
    driver_ops_t *ops;
    uint32_t driver_options = 0;
    if (mblkp->b_cont == NULL) {
        mblkp->b_datap->db_type = M_IOCNAK;
        ioctl_mblkp->ioc_error = EINVAL;
        ioctl_mblkp->ioc_count = 0;
        qreply(q, mblkp);
        return;
    }
    ops = (driver_ops_t *)(mblkp->b_cont->b_rptr);
    if ((ops->driver_ops_type & DRIVER_CKO_IN) &&
        !(hwiftp->features & DRIVER_CKO_IN))
        driver_options |= DRIVER_CKO_IN;
    mblkp->b_datap->db_type = M_IOCACK;
    ioctl_mblkp->ioc_error = 0;
    qreply(q, mblkp);
    return;
}

/*switch(cmd)*/

/*enet_dlpi_stream_ioctl */

DLPI_SET_NOLOOPBACK
Use this ioctl to enable or disable looping back of multicast and broadcast packets on a per-stream basis. For more information about this DLPI primitive, see the HP-UX DLPI Programmers Guide. For more information on this ioctl and how to implement it in your driver, see the enet_dlpi_stream_ioctl ENET sample driver function.

DLPI_NO_SRC_ROUTING
For more information on this ioctl and how to implement it in your driver, see the ENET sample driver function enet_dlpi_process_ioctl.

DLPI_GET_SRC_ROUTE_FLAG
For more information on this ioctl and how to implement it in your driver, see the ENET sample driver function enet_dlpi_process_ioctl.

DLPI_SET_SRC_ROUTE_FLAG
For more information on this ioctl and how to implement it in your driver, see the ENET sample driver function enet_dlpi_process_ioctl.

DL_HP_GET_DRV_PARAM_IOCTL and DL_HP_SET_DRV_PARAM_IOCTL
These two are new ioctls introduced in HP-UX 11i v2. For complete details on these ioctls, see the DLPI Programmer’s Guide. The ENET sample driver implements the following requests for these ioctls:

- DL_HP_DRV_MTU
- DL_HP_DRV_SPEED
- DL_HP_DRV_DUPLEX
```
For information on how to implement these ioctls, see the `enet_dlpi_stream_ioctl` ENET sample driver function.

Table 12-7 summarizes the DLPI primitives and ioctls used in the sample drivers. The processing of most DLPI primitives and ioctls involves driver interaction.

**Table 12-7 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 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 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</td>
<td></td>
</tr>
<tr>
<td>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>Other</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</td>
<td></td>
</tr>
<tr>
<td>DL_OK_ACK</td>
<td></td>
</tr>
<tr>
<td>DL_PHYS_ADDR_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_PROMISCOFF_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_PROMISCON_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_UNITDATA_IND DL_UNITDATA_REQ</td>
<td>DLPI Version 2.0 connectionless data transfer</td>
</tr>
<tr>
<td><strong>HP EXTENDED DLPI PRIMITIVES</strong></td>
<td></td>
</tr>
<tr>
<td>These are HP extensions to DLPI 2.0 defined in <code>&lt;sys/dlpi_ext.h&gt;</code></td>
<td></td>
</tr>
<tr>
<td>DL_HP_HW_RESET_REQ</td>
<td>Reset the physical interfaces.</td>
</tr>
<tr>
<td>DL_HP_HW_STATUS_REQ</td>
<td>Get hardware status requirements.</td>
</tr>
<tr>
<td>DL_HP_MULTICAST_LIST_REQ</td>
<td>Requests the DLS provider to return a list of all currently enabled multicast addresses on specific LAN interfaces.</td>
</tr>
<tr>
<td>DL_HP_PPA_REQ</td>
<td>Used by LAN commands.</td>
</tr>
<tr>
<td>DL_HP_RESET_STATS_REQ</td>
<td>Reset statistics.</td>
</tr>
<tr>
<td>DL_HP_NOTIFY_EVENT_REQ</td>
<td>Sent by a DLS user to be added to an event notification list.</td>
</tr>
<tr>
<td>DL_HP_USAGE_INFO_REQ</td>
<td>Return information about application with active streams.</td>
</tr>
<tr>
<td><strong>These ioctls are specific to HP defined in <code>sys/mci.h</code></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>Used between and DLS provider for negotiations.</td>
</tr>
<tr>
<td>DLPI_IOC_HDR_INFO</td>
<td>Get LLC header and CKO, if supported. This establishes the Fastpath stream.</td>
</tr>
</tbody>
</table>
Table 12-7 DLPI Primitives and ioctl (continued)

<table>
<thead>
<tr>
<th>DLPI Primitive or ioctl</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>DLPI_SET_NOLOOPBACK</td>
<td>Do not loop back the message.</td>
</tr>
<tr>
<td>DL_HP_GET_DRV_PARAM_IOCTL</td>
<td>Sent to retrieve driver-specific information.</td>
</tr>
<tr>
<td>DL_HP_SET_DRV_PARAM_IOCTL</td>
<td>Sent to set driver-specific information.</td>
</tr>
</tbody>
</table>

**NOTE:** The DL_HP_USAGE_INFO_REQ/ACK primitive is an HP extension of the DLPI standard. The definitions and structures associated with this primitive were updated for HP-UX 11i v3. Use this information in your network interface driver for HP-UX 11i v3.

Notifying HP-DLPI of Hardware State Changes

As the network driver hardware state changes, it must notify HP-DLPI of this event by calling the dlpi_propp routine. When calling dlpi_propp, use the DLPI handle for the interface whose hardware state has changed. The prop_name argument is set to DLPI_PROP_HDW_STATE and opcode is set to DL_HP_OP_SET. The new state is passed in as a pointer to the dl_hp_hw_state_t structure as the valuep argument.

Notifying HP-DLPI of MAC Address Change

A Native STREAMS driver must update its MAC address and hardware state by calling HP-DLPI. Whenever a network interface MAC address changes, the driver must notify HP-DLPI by calling dlpi_propp with the DLPI_PROP_MAC_ADDR property and opcode set to DL_HP_OP_SET. The new MAC address is passed in as argument valuep, a pointer to unsigned char.

Network Hardware Interface Driver

This section explains the code flow of the lower part of the ENET driver. This part 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 on. This section presents the code flow of the sample ENET driver as a background to the sample driver code.

This section addresses the following topics:

- “Major Data Structures”
- “Control Functions” (page 321)
- “Outbound Data Path” (page 323)
- “Outbound Promiscuous Handling” (page 324)
- “Inbound Data Path” (page 324)
- “Inbound Promiscuous Handling” (page 326)
- “Interrupt Service Routine” (page 326)
- “Releasing Pending Timeouts” (page 326)

Major Data Structures

Following are the major data structures used in the lower part of the ENET driver.

```
enet_ift_t
```

Holds network interface card PCI information, register addresses, transmit and receive buffers and descriptors, driver state, and MIB statistics. This structure also embeds a lan_ift structure that holds generic LAN information pertaining to this interface. Both enet_ift_t and lan_ift structures are ENET private and not HP-UX supported or exported. You can change these structures to fit your needs. The following shows the structure organization:

```
typedef struct enet_ift {
  /***** Device-Specific Section */
  struct isc_table_type *isc;
```
enet_srom_t *srom;  /* Serial ROM layout */
uint32  drv_state;  /* Driver state info. */
uint32  reset_state;  /* Driver reset state */
ubit8   cable_state;  /* Current Cable State */
uint32  cable_disc;  /* Count of Disconnects */
sbit8   phy_addr;  /* First phy address found */
ubit32  phy_type;  /* Phy Type */
int     fiber_card;  /* true if detected card is 100FX */

/***** PCI Configuration information */
ubit16  vendor_id;  /* Vendor ID */
ubit16  device_id;  /* Device ID */
ubit16  sub_id;  /* Subsystem ID */
ubit16  sub_vendor_id;  /* Subsystem Vendor ID */

/***** Device Configuration info */

/***/ Transmit Section */
tenet_tb_t *tbr;  /* Transmit buffer Ring */
tenet_td_t *tdr;  /* Transmit Descriptor Ring */
void    *tdr_DMA_handle;  /* DMA handle for Tx-desc ring */

ubit32  tdr_act_cnt;  /* Count of active TD's */
ubit32  tdr_act_index;  /* Active TD index */
ubit32  tdr_free_index;  /* First free TD */
ubit32  tdr_free_cnt;  /* Number of free TD's */
ubit32  tdr_errs;  /* Transmit Errors */

ubit32  tbr_act_cnt;  /* Count of active TB's */
ubit32  tbr_act_index;  /* Active TB index */
ubit32  tbr_pend_cnt;  /* Number of pending TB's */
ubit32  tbr_pend_index;  /* First pending TB */
ubit32  tbr_free_index;  /* First free TB */
ubit32  tbr_free_cnt;  /* Number of free TB's */
ubit32  tbr_pend_errs;  /* Pending Packets Dropped */
ubit32  tbr_pend_reqs;  /* total Pended Packets */
ubit32  tbr_cmplt_flg;  /* Transmission Completion Flag */

ubit32  tbr_timed_index;  /* tbr timer is set for */
ubit32  timer_ignored;  /* Count of ignored pops */
ubit32  timer_reset;  /* Count of reset pops */
ubit32  timer_valid;  /* Count of valid pops */
ubit32  timer_missed_int;  /* Count of missed int pops */
void    *setup_DMA_handle;  /*DMA handle for setup frame */

/***** Receive Section */
tenet_rd_t *rdr;  /* Receive Descriptor Ring */
tenet_rb_t *rbr;  /* Receive buffer Ring */
ubit32  rdr_raw_phys_addr;  /* Physical Address of RDR */
void    *rdr_DMA_handle;  /* DMA handle for Rx-desc ring */
ubit32 rdr_cnt; /* Count of valid RD's */
ubit32 rdr_index; /* Current RD index */
ubit32 rdr_empty_index; /* First Empty RD */
sbit32 rdr_empty_cnt; /* Number of empty RD's */
sbit32 rdr_half_empty; /* Number of times half of receive desc were empty */
ubit32 rdr_errs; /* Receive Descriptor errors */

/***** Full Duplex, speed and Transmit Threshold settings */
ubit32 full_duplex; /* Full Duplex setting */
ubit32 speed; /* Speed setting (10 or 100) */
ubit32 conn_type; /* Selected Connection type */
ubit32 xmt_threshold; /* CSR6 xmt threshold value */
ubit32 tt_display; /* xmit thrshld display val */

/***** Driver statistics */
rcv_stats_t rstats; /* Receive Statistics */
trx_stats_t tstats; /* Transmit Statistics */
uint32_t BAD_CONTROL; /* number of in pkts w/ bad contl */
uint32_t UNKNOWN_PROTO; /* number of in pkts w/ bad proto */
uint32_t RXD_XID; /* number of received XID pkts */
uint32_t RXD_TEST; /* number of received TEST pkts */
uint32_t RXD_SPECIAL_DROPPED; /* num of XID/TEST resp dropped */
mib_xEntry mib_xstats; /* MIB objects */
mib Dot3StatsEntry dot3_ext_stats; /* extended mib statistics */
mib Dot3CollEntry dot3_ext_coll; /* extended mib collisions */

/***** OLA/R event structure pointer */
et_wdio_event_t *wsioeventp;

/***** Interrupt object */
wsio_intr_object_t enet_wdio_intr;

/***** HP-DLPI related field */
void *dlpi_handle;

/***** LAN driver common state and hardware request information */
etnet_timer lantimer; /* DMA/Chntrl timer */
etnet_timeout_list_t timeout_list;

/***** Driver per instance spinlocks */
lock_t *enet_lock; /* protects enet_ift*/
lock_t *enet_r_lock; /* protects enet_ift */

struct enet_ift *next; /* link to next enet_ift */
}
etnet_ift_t;

Table 12-8 describes the fields and their purpose.

### Table 12-8 enet_ift Data Structure Fields

<table>
<thead>
<tr>
<th>Field Name/Generic Description</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<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>PCI INFO</td>
<td>Contains PCI configuration information.</td>
</tr>
<tr>
<td>DEV REG</td>
<td>Fields have control and status register addresses.</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>
</tbody>
</table>
Table 12-8 enet_ift Data Structure Fields

<table>
<thead>
<tr>
<th>Field Name/Generic Description</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>dot3_ext_coll</td>
<td>Extended MIB collisions.</td>
</tr>
<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>

enet_logged_info, enet_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 enet_logged_info{
    int protocol_val[5];
    caddr_t ift_ptr;
    queue_t *q_ptr;
    int flags;
};
```

Table 12-9 describes the fields and their purpose.

Table 12-9 Bound SAP Data Fields

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>protocol_val</td>
<td>SAP, type, or canonical value.</td>
</tr>
<tr>
<td>ift_ptr</td>
<td>Driver control block (enet_ift).</td>
</tr>
<tr>
<td>q_ptr</td>
<td>Queue pointer of the stream that did the bind.</td>
</tr>
<tr>
<td>flags</td>
<td>LANC_ON_ICS and LANC_STRIP_HEADER bits.</td>
</tr>
</tbody>
</table>

The following structure is used to link the enet_logged_info structures:

```
struct enet_logged_link{
    struct enet_logged_link *next;
    struct enet_logged_info log;
};
```

Control Functions

The enet_dlpi_* functions communicate 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, along with the message block for the request.

DL_HP_ENABMULTI

The 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.

```
typedef struct {
    mcast_list_t *hw_mcast;
    int mc_threshold;
    /* Threshold for mcast addresses */
} hw_mcast_entry_t;
```

The hw_mcast points to a linked list of mcast_list_t structures, which hold multicast addresses enabled on an interface.

```
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_dlipi_enabmulti` function calls the `enet_media_control` function to process the DL_HP_ENABMULTI command. Then the `enet_media_control` function checks validity of multicast address and calls the `ENET_UPDATE_EXT_MCAST` macro 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 updates `ref_cnt` and returns.

If the requested multicast address is not in the list, `enet_media_control` calls the `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 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**

The `enet_dlipi_disabmulti` function calls the `enet_media_control` function to process the DL_HP_DISABMULTI command. Then the `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, `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_PROMISCON**

The `enet_promisc_list[]` is a global array. Each array element is a `p_entry_t` structure corresponding to an interface.

typedef struct {
    int (*func_ptr)(__((struct lan_ift *,
                          void *, void *,  u_int));
    caddr_t     data_ptr;
    /* Queue pointer of the promiscuous stream */
    uint32_t    filter_cnt;
    /* Reference counter for SAP-based request to filter */
    uint32_t    no_filter_cnt;
    /* Reference counter for requests to receive all packets */
    uint32_t    phys_ref_cnt;
    /* Reference counter to enable physical promiscuous mode*/
    uint32_t    multi_ref_cnt;
    /* Reference counter to enable multi promiscuous mode*/
    uint32_t    sap_ref_cnt;
    /* Reference counter to enable SAP promiscuous mode */
} p_entry_t;

The `enet_dlipi_promiscon` function calls the `enet_media_control` function to process the DL_PROMISCON_REQ command. Then the `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. See the ENET driver example source in the DDK.
DL_PROMISCOFF

The enet_dlpi_promiscoff function calls the enet_media_control function to process the DL_PROMISCOFF_REQ command. Then the enet_media_control updates related fields in the p_entry_t structure and calls enet_hw_req to disable promiscuous mode on the device. See the ENET driver example source in the DDK.

DL_HP_SET_PHYS_ADDR

The driver calls enet_dlpi_set_phys_addr and enet_hw_reg, which in turn calls enet_ctrl_req to change the local address.

DL_HP_RESET_STATS

The enet_dlpi_hp_reset_stats, enet_hw_reg, enet_ctl_req, and enet_ext_clearmib functions called are to clear the MIB.

DL_HP_HW_RESET

The following functions are called in order to perform hardware reset:
enet_dlpi_hp_hw_reset, enet_hw_reg, enet_ctl_req, and enet_reset.

Outbound Data Path

The ENET driver outbound data path starts with the enet_dlpi_unitdata_out function. Figure 12-4 shows the code path for outbound data in the ENET sample driver.

Figure 12-4 Control Flowchart for Outbound Path

The enet_dlpi_unitdata_out function calls enet_hw_req to handle the write request. The enet_hw_req function processes the write requests. 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; other write requests are discarded.

Nonloopback unicast packets are transmitted in the Fastpath by calling ENET_TRANSMIT_FRAME. Multicast, broadcast, self addressed frames, frames less than 14 bytes, and frames with buffers greater than ENET_MAX_BUF_PER_FRAME are handled in the slow path by calling enet_slow_hw_req.

Nonwrite requests are passed on to enet_ctl_req.

In enet_slow_hw_req, nonunicast frames are handled in enet_transmit_complt. If the number of buffers is greater than 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 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. Calls enet_slow_complt to process nonunicast frames or setup frames. Transmit error handling is done by calling the enet_trans_error routine. If there are frames queued for transmission, calls enet_transmit_pended_frames to restart the transmission.

enet_transmit_pended_frames

While there are frames pending transmission, maps the frames, sets up the transmit descriptors, and issues a transmit poll to the device to transmit the frames.

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

Both enet_ether_ics and enet_802_2_ics call enet_route_promisc.

The enet_route_promisc function gets the promiscuous stream 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 address 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.

Inbound Data Path

Figure 12-5 shows the code path for inbound data in the ENET sample driver.
The ENET read path is on the ICS. The \texttt{enet_isr} routine is called when the network interface PCI interrupt is received and the \texttt{enet_receive_pkts} routine is invoked to process received frames.

\texttt{enet_receive_pkts} \hspace{1cm} Called from the receive interrupt handler. Some 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 \texttt{ENET\_ONLINE}, calls the \texttt{enet_process_packet} routine to process the frame. Otherwise, calls the \texttt{enet_process_looper} routine to process the frame. Replenishing the receive descriptor ring with buffers is done while doing frame receive processing.

\texttt{enet_process_packet} \hspace{1cm} Determines if the frame header is Ethernet or IEEE 802.2 and calls \texttt{enlanc\_ether\_ics} or \texttt{enlanc\_802\_2\_ics}, respectively.

\texttt{enet_process_looper} \hspace{1cm} Processes the loopback packet. The current driver substate determines the action taken. The packet buffer is validated but not used, then discarded.

\texttt{enet\_802\_2\_ics} \hspace{1cm} The packet type (802.2 or 802.2 SNAP), protocol kind (\texttt{LAN\_TYPE}, \texttt{LAN\_SNAP}, or \texttt{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 \texttt{enet\_route\_promisc} routine. The lookup for logged DLSAPs is \texttt{enet\_sap\_lookup}. 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 it is set, routes 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. 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 directly destined for the interface.

For example, the device receives all the packets on the wire for PROMISC_PHYS and all multicast and 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 then passes to the stream.

Unbound Promiscuous Stream
The enet_ether_ics and enet_802_2_ics functions call enet_route_promisc.
The enet_route_promisc gets the promiscuous stream 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.

Interrupt Service Routine
The enet_isr routine handles the interrupt generated by the NIC. It can also be invoked by the kernel when any other device that shares the same interrupt resource as the NIC generates the interrupt.
The enet_isr routine must check if the interrupt is generated by the NIC before processing the interrupt. If it is not generated by the NIC, enet_isr must return zero. The zero value indicates to the kernel that the interrupt is generated by another device.
The enet_isr routine can be called even when the NIC is suspended (see Chapter 20 (page 455)) because of interrupts generated by other devices that 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, enet_isr must return zero. The zero value indicates to the kernel that the interrupt is generated by another device.

Releasing Pending Timeouts
Before the driver gets suspended during an OLARD event or before the driver is unloaded in a DLKM operation, the driver must be free of any pending callback routines. For more information on OLARD and DLKM, see Chapter 20 (page 455).
The ENET driver maintains a list of pending timeout routines. On an OLARD suspend event or during a DLKM unload, the driver calls untimeout on all the pending timeout entries in the timeout list.
The following enum is a field in the enet_ift structure that 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 the enum refers to a function that can be on the timeout list. The flag for the function that is called through timeout is set until the function passed to timeout is called.

**SMH Support**

System Management Homepage (SMH) is the new HP-UX system administration tool. It provides both GUI and TUI interfaces for the HP-UX system administrators to configure system resources such as file systems and networks. Starting with HP-UX 11i v3, SMH also supports configuring IHV network interface drivers.

For a Native STREAMS DLPI LAN driver to be configurable using the Networking Interface Configuration tool under SMH, it must support certain DLPI primitives, including HP extensions to DLPI primitives and ioctl. In addition, a developer must also provide certain configuration files used by the NIC tool. All of the requirements and sample driver configuration files and scripts are described in Chapter 16 (page 387).

**HP-UX LAN Commands Support**

The HP-UX LAN commands enable an HP-UX user to identify and administer LAN interfaces on an HP-UX system. In addition, HP-UX LAN commands can also be used for troubleshooting purposes such as verifying data-link level connectivity between two LAN interfaces.

Starting with HP-UX 11i v3, the `nwmgr` command is the new recommended command line interface in HP-UX. For the `nwmgr` command to recognize and work with an IHV LAN interface, the network driver must follow specified guidelines and implement required ioctl and DLPI primitives.

For more information on making your Native STREAMS DLPI LAN driver work with HP-UX LAN commands, see Chapter 13 (page 337).

**Network Tracing and Logging (NetTL)**

HP-UX provides a comprehensive set of Network Tracing and Logging (NetTL) interfaces, which are part of the NetTL subsystem. Network interface drivers can use these interfaces and provide their own formatter to aid in network tracing and logging. For more information on HP-UX Network Tracing and Logging, see Chapter 14 (page 353).

**Ioctl and OOP**

During STREAMS initialization, the transport layer sends certain ioctl to the driver to learn about driver capabilities and to get a packet header template. For more information about these ioctl and OOP data, see Chapter 15 “OOP and Transport ioctl”.

**LAN Driver Statistics**

In HP-UX, administrators use the `lanadmin nwmgr` command to display driver statistics. Following is the output of `nwmgr --st -C enet -I 2` on the ENET sample network interface driver.

```
$ nwmgr --st -C enet -I 2
*** enet2 MIB statistics:
PPA Number = 2
Description = lan2 Hewlett-Packard 10/100 TX Full-Duplex Hw Rev
0. TT = 1500
MTU Size = 1500
```
For the `nwmgr` command to display these statistics, you must incorporate MIB statistics data structure into a driver per-instance data structure. The MIB statistics data structures are defined in the header file `/usr/include/sys/mib.h`.

The previous output from the `nwmgr` command is generated from the `mib_ifEntry` and `mib_Dot3StatsEntry` MIB data structures. The `mib_ifEntry` data structure is where statistics that apply to all NICs are kept. The `mib_Dot3StatsEntry` data structure maintains statistics classified as Ethernet-like. The `mib_ifEntry` data structure is defined as follows.

```c
typedef struct {
    int     ifIndex;
    char    ifDescr[64];
    int     ifType;
    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 ifOutUcastPkts;
    counter ifOutNUcastPkts;
    counter ifOutDiscards;
    counter ifOutErrors;
    gauge   ifOutQlen;
    int     ifSpecific;
} mib_ifEntry
```

Table 12-10 describes the `mib_ifEntry` data structure fields.
### Table 12-10 Fields in mib_ifEntry Data Structure

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ifIndex</td>
<td>The network management ID of the LAN interface.</td>
</tr>
<tr>
<td>ifDescr</td>
<td>A brief description of the LAN interface.</td>
</tr>
<tr>
<td>ifType</td>
<td>The type of the LAN interface.</td>
</tr>
<tr>
<td>ifMtu</td>
<td>The current MTU of the LAN interface.</td>
</tr>
<tr>
<td>ifSpeed</td>
<td>The current speed of the LAN interface.</td>
</tr>
<tr>
<td>ifPhyAddress</td>
<td>Information related to the LAN interface MAC address.</td>
</tr>
<tr>
<td>ifAdmin</td>
<td>The administrative status of the LAN interface.</td>
</tr>
<tr>
<td>ifOper</td>
<td>The operational status of the LAN interface.</td>
</tr>
<tr>
<td>ifLastChange</td>
<td>The last time (in ticks) the ifOper attribute changed for the LAN interface.</td>
</tr>
<tr>
<td>ifInOctets</td>
<td>The number of octets received from good packets.</td>
</tr>
<tr>
<td>ifInUcastPkts</td>
<td>The number of good unicast packets received.</td>
</tr>
<tr>
<td>ifInNUcastPkts</td>
<td>The number of good broadcast and multicast packets received.</td>
</tr>
<tr>
<td>ifInDiscards</td>
<td>The number of packets discarded.</td>
</tr>
<tr>
<td>ifInErrors</td>
<td>The number of received errors.</td>
</tr>
<tr>
<td>ifInUnknownProtos</td>
<td>The number of packets received with unknown protocol.</td>
</tr>
<tr>
<td>ifOutOctets</td>
<td>The number of octets transmitted.</td>
</tr>
<tr>
<td>ifOutUcastPkts</td>
<td>The number of unicast packets transmitted.</td>
</tr>
<tr>
<td>ifOutNUcastPkts</td>
<td>The number of broadcast and multicast packets transmitted.</td>
</tr>
<tr>
<td>ifOutDiscards</td>
<td>The number of packets discarded during outbound processing.</td>
</tr>
<tr>
<td>ifOutErrors</td>
<td>The number of transmit errors.</td>
</tr>
<tr>
<td>ifOutQlen</td>
<td>The number of packets waiting to be transmitted.</td>
</tr>
<tr>
<td>ifSpecific</td>
<td>A driver-specific counter or flag.</td>
</tr>
</tbody>
</table>

The `mib_Dot3StatsEntry` data structure is defined as follows. The field names are self-explanatory. The following structure contains the Ethernet-like statistics.

```c
typedef struct {
    int    dot3StatsIndex;
    counter dot3StatsAlignmentErrors;
    counter dot3StatsFCSErrors;
    counter dot3StatsSingleCollisionFrames;
    counter dot3StatsMultipleCollisionFrames;
    counter dot3StatsSQETestErrors;
    counter dot3StatsDeferredTransmissions;
    counter dot3StatsLateCollisions;
    counter dot3StatsExcessiveCollisions;
    counter dot3StatsInternalMacTransmitErrors;
    counter dot3StatsCarrierSenseErrors;
    counter dot3StatsFrameTooLongs;
    counter dot3StatsInternalMacReceiveErrors;
    counter dot3StatsExcessCollisions;
    counter dot3StatsControlFieldErrors;
    counter dot3StatsMulticastsAccepted;
} mib_Dot3StatsEntry;
```
Statistics Initialization

During driver initialization, the MIB statistics must be allocated and zeroed out. The following sample code from the ENET driver illustrates this.

```c
int enet_init( struct isc_table_type *isc)
{
    mib_ifEntry   *mib_ptr;
    char *ZZ_descr = "enet%d Hewlett-Packard LAN Interface Hw Rev %d\0";

    /* Set MIB statistics pointer */
    enet_iftp->lancift.mib_xstats_ptr = &enet_iftp->mib_xstats;

    /* Initialize the MIB objects */
    mib_ptr = &enet_iftp->mib_xstats.mib_stats;
    mib_ptr->ifPhysAddress.o_length = 6;
    bcopy((caddr_t)mac_addr,(caddr_t)mib_ptr->ifPhysAddress.o_bytes, 6UL);
    /* Extract the hardware revision from the card ID. */
    * Store the ID in the table pointer.
    /*
    msg_printf(mib_ptr->ifDescr, 64, (caddr_t)ZZ_descr,
     enet_iftp->lancift.hwift.instance_num, 0, enet_iftp->tt_display);
    mib_ptr->ifIndex   = enet_iftp->lancift.hwift.nm_id;
    mib_ptr->ifType    = ETHERNET_CSMACD;
    mib_ptr->ifMtu     = ETHERMTU;
    mib_ptr->ifSpeed   = ETHER_BANDWIDTH;
    mib_ptr->ifAdmin   = LINK_UP;
    mib_ptr->ifOper    = LINK_DOWN;
    mib_ptr->ifLastChange      = 0;
    mib_ptr->ifInOctets        = 0;
    mib_ptr->ifOutOctets       = 0;
    mib_ptr->ifInUcastPkts     = 0;
    mib_ptr->ifInNUcastPkts    = 0;
    mib_ptr->ifOutUcastPkts    = 0;
    mib_ptr->ifOutNUcastPkts   = 0;
    mib_ptr->ifInDiscards      = 0;
    mib_ptr->ifOutDiscards     = 0;
    mib_ptr->ifOutErrors       = 0;
    mib_ptr->ifInUnknownProtos = 0;
    mib_ptr->ifOutQlen         = 0;
    mib_ptr->ifSpecific        = ID_dot3;
    enet_iftp->dot3_ext_coll.dot3CollIndex   = mib_ptr->ifIndex;
    enet_iftp->dot3_ext_stats.dot3StatsIndex = mib_ptr->ifIndex;
    
}

Inbound Packet Processing

During the inbound process, the driver must update the number of inbound broadcast and multicast packets (ifInNUcastPkts), inbound unicast packets (ifInUcastPkts), and number of octets received (ifInOctets). These values are updated only upon reception of properly formed, good packets. If any errors are detected in a received packet, the number of inbound discards (ifInDiscards) and errors (ifInErrors) must be incremented.
Outbound Packet Processing

Just as in the inbound processing routine, a driver's outbound processing routines must update counters for the number of nonunicast (ifOutNUcastPkts) and unicast (ifOutUcastPkts) packets transmitted. In addition, the total number of octets transmitted must be updated (ifOutOctets). These statistics are updated only for good packets. If the transmit routines detect errors, the outbound discards (ifOutDiscards) and error (ifOutErrors) counts must be incremented.

In addition, each time a packet is successfully processed by the driver and passed on to the NIC to be transmitted, the ifOutQlen field must be updated.

Statistics Primitives

The ENET enet_dlpi_get_mibstats and enet_dlpi_get_mib_req functions provide the statistics information to applications. These functions are called in response to the DLPI primitives DL_GET_STATISTICS_REQ and MC_GET_MIB_REQ. For information on these DLPI primitives, see the DLPI Programmer’s Guide. For information on ENETs implementation, see the ENET sample driver in the DDK. For 64-bit statistics, a new primitive was added (see the following section).

64-bit Statistics Primitives

The primitive sent by a DLS user to get 64-bit statistics is DL_HP_GET_64BIT_STATS_REQ. The primitive is sent in the dl_hp_get_64bit_stats_req_t structure, shown as follows:

```c
typedef struct {
    uint32_t   dl_primitive;
    uint32_t   dl_reserved1;
    uint32_t   dl_reserved2;
} dl_hp_get_64bit_stats_req_t;
```

This primitive is valid in any attached state in which a local acknowledgement is not pending. The DLS provider responds to the request with a DL_HP_GET_64BIT_STATS_ACK if the primitive is supported and successful. Otherwise, DL_ERROR_ACK is returned. The DL_HP_GET_64BIT_STATS_ACK returns standard 64-bit statistics and 32-bit MIB-II statistics and optionally extended interface-specific MIB information for all interfaces that support the DL_HP_GET_64BIT_STATS_REQ request. The statistics are structured in the Ext_64bit_mib_t structure, defined in <sys/mib.h>. It is up to the DLPI user to determine whether there is an interface-specific MIB. If the interface does not support an interface-specific MIB, that part of the statistics is all zeros. DLPI users must also refer to and follow the relevant RFC (for standard 64-bit statistics, it is RFC 2363) to interpret the statistics.

DLKM

This section describes how to add DLKM functionality to an existing static network interface driver. For additional information on the DLKM data structure, DLKM entry points, changes to the modmeta structure, and kernel configuration utilities to manage the DLKM drivers, see Chapter 5 (page 137). Changing a static driver into a DLKM driver affects the driver compiling, linking, building, installation. Also affected are the driver bring-up and initialization. In addition, you must implement a driver unload routine. The actions that must be taken prior to unloading a driver are also described in this section.
DLKM Driver Modmeta

For a driver to be treated as a DLKM driver by the kernel configuration infrastructure, it must be compiled and built as such. You must make the following changes to the driver modmeta to notify the HP-UX kernel configuration subsystem that the driver is a DLKM driver:

- Add the loaded state to the states keyword.
- Add the unloadable keyword.

The following is a sample modmeta file:

```c
/* (C) Copyright 2002-2006 Hewlett-Packard Development Company, L.P. */

module enet {
    desc        "ENET : HP-UX 11i v3.0 (11.31) sample network driver"
    version     3.0
    type        streams_drv
    states      loaded static
    loadtimes   run driver_install
    unloadable
    dependency   dlpi
    dependency   pci
    dependency   wsio
    initfunc   driver_install   enet_install static
    initfunc   mod_load enet_load
    initfunc   mod_unload enet_unload

driver   {
    char
    class   lan
    flags   mp_safe save_conf
}
}
```

DLKM Data Structures

The DLKM data structures convey the information required by the DLKM subsystem. These data structures are in addition to the other driver data structures discussed earlier. These data structures are defined in the `<sys/moddefs.h>` and `<sys/mod_conf.h>` header files.

Following are the DLKM data structures defined by the DLKM ENET sample driver. A Native STREAMS driver must use the `str_drv_ops` structure as its `mod_operations`. For information on the DLKM data structures, see Chapter 5 (page 137).

```c
/* Wrapper table */
extern struct mod_operations str_drv_ops;       /* For STREAMS drivers only */
extern struct mod_conf_data  enet_conf_data;

/* Data structures */
static struct mod_type_data enet_drv_link = {
    "ENET - Native STREAMS DLPI Network Interface Driver (sample)",
    (void *) NULL
};
static struct modlink enet_mod_link[] = {
    { &str_drv_ops, (void *) &enet_drv_link },
    { NULL, (void *) NULL } /* Must be terminated by a NULL entry */
};
/* Main DLKM structure */
struct modwrapper enet_wrapper = {
    MODREV,                 /* Module version */
    enet_load,              /* Driver load function */
    enet_unload,            /* Driver unload function */
    (void (*)()) NULL,      /* Must be set to NULL*/
};
```

332 Writing a Native DLPI LAN Driver
DLKM Driver Entry Points

Unlike a static driver, a DLKM driver can be loaded and unloaded. Unlike a static driver, a DLKM driver does not insert itself to the PCI attach and system init chains. Instead, a new mod_wsio_attach_list_add interface is provided for a DLKM driver to identify the hardware it supports. The new load and unload entry points are prototyped as follows:

```c
int driver_load (const drv_info_t* arg)
int driver_unload (const drv_info_t* arg)
```

Following is a code fragment from the DLKM ENET sample driver enet_load function. For information on the driver load function, see Chapter 5 (page 137).

```c
int enet_load (const drv_info_t* arg)
{
    .
    .
    msg_printf ("ENET: Driver loading ...\n");
    bzero((caddr_t)&enet_drv_ops, sizeof(drv_ops_t));
    /* Register the driver with WSIO. */
    /* (void *) &enet_conf_data, /* Driver configuration data */
    enet_mod_link /* Driver modlink structure */
}
```

As a part of unloading the ENET driver, the following actions must be accomplished for each instance of the driver:

1. Unregister all the driver instances from HP-DLPI by calling the dlpi_propp interface with DL_HP_OP_DELETE.
2. Release all the allocated memory by calling kmem_arena_free.
3. Unmap all packet DMA buffers by calling wsio_unmap_dma_buffer.
4. Free all shared memory (continuous DMA) buffers by calling wsio_free_shared_mem.
5. Free all DMA handles, continuous and packet DMA, by calling wsio_free_dma_handle.
6. Release and free any spinlocks by calling spinunlock and dealloc_spinlock.
7. Call untimout for any timeouts that have not yet elapsed.
8. Halt the hardware. The ENET sample driver accomplishes this by calling enet_halt_card.
9. Deactivate and unregister the driver ISR by calling wsio_intr_deactivate and wsio_intr_free.
A driver must take the following actions before unloading the driver.

1. Destroy all the memory Arena objects by calling \texttt{kmem\_arena\_destroy}.
2. Remove the driver from system attach list by calling \texttt{mod\_wsio\_attach\_list\_remove}.
3. Unregister from the STREAMS subsystem by calling \texttt{str\_uninstall}.
4. Unregister from WSIO by calling \texttt{wsio\_uninstall\_driver}.

Certain race conditions during driver unload must be protected against by the driver. The driver is protected against any more calls to the driver open routine when the driver unload is called. Also, the system checks for current opens on the driver before calling the driver unload routine. However, the driver must ensure that there are no streams active on any driver instances before unloading the driver. The driver is guaranteed that no new streams will be created because the system blocks or fails all calls to the driver open during driver unload.

Following is a code snippet from the \texttt{enet\_unload} routine. This routine is called by the system to unload the driver. For more information about the unload process, see Chapter 5 (page 137). The ENET unload routine calls the \texttt{enet\_release\_all} routine to do most of the cleanup activities. In \texttt{enet\_release\_all}, ENET checks for active streams. If any open streams are found, the ENET unload fails with an error message to the system log. Because of timeout issues, ENET cannot unload if any instances of the ENET driver are in the \texttt{RESET} state. In this routine, ENET checks that no instances are in \texttt{RESET} before bringing down the instance and releasing resources.

```c
int enet_unload (const drv_info_t* arg){
    msg_printf ("ENET: Driver unloading ...
);    /* Check and release all per-NIC resources. */
    if ( (retval = enet_release_all()) != 0 ) {
        return retval;
    }
    /* Remove the driver from the WSIO attach list */
    if ( mod_wsio_attach_list_remove (MOD_WSIO_PCI, &enet_attach) ) {
        msg_printf ("ENET: mod_wsio_attach_list_remove() failed.
);        return ENXIO;
    }
    /* Unregister from the STREAMS subsystem */
    if ( (retval = str_uninstall (&enet_str_info) != 0) ) {
        msg_printf ("ENET: str_uninstall() failed.
);        return retval;
    }
    /* Unregister from WSIO */
    if ( wsio_uninstall_driver (&enet_wsio_drv_info) ) {
        msg_printf ("ENET: wsio_uninstall_driver() failed.
);        return ENXIO;
    }
    msg_printf ("ENET: Driver unload completed successfully.
);    return 0;
}
```

Following is the \texttt{enet\_release\_all} routine, where most of the unload activities are performed.

```c
int enet_release_all (void)
{
    /* Check for any open streams. No need to hold the enet\_dlpi\_data\_ptr spinlock, since there are no new opens or closes while the driver unloads. */
    for(i = 0; i < MAXOPENS; i++) {
        if (enet_dlpi_data_ptr_array[i] != (enet_dlpi_data_t *)NULL) {
            /* Found an open stream. Fail unload. */
            msg_printf ("ENET: Cannot unload driver. Open stream present - \%d. \n", i);
            return EBUSY;
        }
    }
}
```

334 Writing a Native DLPI LAN Driver
/* Check that none of the ENET NICs are in RESET. If any NICs are in
 * RESET, unload fails.
 * Currently a possibility exists where unload() fails because of a
 * NIC drv_state = ENET_RESET, and marks other NICs down.
 */
enet_iftp = enet_ift_list_ptr;
while (enet_iftp) {
    spinlock (enet_iftp->enet_lock);
    if (enet_iftp->drv_state == ENET_RESET) {
        spinunlock (enet_iftp->enet_lock);
        msg_printf ("ENET: Cannot unload driver. ENET PPA %d in RESET.\n", 
enet_iftp->lancift.hwift.ppa);
        return EBUSY;
    }
    /* Set the state to inform the ISR routine not to process
     * any interrupts.
     */
enet_iftp->drv_state = ENET_DOWN;
    msg_printf ("ENET: ENET PPA %d state now down.\n", 
enet_iftp->lancift.hwift.ppa);
    spinunlock (enet_iftp->enet_lock);
    enet_iftp = enet_iftp->next;
}
/* Release all resources for all the ENET NICs. No
 * need to hold any locks any more because:
 * - no streams open, and none will be allowed by the kernel.
 * - all drv_states are marked ENET_DOWN, so no interrupts will be
 *   processed by the ISR.
 * - no possibility of timeout() because no cards are in reset.
 */
/* Free all spinlocks */
    
    msg_printf ("enet_release_all: ENET PPA %d all resources released.\n", 
enet_iftp->lancift.hwift.ppa);
    temp_enet_iftp = enet_iftp->next;
    kmem_arena_free (enet_iftp, M_WAITOK);
    enet_iftp = temp_enet_iftp;
} 

return 0;
The HP-UX LAN commands enable you to list and administer LAN interfaces on an HP-UX system. You can also use HP-UX LAN commands for troubleshooting purposes, such as verifying data-link level connectivity between two LAN interfaces.

The `nwmgr` command is introduced with the HP-UX 11i v3 release. It enables you to manage LAN-based network interfaces and RDMA interfaces, and supports the features of legacy LAN commands.

The `nwmgr` command line interface is different from the legacy LAN command line interface. For more information about the `nwmgr` command line interface, see `nwmgr(1M).

The `nwmgr` command supports Role Based Access Control (RBAC) feature, which administers security in large networked applications. For more information about RBAC, see the *HP-UX System Administrator’s Guide*.

HP recommends that Independent Hardware Vendor (IHV) drivers use the `nwmgr` command to manage LAN-based network interfaces. For the HP-UX LAN commands and the `nwmgr` command to recognize and work with a network interface driver, the driver must follow certain guidelines and implement required ioctls and DLPI primitives listed in this chapter. These commands work with both native STREAMS DLPI drivers and non-native HP-DLPI drivers. HP-DLPI based drivers handle and provide most of the ioctls and DLPI primitives.

The legacy LAN commands recognize only those interfaces registered with HP-DLPI as LAN class drivers. The HP-DLPI primitives and ioctls mentioned in this chapter are defined in the `/usr/include/sys/dlpi.h` and `/usr/include/sys/dlpi_ext.h` header files.

NOTE: The `lanadmin`, `lanscan`, and `linkloop` commands are deprecated. These will be removed in a future HP-UX release. HP recommends the `nwmgr` command to perform all network interface-related tasks.

The `nwmgr` command recognizes both interface drivers that are registered and not registered with HP-DLPI as LAN class drivers. For more information on primitives, ioctls, and related structures, see the HP-DLPI manpages in the *HP-UX 11i v3 Driver Development Reference*.

Figure 13-1 shows how various network interface drivers, LAN commands, and shared libraries fit and interact.

---

**Figure 13-1 LAN Command Flow**
The nwmgr Command Architecture

The nwmgr command includes the following components:

- The nwmgr command executable
- The nwmgr driver shared libraries, which have routines that perform specific tasks for HP-UX subsystems
- The nwmgr service shared library, which exports common services to other driver shared libraries.

The nwmgr service shared library includes the following services:

- Common services for all the subsystems
- Services specific to HP-DLPI recognized subsystems
- Services specific to non HP-DLPI recognized subsystems

Figure 13-2 shows the high-level architecture of the nwmgr command.

Figure 13-2 The nwmgr Command Architecture

How nwmgr Works

The nwmgr command interacts with the driver shared library to perform driver-specific tasks, and uses the services library to perform common tasks. The nwmgr command also loads the services library.

When executed, the nwmgr command parses the command-line arguments and validates the command for the correct usage before passing the control to the appropriate driver shared library. The information obtained from parsing the command-line arguments is filled in the netmgr_cmd_t structure and passed as a parameter to the driver_netmgr_main driver shared library entry point function.

You can create configuration file for a driver, and include it in the /etc/rc.config.d directory. The configuration file name must be of the form hpdriver_name conf, and all the entries in this file must adhere to the standard name=value pair format.

nwmgr Command Syntax

The nwmgr command syntax is as follows:

$ nwmgr [operation] [operational qualifier] [target] [target qualifier]

The four options specified in the nwmgr command syntax supports both single letter and multiletter abbreviations in the command-line interface. For more information on command syntax, see nwmgr(1M).

The nwmgr command provides the following output:

- Human-readable output Used to view the output.
- Scriptable (Parsable output) Used by scripts to parse the output.
Getting Subsystem or Interface Information

Use the `get` operation to receive information about a driver or an interface. This operation supports the following options:

- Display all subsystems or interfaces.
- Display a tabular interface listing (similar to the `lanscan` command) or an extended interface listing.
- Display attributes or other information about specific subsystems or interfaces.
- Display statistics for an interface.
- Display statistics counters iteratively (similar to the `netstat` command).
- Display help for a subsystem.

Usage

To get information about a driver or interface, use the following syntax:

```
$ nwmgr [--get | --script] [target]
```

The following primitives and ioctl calls are used by `nwmgr` to get attribute information:

The DLPI primitive used to get a list of all subsystems or interfaces is `DL_HP_EXT_PPA_REQ`.

The following table lists the standard attributes:

<table>
<thead>
<tr>
<th>Task</th>
<th>Request</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get information about speed</td>
<td><code>DL_HP_DRV_SPEED</code></td>
<td><code>DL_HP_GET_DRV_PARAM_IOCTL</code></td>
</tr>
<tr>
<td>Get information about MTU</td>
<td><code>DL_HP_DRV_MTU</code></td>
<td><code>DL_HP_GET_DRV_PARAM_IOCTL</code></td>
</tr>
</tbody>
</table>

To get information about MAC address, `nwmgr` invokes the `netmgr_get_macaddr` service. The DLPI primitive that gets the MAC address is `DL_PHYS_ADDR_REQ`.

To set the MAC address, use the `netmgr_set_mac` service. The DLPI primitive used for this operation is `DL_PHYS_ADDR_REQ`.

Modifying Attribute Values of a Network Interface

Use the `set` operation to do the following:

- Modify the current attribute values of a network interface.
- Modify the persistently saved attribute values of a network interface.

Usage

To set the attributes of an interface to a new value, use the following syntax:

```
```

The following table lists the standard attributes, and the following primitives and ioctl calls are used to set or modify the attributes:

<table>
<thead>
<tr>
<th>Task</th>
<th>Request</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set speed</td>
<td><code>DL_HP_DRV_SPEED</code></td>
<td><code>DL_HP_SET_DRV_PARAM_IOCTL</code></td>
</tr>
<tr>
<td>Set MTU</td>
<td><code>DL_HP_DRV_MTU</code></td>
<td><code>DL_HP_SET_DRV_PARAM_IOCTL</code></td>
</tr>
</tbody>
</table>

To set the MAC address, `nwmgr` invokes the `netmgr_set_mac` service. The DLPI primitive used to set the MAC Address is `DL_SET_PHYS_ADDR_REQ`.

nwmgr Command Syntax  339
Resetting or Clearing Components of a Subsystem or an Interface

Use the reset operation to do the following:

• Reset an interface, a component of an interface (for example, Ethernet PHY), or an entire subsystem.
• Clear statistics of an interface or a subsystem.

Usage

To reset an interface or a subsystem, use the following syntax:

```
netmgr --reset [-script] [other options] target
```

The `netmgr_reset_stats` service is used to reset the statistics. This service in turn uses the DLPI primitive `DL_HP_RESET_STATS_REQ`.

The DLPI primitive used to reset an interface is `DL_HP_HW_RESET_REQ`.

Performing a Critical Resource Analysis (CRA)

A CRA operation performs a critical resource analysis of LAN network interfaces.

The DLPI and non-DLPI drivers can use either the HP CRA or non-HP CRA to analyze the network interfaces.

If the IHV DLPI drivers use their own CRA mechanism, the `DL_HP_EXT_SUPP_CRA` flag must be set in the `dl_features_one` field of the `dl_hp_create_info_t` structure while registering with HP-DLPI. For more information on how to develop a CRA shared library, see the CRA chapter in the DDG guide.

NOTE: Currently, there is no support for CRA in the `netmgr` services shared library for the non-DLPI drivers. Non-DLPI drivers must provide their own CRA.

The critical nature of the interface is determined based on the use of the interface by any of the following upper layers:

• Applications attached to it
• Upper protocol layer kernel modules bound to it, such as IP and ARP modules

An interface is assigned as one of the following:

Data critical interface  If this interface is deleted, it results in data loss to the upper layers.

System critical interface  If this interface is deleted, it affects the system.

Not critical interface  If this interface is deleted, there is no data loss to upper layers.

Usage

The syntax for performing a CRA of an interface is as follows:

```
netmgr [-cra] [--script] [other options] target
```

To analyze a LAN network critical resource, the `netmgr_perform_cra` service is used. This service uses the main CRA infrastructure by invoking the `cra_main` CRA entry point. The CRA framework performs the analysis.

Troubleshooting Network Interface Level Problems

The `diagnose` operation enables you to troubleshoot problems at a network interface level.

Each of the `diagnose` set of operations in the `netmgr` command exercises targeted functionality on a NIC and networking subsystem. The `netmgr diagnose` operation helps in troubleshooting a link connectivity check.
Usage
nwmgr --diagnose [ [diag_op_type] |[-A attributes]] [[other options]]
Where:

```
diag_op_type  link
```

Displaying Driver-Specific Help

The driver shared library must handle help operations by providing information about different options supported by the subsystem.

Usage

Use the following syntax to get help on the subsystems:

```
nwmgr --help --subsystem subsystem name
```

Example:

The following example output is displayed for the enet subsystem:

```
nwmgr -h [-g | -s | -r | --diag] [ -c lan | -S enet ]
nwmgr [-g] [-v] [ -c lan<instance> | -S enet ]
nwmgr [-g] --st [mib | extmib | all]* -c lan<instance>
nwmgr [-g] {-A all | -A attr1, attr2, ...} -c lan<instance>
nwmgr [-g] -q info -c lan<instance>
nwmgr -s -A attr1=value1, attr2=value2... -c lan<instance>
nwmgr -s -A all --fr cu[rent] -c lan<instance>
nwmgr -r -c lan nwmgr -r --st -c lan
nwmgr --diag -A dest=<mac addr>[-A timeout=<seconds>, pktsize=<bytes>]
    [-it<number> ] -c lan<instance>
```

Other general command options: -v (verbose), --sc[ript] (parsable output format)

The nwmgr Driver Shared Library

The `nwmgr` command enables you to write driver shared libraries for LAN-based interfaces. The following topics are covered in this section to help IHVs understand and write subsystem shared libraries:

- The `nwmgr` services library provided for driver shared libraries.
- Guidance on how to write a driver shared library. A sample shared library, the ENET subsystem shared library sample code, is provided.
- Instructions on how to compile, build, and install a driver shared library.

The `nwmgr` command offers services listed in the following tables for a driver shared library. Some of the services can be used by non HP-DLPI drivers.

Table 13-1 provides information about APIs that enable you to get data from subsystems or interfaces.

### Table 13-1 General Purpose and Data Retrieval Services

<table>
<thead>
<tr>
<th>API</th>
<th>Description</th>
<th>Used by Non-HP-DLPI Drivers</th>
</tr>
</thead>
<tbody>
<tr>
<td>netmgr_get_dlpi_fd</td>
<td>Opens the device file for the specified LAN interface and then attaches to its Physical Point of Attachment (PPA). The <code>dl_device_name</code> field in the <code>dl_hp_ext_ppa_info_t</code> structure for the interface is used to form the device file name.</td>
<td>No</td>
</tr>
<tr>
<td>netmgr_dlpi_open</td>
<td>Opens the <code>/dev/dlpi</code> device file and returns the file descriptor to the caller.</td>
<td>No</td>
</tr>
<tr>
<td>API</td>
<td>Description</td>
<td>Used by Non-HP-DLPI Drivers</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>netmgr dlpi_attach</td>
<td>Attaches the stream, opened using the netmgr dlpi_open function, to the instance specified as an input parameter.</td>
<td>No</td>
</tr>
<tr>
<td>netmgr dlpi_detach</td>
<td>Detaches the stream, represented by fd, from its currently attached interface.</td>
<td>No</td>
</tr>
<tr>
<td>netmgr get ppa list</td>
<td>Obtains the list of dl_hp_ext_ppa_info_t structures from the HP-DLPI driver by invoking the dl_hp_ext_ppa_req primitive. Because the function dynamically allocates memory required to hold the list of structures, the caller must free the memory with the *free_ppa_list function.</td>
<td>No</td>
</tr>
<tr>
<td>netmgr get subsys ppa list</td>
<td>Obtains the list of dl_hp_ext_ppa_info_t structures from the HP-DLPI driver for a given subsystem by invoking the dl_hp_ext_ppa_req primitive. Because the function dynamically allocates the memory required to hold the list of structures, the caller must free the memory with the *free_ppa_list function.</td>
<td>No</td>
</tr>
<tr>
<td>netmgr get ppa info</td>
<td>Obtains the dl_hp_ext_ppa_info_t structure from the HP-DLPI driver for an interface. The interface name is specified in the form lan instance number.</td>
<td>No</td>
</tr>
<tr>
<td>netmgr perform cra</td>
<td>Serves as a wrapper and invokes the CRA framework to perform a Critical Resource Analysis on a set of hardware paths. This routine must be invoked by subsystems that choose to perform an implicit CRA before executing a destructive operation on its interfaces. This routine can also be invoked as part of a preview for a destructive operation. For more information on CRA framework, see Chapter 21 (page 477). This routine is a wrapper that invokes the cra_main API in the CRA framework shared library. Hence the function supports all the events and contexts identified by the CRA framework. The calling subsystem must determine the required events and contexts.</td>
<td>No</td>
</tr>
<tr>
<td>netmgr find opt</td>
<td>Searches for a specific option within a list of netmgr_nameval_t structures, based on its multiletter name.</td>
<td>Yes</td>
</tr>
<tr>
<td>netmgr find opt for int</td>
<td>Searches for a specific option within a list of netmgr_nameval_t structures, based on its integer equivalent.</td>
<td>Yes</td>
</tr>
<tr>
<td>netmgr find opt by value</td>
<td>Searches for a specific option within a list of netmgr_nameval_t structures, based on its name and value.</td>
<td>Yes</td>
</tr>
<tr>
<td>netmgr find num opts</td>
<td>Finds the number of occurrences of an option within a given list based on its multiletter name.</td>
<td>Yes</td>
</tr>
<tr>
<td>API</td>
<td>Description</td>
<td>Used by Non-HP-DLPI Drivers</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>netmgr_get_instance</td>
<td>Extracts the instance number for a given class instance in string form.</td>
<td>Yes</td>
</tr>
<tr>
<td>netmgr_get_class_in_cmd</td>
<td>Extracts the class name from the options specified in the command line.</td>
<td>Yes</td>
</tr>
<tr>
<td>netmgr_get_inst_in_cmd</td>
<td>Extracts the instance specified in the command line.</td>
<td>Yes</td>
</tr>
<tr>
<td>netmgr_get_ifname</td>
<td>Forms the class_instance and validates that instance to contain only numerals if the command line options contain either --class_instance or --class --instance.</td>
<td>Yes</td>
</tr>
<tr>
<td>netmgr_get_dlpi_ifname</td>
<td>If options in the command line contain --class_instance or --class --instance, this routine forms the class_instance and validates the instance to contain only numerals. In addition, it validates the class_instance among the HP-DLPI registered interfaces. If --subsystem is specified along with --instance, this routine gets the class from HP-DLPI and forms the class_instance from it. This service must only be used on existing interfaces for operations such as --get, --set, --reset, --enable, --disable, --cra, --delete, or --diagnose. If the nwmgr command line creates a new virtual interface using the --add operation, this service returns EINVAL because it will not yet be present in the HP-DLPI list of registered interfaces.</td>
<td>No</td>
</tr>
<tr>
<td>netmgr_check_class_in_dlpilist</td>
<td>Checks whether the given class is a valid HP-DLPI registered interfaces class.</td>
<td>No</td>
</tr>
<tr>
<td>netmgr_check_class_inst_in_dlpilist</td>
<td>Checks whether the given class instance is a valid HP-DLPI registered interfaces class instance.</td>
<td>No</td>
</tr>
<tr>
<td>netmgr_get_dlpi_usage_info</td>
<td>Gets the usage information of the file descriptor that was opened and attached to an interface in the form of a dl_hp_usage_info_t structure. The size of the usage_info variable varies depending on the number of upper layer modules and applications that are attached to it. Memory for the variable is allocated by the routine. It is the caller’s responsibility to invoke the free_func function to free the allocated memory after the process is completed.</td>
<td>No</td>
</tr>
<tr>
<td>netmgr_get_macaddr</td>
<td>Gets the MAC address of the file descriptor that was opened and attached to an interface from HP-DLPI. The memory required for storing the MAC address needs to be preallocated by the caller.</td>
<td>No</td>
</tr>
<tr>
<td>API</td>
<td>Description</td>
<td>Used by Non-HP-DLPI Drivers</td>
</tr>
<tr>
<td>-------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>netmgr_get_fact_macaddr</td>
<td>Gets the factory MAC address of the file descriptor that was opened and attached to an interface from HP-DLPI. The memory required for storing the factory MAC address must be preallocated by the caller.</td>
<td>No</td>
</tr>
<tr>
<td>netmgr_get_iface_type</td>
<td>Gets the string defining the interface type of the file descriptor that was opened and attached to an interface by invoking the DL_HP_GET_DRV_IOC ioctl. The memory required for storing the interface type string must be preallocated by the caller.</td>
<td>No</td>
</tr>
<tr>
<td>netmgr_get_mibstats</td>
<td>Gets the MIB statistics of the file descriptor that was opened and attached to an interface by invoking the DL_GET_STATISTICS primitive. The memory required for storing the Ext_mib_t structure must be preallocated by the caller.</td>
<td>No</td>
</tr>
<tr>
<td>netmgr_get_64bit_stats</td>
<td>Gets the 64-bit MIB statistics of the file descriptor that was opened and attached to an interface by invoking the DL_GET_64BIT_STATS_REQ primitive. The memory required for storing the Ext_64bit_mib_t structure must be preallocated by the caller.</td>
<td>No</td>
</tr>
<tr>
<td>netmgr_get_dlpi_stats</td>
<td>Gets the internal statistics of the file descriptor that was opened and attached to an interface by invoking the DL_HP_GET_INTERNAL_STATS_REQ primitive. The memory required for storing the dl_hp_internal_stats_t structure must be preallocated by the caller.</td>
<td>No</td>
</tr>
</tbody>
</table>
Table 13-1 General Purpose and Data Retrieval Services (continued)

<table>
<thead>
<tr>
<th>API</th>
<th>Description</th>
<th>Used by Non-HP-DLPI Drivers</th>
</tr>
</thead>
<tbody>
<tr>
<td>netmgr_update_version</td>
<td>If the nwmgr version supported by the subsystem is less than the version maintained in the nwmgr command, the subsystem must invoke this function to downgrade the version maintained in the nwmgr command. The version supported by the nwmgr command is passed to the subsystem through the subsystem_netmgr_main entry point. This API allows the nwmgr command to work with older versions of the nwmgr subsystem library.</td>
<td>Yes</td>
</tr>
<tr>
<td>netmgr_set_priv</td>
<td>A network management-specific authorization called hpux.network.config was created in the RBAC database. This authorization is required to perform potentially destructive operations such as --set, --add, --delete, --enable, --disable, or --reset. If the user-specified operation is among the list of potentially destructive operations, this function must be invoked. This function checks whether the user is authorized to perform configuration related operations by invoking the ACPS API acps_simplecheckauth. If it returns ACPS_ALLOW, the effective privileges for the process are upgraded to the privilege set specified in the priv_set input parameter by invoking the priv_set_effective routine. If both these steps succeed, a success value is returned. The caller must proceed with the user request based on the return value.</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 13-2 provides information about APIs that enable IHVs to display the retrieved data either in readable or parsable format.

Table 13-2 General Display Services

<table>
<thead>
<tr>
<th>API</th>
<th>Description</th>
<th>Used by HP-DLPI / Non HP-DLPI Drivers</th>
</tr>
</thead>
<tbody>
<tr>
<td>netmgr_show_union</td>
<td>Displays a single attribute value, given its data type and display format.</td>
<td>Yes</td>
</tr>
<tr>
<td>netmgr_show_attrib_value</td>
<td>Displays a single attribute given in the netmod_attrib_t structure. This function also displays errors (if any) for the previous failure on that attribute.</td>
<td>Yes</td>
</tr>
</tbody>
</table>
| netmgr_show_attribs          | Displays the following:  
  • All attributes of an interface, given in the list of netmod_attrib_t structures.  
  • Errors (if any) for the previous failure on an attribute set such as current, saved, or any other named attribute set based on the contents of the attrib_set input parameter.                                                                                                                                       | Yes                                   |
<table>
<thead>
<tr>
<th>API</th>
<th>Description</th>
<th>Used by HP-DLPI / Non HP-DLPI Drivers</th>
</tr>
</thead>
<tbody>
<tr>
<td>netmgr_clone_attrib</td>
<td>Creates a duplicate copy of the source attribute.</td>
<td>Yes</td>
</tr>
<tr>
<td>netmgr_free_cloned_attrib</td>
<td>Frees the cloned attribute.</td>
<td>No</td>
</tr>
<tr>
<td>netmgr_show_lan_if_list</td>
<td>Displays a fixed set of attributes for each of the PPAs represented by its dl_hp_ext_ppa_info_t structure in ppa_list. If the disp_flag input parameter is set to NI_SCRIPT, the display appears in scriptable format. Otherwise, it appears in tabular format.</td>
<td>No</td>
</tr>
<tr>
<td>netmgr_show_ib_if_list</td>
<td>Displays a fixed set of attributes for each interface represented by its it_netmgr_ia_t structure in ib_list. If the disp_flag input parameter is set to NI_SCRIPT, the display appears in scriptable format. Otherwise, it appears in tabular format.</td>
<td>No</td>
</tr>
<tr>
<td>netmgr_show_lan_if_verbose</td>
<td>Displays a fixed set of attributes in the dl_hp_ext_ppa_info_t structure for the given PPA. If the disp_flag input parameter is set to NI_SCRIPT, the display appears in scriptable format. Otherwise, it appears in readable form. This service excludes display of the autoneg attribute for virtual interfaces. If there are attributes to be displayed in addition to the fixed set, subsystems can provide them in the attrib_list parameter.</td>
<td>Yes</td>
</tr>
<tr>
<td>netmgr_show_stats</td>
<td>Gets the statistics for the given instance based on the option specified in stats_option, and displays it. If stats_option is set to NULL, the default is mibstats. If the disp_flag input parameter is set to NI_SCRIPT, the display appears in scriptable format. Otherwise, it appears in readable format.</td>
<td>Yes</td>
</tr>
<tr>
<td>netmgr_show_cra</td>
<td>Invokes the CRA framework to perform a CRA on a given hardware path and displays the result. This function must be invoked by subsystems using the -cra operation. If the disp_flag input parameter is set to NI_SCRIPT, the display appears in scriptable format. Otherwise, it appears in readable form. This routine invokes the cra_main API of the CRA framework with event set to ANALYZE and context string set to CardDelete. The usage information returned by cra_main is displayed without any further formatting.</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 13-3 provides information about APIs that enable IHVs to modify card identification and statistics information online.
Table 13-3 Online Configuration Services

<table>
<thead>
<tr>
<th>API</th>
<th>Description</th>
<th>Used by HP-DLPI / Non HP-DLPI Drivers</th>
</tr>
</thead>
<tbody>
<tr>
<td>netmgr_set_macaddr</td>
<td>Changes the MAC address of an interface to the address specified in macaddr.</td>
<td>No</td>
</tr>
<tr>
<td>netmgr_set_macaddr_to_fact</td>
<td>Obtains the factory MAC address of the interface and sets it as the current MAC address. The file descriptor must be opened and attached to an interface whose MAC address is already set.</td>
<td>No</td>
</tr>
<tr>
<td>netmgr_reset_stats</td>
<td>Resets the interface related statistics through the HP-DLPI primitive DL_HP_RESET_STATS_REQ.</td>
<td>No</td>
</tr>
<tr>
<td>netmgr_reset_if</td>
<td>Invokes the HP-DLPI primitive DL_HP_HW_RESET_REQ to perform a hardware reset of the interface.</td>
<td>No</td>
</tr>
<tr>
<td>netmgr_set_ifadmin</td>
<td>Invokes the HP-DLPI primitive DL_HP_SET_IFADMIN_REQ to set the ifAdmin state of the interface to UP or DOWN.</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 13-4 provides information about APIs that enable IHVs to validate the nwmgr command-line interface.
### Table 13-4 Command Validation Services

<table>
<thead>
<tr>
<th>API</th>
<th>Description</th>
<th>Used by HP-DLPI / Non HP-DLPI Drivers</th>
</tr>
</thead>
<tbody>
<tr>
<td>netmgr_verify_one_instance</td>
<td>Verifies whether the command line specifies exactly one network interface. If not, it generates an error message to stderr. The interface can be specified with a class-instance, a class and an instance, or an equivalent.</td>
<td>Yes</td>
</tr>
</tbody>
</table>
| netmgr_validate_cmd          | Validates the command line by applying the given set of policies to the command structure, except for those that are marked with NI_IGNORE bit. If the policy type is ERROR, it generates error messages to stderr for policies that match the command. If the policy type is WARN, it generates informative messages to stderr for matching policies that some options are being ignored. A policy is a bitmask, with one bit for each option. The policy bitmask indicates which option or option combination is considered erroneous or superfluous. The --time, --interval, and --iteration options are treated as a single group, and are specified with the bit NI_TIME. The NI_IGNORE bit indicates that the associated policy must be ignored. Indicate which options are unsupported rather than which are supported. This helps generate precise error or warning messages. The bitmask model can handle many policies, but not all. Policies that do not fit the model are as follows:  
• Policies that require knowing how many times an option occurred on the command line. For example, check that there is exactly one --class_instance option in the command.  
• Policies that treat many options as equivalent. For example, check that at least one among --instance and --class_instance is specified.  
• Policies that involve option arguments. For example, check that an option has at least one argument. | Yes |

Table 13-5 provides information about APIs that enable IHVs to manage subsystem configuration file, such as adding, deleting, and modifying the configuration file entries.

### Table 13-5 Configuration File Services

<table>
<thead>
<tr>
<th>API</th>
<th>Description</th>
<th>Used by HP-DLPI / Non HP-DLPI Drivers</th>
</tr>
</thead>
<tbody>
<tr>
<td>netmgr_conf_get_fd</td>
<td>Opens the configuration file (in read and write mode) specified as an input parameter.</td>
<td>No</td>
</tr>
<tr>
<td>netmgr_conf_get_indices_for_nameval</td>
<td>Retrieves indices in sorted order for entries in the configuration file which match the name and value of the attribute specified as input parameter.</td>
<td>Yes</td>
</tr>
<tr>
<td>netmgr_conf_get_indices_for_name</td>
<td>Retrieves indices in sorted order for entries in the configuration file that match the name of the attribute specified as an input parameter.</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Table 13-5 Configuration File Services (continued)

<table>
<thead>
<tr>
<th>API</th>
<th>Description</th>
<th>Used by HP-DLPI / Non HP-DLPI Drivers</th>
</tr>
</thead>
<tbody>
<tr>
<td>netmgr_conf_get_val_for_name_idx</td>
<td>Retrieves the value of the input attribute for the given index from the configuration file. It sets the retrieved value to the <code>na_value</code> field of the input <code>netmod_attrib_t</code> structure.</td>
<td>Yes</td>
</tr>
<tr>
<td>netmgr_conf_get_atlist_for_idx</td>
<td>Retrieves the list of attributes for a given index. It sets the name and value to the <code>na_name</code> and <code>na_value</code> fields of the <code>netmod_attrib_t</code> structure of each attribute in the list.</td>
<td>Yes</td>
</tr>
<tr>
<td>netmgr_conf_get_idx_list</td>
<td>Retrieves the list of unique indices present in the configuration file. If any global entries are present (entries without indices), they are represented by a single entry with –1 in the index list. Multiple entries with duplicate indices are not reported and are represented by a single entry in the index list. The <code>index_list</code> array must be preallocated by the caller and the maximum size of the preallocated array must be passed as an input parameter in <code>max_inds</code>. If the number of unique indices exceeds <code>max_inds</code>, the function is restricted by <code>max_inds</code>.</td>
<td>Yes</td>
</tr>
<tr>
<td>netmgr_conf_set_atlist_for_idx</td>
<td>Sets the list of attributes for a given index. It sets the name and value to the <code>na_name</code> and <code>na_value</code> fields of the <code>netmod_attrib_t</code> structure of each attribute in the list. If the index does not exist, this routine creates a new entry for the index in the configuration file and inserts the list of attributes for it.</td>
<td>Yes</td>
</tr>
<tr>
<td>netmgr_conf_delete_nameval_for_idx</td>
<td>Deletes the entry from the configuration file that matches the attribute name and index value given in the <code>nameval</code> structure and index input parameters, respectively.</td>
<td>Yes</td>
</tr>
<tr>
<td>netmgr_conf_delete_idx</td>
<td>Deletes all entries from the configuration file that match the index given as an input parameter.</td>
<td>No</td>
</tr>
<tr>
<td>netmgr_conf_delete_attrib</td>
<td>Deletes all entries that match the attribute name given in <code>nameval</code> from the configuration file.</td>
<td>Yes</td>
</tr>
<tr>
<td>netmgr_conf_set_attrib_value</td>
<td>Sets or modifies the value for the entry corresponding to the given attribute name and index to <code>na_value</code> present in the <code>nameval</code> structure. If the attribute does not exist for the specified index, a new entry for the attribute is created.</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 13-6 provides information about APIs that enable IHVs to troubleshoot problems encountered while using a card.
Table 13-6 Diagnostic Services

<table>
<thead>
<tr>
<th>API</th>
<th>Description</th>
<th>Used by HP-DLPI / Non HP-DLPI Drivers</th>
</tr>
</thead>
<tbody>
<tr>
<td>netmgr_linkloop</td>
<td>Performs linkloop diagnostics from a specific interface to the destination interface's MAC address. The MAC address must be specified as an attribute.</td>
<td>No</td>
</tr>
<tr>
<td>netmgr_show_linkloop_stats</td>
<td>The statistics of the linkloop diagnostics performed are available in this structure. It contains the number of test packets sent and number of test packets received.</td>
<td>No</td>
</tr>
</tbody>
</table>

For more information about `nwmgr` service APIs and data structures, see the *HP-UX 11i v3 Driver Development Reference Guide*.

Writing a Subsystem Shared Library

To write a driver shared library, follow these steps:

1. Implement the `netmgr_main` function in the driver shared library.
   The `nwmgr` core enters the driver through the `netmgr_main` function defined in the driver shared library. For more information about this API, see the API manpage definition provided in the DDK. For compiling the shared library on Itanium-based systems, 64-bit libraries must be installed on the system. For compiling the shared library on PA-RISC based systems, 32-bit libraries must be installed on the system.

2. Create a `libnetmgr_drivername.sl` symbolic link for PA-RISC systems and `libnetmgr_drivername.so` link for Itanium systems. The driver shared library must be delivered as `libnetmgr_drivername.1` on PA-RISC systems and `libnetmgr_drivername.so.1` on Itanium systems.

For more information about writing a subsystem shared library, see the sample ENET driver shared library (and associated Makefile) provided with DDK.

Sample Driver Shared Library

A sample driver shared library is provided with the ENET sample network driver. This driver shared library is functional and includes working makefiles. You are encouraged to use it as a starting point to write your own driver shared library.

Table 13-7 lists the ENET driver shared library files.

Table 13-7 ENET Shared Library Files

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Makefile</td>
<td>Automates the compilation process of the ENET driver shared library.</td>
</tr>
<tr>
<td>dsenet.msg</td>
<td>Contains standard error messages used by the ENET driver shared library.</td>
</tr>
<tr>
<td>enet_attribs.c</td>
<td>Contains attributes provided by <code>nwmgr</code>.</td>
</tr>
<tr>
<td>enet_netmgr.c</td>
<td>Contains routines specific to the ENET driver shared library.</td>
</tr>
<tr>
<td>enet_netmgr.h</td>
<td>Contains data structures and function declarations used by the ENET driver shared library.</td>
</tr>
<tr>
<td>enet_svcs.c</td>
<td>Contains the <code>nwmgr</code> service library routine definitions.</td>
</tr>
<tr>
<td>enet_svcs.h</td>
<td>Contains data structures and function declarations used by the <code>nwmgr</code> service library.</td>
</tr>
</tbody>
</table>
Compiling and Building the Shared Library

To compile and build a driver shared library, you can use the sample makefile shipped with the ENET driver shared library tar file.

The sample nwmgr shared libraries come with a Makefile file. The makefile detects the underlying platform and selects the correct options to compile and build the shared libraries. To compile and build the shared library, execute the following command in the source file folder:

```
$ make
```

On a PA-RISC system, a 32-bit shared library is built. On an Itanium-based system, a 64–bit shared library is built.

Installing the Shared Library

To install the built shared library, execute the following command:

```
$ make install
```

The Makefile copies the shared library to the appropriate directory depending upon the platform and creates all the needed symbolic links.

To uninstall the shared libraries, execute the following command:

```
$ make uninstall
```

Standard Error Codes

Table 13-8 lists the standard error codes you must use in your driver shared library.

Table 13-8 Standard Error Codes

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EINVAL</td>
<td>Invalid attribute name or unrecognized option</td>
</tr>
<tr>
<td>ERANGE</td>
<td>Invalid attribute value</td>
</tr>
<tr>
<td>EACCES</td>
<td>Attribute is read-only</td>
</tr>
<tr>
<td>EPERM</td>
<td>Insufficient privilege (no netadmin privilege)</td>
</tr>
<tr>
<td>EIO</td>
<td>Primitive or ioctl failed</td>
</tr>
<tr>
<td>ENOMEM</td>
<td>Memory allocation failure</td>
</tr>
<tr>
<td>ENXIO</td>
<td>Target unrecognized; bad class, subsystem, interface</td>
</tr>
<tr>
<td>ENOTSUP</td>
<td>Unsupported option combination</td>
</tr>
<tr>
<td>EBUSY</td>
<td>Operation not performed because of implicit CRA result</td>
</tr>
<tr>
<td>EFAULT</td>
<td>Miscellaneous</td>
</tr>
</tbody>
</table>
This chapter 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 increasing complexity of network systems.
- The large and growing number of protocols and standards.
- The increasing number of possible combinations of services and applications created and used on networks.
- The administrator troubleshooting is usually far removed from those who best understand the network, products, and systems.

The person troubleshooting needs knowledgeable support tools to address these issues. 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 that capture 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.

This chapter addresses the following topics:

- “Overview”
- “Using HP-UX Logging and Tracing for Troubleshooting Support” (page 355)
- “Passing Data to HP-UX Tracing and Logging” (page 358)
- “Formatting Networking Trace and Log Messages” (page 364)

Overview

HP-UX network tracing and logging facilities provide the following general features:

- Record log events and trace data
- Determine what information to capture
- Select and format the recorded information
- User interface commands that do the following:
  - Configure, start, and stop the trace and log services.
  - Format captured messages.

The commands and other HP-UX network tracing and logging facilities (for example, files and subroutines) described in the following sections provide a programmatic interface that enable user and kernel routines to access the following services:

- nettlconf command
  - nettlgen.conf subsystem configuration database
  - nettl 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
- nktl_daemon daemon
Figure 14-1 shows the data flow of the elements in the HP-UX trace and log system.

The nettlgen.conf File

The nettlgen.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. To obtain this subsystem ID from HP, see “Assigning a Subsystem ID” (page 355). For more information, see nettlgen.conf(4).

The nettlconf Command

The nettlconf command creates and updates the /etc/nettlgen.conf database file, 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. See nettlconf(1M), nettl(1M), and netfmt(1M) and the HP-UX 11i v3 Driver Development Reference.

Information such as the subsystem name, library name, and subformatter function is given to the nettlconf command, which stores it in the /etc/nettlgen.conf configuration file. Use this command in the configure script of the subsystem module during system install or 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 at product installation time.

The nettl Command

The nettl 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 starts the nktl_daemon and ntl_reader daemons. For more information, see nettl(1M).

The netfmt Command

The netfmt command formats binary trace and log data into readable ASCII text. This command controls cost-filtering of the data.

1. The netfmt command uses subsystem configuration information to identify shared libraries provided by subsystems, which 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 netfmt command calls the functions of subsystems for which it has data.
4. The netfmt 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 interprets and stores 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 does not format the record if the values in the record match the values specified in the filters. The subsystem formats the record according to the format options specified: for example, nice, terse, and raw. For more information, see netfmt(1M).

Using HP-UX Logging and Tracing for Troubleshooting Support

Follow these guidelines to design user friendly tracing and logging facilities to solve the troubleshooting problems:

• Log only what is needed to solve problems.
• Record all information to diagnose the problem in the log.
• Provide a hex dump to the administrator as the last resort.
• Make each product do as much self-diagnosis and repair as possible. Notify the administrator only when intervention is required or requested.
• Give the administrator the necessary problem-solving information.

The following information helps set up tracing and logging to support troubleshooting:

• “Assigning a Subsystem ID”
• “Classifying Trace Data” (page 356)
• “Formatting Trace Data” (page 356)
• “Classifying Log Data” (page 357)
• “What and When to Log” (page 357)

Assigning a Subsystem ID

Each networking product requires its own unique subsystem ID number, which must be assigned by the HP NetTL team.
To request a unique subsystem ID for your product, email HP 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. Your name will be assigned if it is not already in use. Do not request names such as lan, lo, ni, x25, and others 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.

**NOTE:** Use the file received from HP as /usr/include/sys/subsys_id.h in the HP-UX Device Driver Development system when compiling the networking device driver.

### Classifying Trace Data

Tracing can capture or make snapshots of loopback or header information, and 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. 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 retransmitted data.

Traces are defined as follows:

- **PDU**
  - Inbound and outbound Protocol Data Units (PDU) (including header and data).

- **Header**
  - Inbound and outbound protocol headers.

- **Loopback**
  - Trace packets emanating and returning to the same system.

- **Procedure**
  - Trace entry and exit from all procedures.

- **Error**
  - Invalid state transitions, invalid PDUs, 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 when recording information about normal state transitions.

- **Connection**
  - Information about connections as they are made and destroyed.

- **Logging Trace**
  - A trace that contains a log message. This trace helps troubleshooting by locating and synchronizing logging and tracing output.

### Formatting Trace Data

Troubleshooters must trace both incoming and outgoing data through the stack. The trace records from different processes must be threaded together to form a complete record of the path the PDU takes from the user application out the wire, and vice versa.
Use the following guidelines when implementing tracing routines:

- Each process must trace incoming and outgoing data from both top and bottom. Alternatively, each protocol can 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 parts, leaving the upper layers to the application processes.

**Classifying Log Data**

Logging captures and records specific network activities and infrequent significant network events, such as state changes, errors, and connection establishment. The purpose of logging is to inform the system administrator about these significant events and to create a permanent record for later interrogation. Typical log messages are about errors (catastrophic, recoverable and non-recoverable), warnings (major and minor), or systemwide information (changes to configuration or operation).

The following events are logged:

- **Disaster**
  - Signals that the software detected a severe and irrecoverable error condition that 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 states that an action generated by one process can damage other processes.

- **Error**
  - Signals an event or condition that, though 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 correctly but the underlying networking subsystem recovered. For example, an error 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 pointer alignment problems or data being accessed that was not initialized.

- **Informative**
  - Describes important infrequent operations and current system activities, such as protocol module initiation and termination.

**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 administrator sees on the system console following a network operation event. Deciding what to log and when to log involves trade-offs between usability, performance, schedule constraints, and management and peer pressure. Other than the items outlined in the preceding tracing or logging sections, guidelines for logging are as follows:

- If an event results or causes the product or system to be unusable by all users, log it as a Disaster.
- If an event affects a single application, log it as an Error.
- If an event can cause an error or disaster in the future or cause performance degradations, log it as a Warning.
- If an event occurs infrequently and is something the user might need to know about, but will not cause future problems, log it as Informative.
- If an event occurs frequently, it is not appropriate to log it. Trace it instead. Do not use Informative log messages in place of tracing. See “Passing Data to HP-UX Tracing and Logging” (page 358).
• Do not log \textit{Me Too!} messages as Errors or Disasters. These events occur in response to an error or disaster event in another place, but are not themselves a disaster or error. \textit{Me Too!} messages provide no additional information to solve the problem.

• Do not acquire a new log instance if one is already available for the 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 administrator troubleshooting must know what happened, what caused it, and how 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, state the appropriate manual in which to take further steps or gather more information 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 and to facilitate communication with HP support groups.

• Restrict logged information to only a few well defined types; for example, an event number, a bounded array, or a string.

• Identify error recovery procedures for Disaster and Error class events.

• Emphasize understanding and documenting the previous procedures. After completing error recovery procedures for these events, focus on Informative and Warning class events only if they are useful.

### Passing Data to HP-UX Tracing and Logging

Kernel subsystems that use the trace and log services must include the following statements in their source files and makefiles:

```c
#include <net_diag.h>  // Contains macro calls that check whether tracing and logging is enabled for the subsystem.
#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 Macro

This macro traces on an all interface device basis. It enables the calling process to verify whether tracing is enabled for the current subsystem. The returned value is one (1) if tracing is enabled. It is defined as follows:

```c
KTRC_CK(
    subsys_id,
    trace_kind
) ;
```

Where:

- `subsys_id` Unique subsystem ID of the calling subsystem assigned by HP. See “Assigning a Subsystem ID” (page 355).
- `trace_kind` Specifies the kind of trace. All kinds are defined in the `subsys_id.h` header file and listed in “Classifying Trace Data” (page 356), 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 and 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:  See the net_diag.h header file for aliases or redefinitions of the trace_kind parameter values.

For example:

#define TR_LINK_LOOP LOOP_BACK_BIT
#define TR_LINK_INBOUND PDU_IN_BIT
#define TR_LINK_OUTBOUND PDU_OUT_BIT

A driver named enet.c uses this macro as follows:

if (KTRC_CK(ENET_ID, TR_LINK_INBOUND))
{
  ktrc_write(...);
}

The ktrc_write Routine

This routine sends 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 follows:

ktrc_write(
    int subsys_id,
    int trace_kind,
    int path_id,
    int device_id,
    caddr_t tl_packet,
    int tl_packet_cnt
);  

Where:

subsys_id  Unique subsystem ID of the calling subsystem assigned by HP. See “Assigning a Subsystem ID” (page 355).

trace_kind  Defines the kind of trace. All kinds are defined in the header file subsys_id.h. Following are the defined trace kind values. These values can be ORed to produce combinations of trace kinds. For more information, see “Classifying Trace Data” (page 356).

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 and 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
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>path_id</td>
<td>The connection path on the host. If this is a nonapplicable parameter, pass in -1.</td>
</tr>
<tr>
<td>device_id</td>
<td>Device ID number (for example, if_unit) of the calling subsystem message. If this is a nonapplicable parameter, pass in -1.</td>
</tr>
<tr>
<td>tl_packet</td>
<td>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 passes 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.</td>
</tr>
<tr>
<td>tl_packet_cnt</td>
<td>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.</td>
</tr>
</tbody>
</table>

As with logging, encapsulate tracing calls in functions or macros. The code snippet in the following section shows a typical use of tracing calls.

### Tracing Code Example

The following example shows a trace of an outbound packet with parts located in distinct memory locations. The trace uses the vectored data capability of the ktrc_write call.

```c
#include "../h/netdiag1.h"
#include "../h/net_diag.h"
#include "../h/subsys_id.h"
#include "../h/uio.h"
#define MAX_BUF 3  /* Any number of vectors are allowed. */
#define TRACE   0
#define FALSE   0

int trace_pdu_out(pdu_hdr, pdu_hdr_len, pdu_data, pdu_data_len)
char *pdu_hdr;
int pdu_hdr_len;
char *pdu_data;
int pdu_data_len;
{
  int kind;
  int device_id;
  int path_id;
  short subsys_id;
  struct iovec tl_buf[MAX_BUF];
  int tl_buf_cnt;

  /*
  * Set up variables for KTRC_CHECK()
  */
  subsys_id = MY_SUBSYS_ID_NUMBER; /* Assigned by HP. */
  kind = PDU_OUT_BIT;
  device_id = -1;  /* -1 means not applicable. */
  path_id = -1;    /* -1 means not applicable. */

  if (KTRC_CHECK(subsys_id, kind, device_id)) {
    /*
    * Tracing is enabled for this subsystem
    */
```

360 Tracing and Logging in LAN Drivers
* and kind combination.
*/

tl_buf[0].bufptr = pdu_hdr;
tl_buf[0].buflen = pdu_hdr_len

tl_buf[1].bufptr = pdu_data;
tl_buf[1].buflen = pdu_data_len;
tl_buf[2].bufptr = NULL;
tl_buf[2].buflen = 0;
tl_buf_cnt = 2;

ktrc_write(subsys_id,
    kind,
    path_id,
    device_id,
    &tl_buf,
    tl_buf_cnt);
}

return(0);
}

KLOG_CK Macro

This macro enables the calling process to find out whether logging is enabled for the current subsystem. The returned value is one (1) if logging is enabled. The macro is defined as follows:

KLOG_CK(
    subsys_id,
    log_class
);

Where:

subsys_id Unique ID number of the calling subsystem assigned by HP.

log_class Defines the event classification. All classes are defined in the header file subsys_id.h. See also “Classifying Trace Data” (page 356). Four classes are defined for logging messages:

Informative Normal operation messages only
Warning Warning messages
Error Error condition messages
Disaster Critical error messages

The kget_log_instance Routine

This call accepts no parameters, but returns a unique log instance value. The log instance helps thread log messages together so the administrator 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 obtains a log instance value. Otherwise, the module uses the current log instance value it was passed without calling kget_log_instance. For information on log instance values, see “The klogg_write Routine.”.

The klogg_write Routine

This routine sends log messages to the kernel trace and log facility.

Prefiltering is done during the log call, and unwanted messages are dropped. This routine always returns a success of zero. It is defined as follows:

klogg_write(}
int subsys_id,
int class,
int device_id,
int log_instance,
caddr_t tl_packet,
int tl_packet_cnt
);

Where:

subsys_id  Unique ID of the calling subsystem assigned by HP.
class  Defines the event classification. All classes are defined in the header file subsys_id.h. See also “Classifying Trace Data” (page 356). Four classes are defined for log messages:
  Informative Normal operation 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 that identifies the thread of events attending an interface. 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 passes 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, so the calling routine need not copy the data and then pass a pointer to the data.

tl_packet_cnt  If -1, tl_packet points to an mbuf chain. If the value is greater than zero, it is the number of the iovec structure (as defined in uio.h) the tl_packet points to.

Logging Code Example

The following example describes the intrinsic calls of HP-UX logging facilities. These are code snippets that a subsystem might include to perform logging calls.

#include "../h/netdiag1.h"
#include "../h/net_diag.h"
#include "../h/subsys_id.h"

#define MAX_BUF 1 /* Any number of vectors are 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 to call 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 is enabled for this subsystem
     * and class combination.
     */
    if (log_instance == 0) {
        /* No previous log instance was
         * associated with this event. This is
         * the first module to encounter the
         * problem, so it gets the log instance.
         * Log instance must be available to
         * all modules in the subsystem and to
         * other subsystems.
         */
        log_instance = kget_log_instance();
    }

    event_data.event_number = THIS_EVENT_NUMBER;
    event_data.event_type   = THIS_EVENT_TYPE;
    /* Additional data about the event can be
     * placed in the data structure. You
     * must design this data structure. The
     * subformatter for this subsystem must
     * be able to decode the data structure,
     * but there are no restrictions on what
     * gets passed. You can use a single
     * mbuf chain to hold all the event
     * information, or pass a vectored
     * buffer to the klogg_write() call to
     * hold individual pieces of information.
     * *
     * Callers must not pass strings in this
     * function; the event number as shown in
     * this example must be used to
     * generate an NLS string from a message
     * catalog in the subformatter.
     */
    tl_buf[0].bufptr = *event_data;
    tl_buf[0].buflen = sizeof(event_data_type);
    tl_buf[1].bufptr = NULL;
    tl_buf[1].buflen = 0;
    tl_buf_cnt = 1;

    klogg_write(subsys_id,
                class,
                device_id,
Formatting Networking Trace and Log Messages

The following sections describe facilities and network device driver developer's responsibilities for formatting trace and log output. The netfmt formatter is the HP-UX tool that presents trace and log information in human-readable form. It includes the following components:

Subformatter
Functions provided by the tracing and logging driver subsystem to interpret the subsystem messages and produce human-readable output.

Formatter Core
Handles files, filters globals, and dispatches messages to the appropriate subsystem formatter. The netfmt formatter also provides generic formatting utility functions (listed in “Formatting Routines” (page 365)) help you develop a subformatter and achieve consistency in output by various subsystems.

For the interaction among the various components, see Figure 14-1 (page 354).

Subformatter Registration with NetTL

The subsystem that provides the subformatter registers the relevant information with NetTL in the configuration database, for example nettlgen.conf file (see nettlgen.conf(4)) using the nettlconf command (see nettlconf(1M)). For details on the required information, see “Configuring Subsystems into the System” (page 372).

For example, the subformatter for subsystem \textit{N} includes the following functions:

\begin{description}
  \item[subsys\_\textit{N}\_format] Interprets the trace and log message and presents it in a human-readable form. For more information, see subsys\_\textit{N}\_format(9F) in the \textit{HP-UX 11i v3 Driver Development Reference}.
  \item[subsys\_\textit{N}\_get\_options] Performs any subsystem-specific filtering of its trace and log message. The formatter core provides some common filtering capabilities (see netfmt(1M)) that are applicable to more than one subsystem. For more information, see subsys\_\textit{N}\_get\_options(9F).
\end{description}

These functions must be provided in a shared library, typically named \texttt{lib}\textit{N}fmt.so. The netfmt formatter uses the subsystem configuration information in the nettlgen.conf database file to identify the shared library that contains the listed functions. The netfmt formatter 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 and Features

The netfmt formatter core responsibilities and capabilities are as follows:

\begin{itemize}
  \item Loads subformatter libraries using the subsystem configuration information in the nettlgen.conf database file.
  \item Processes filtering and formatting options.
    The formatter determines filtering and formatting options by processing an auxiliary file.
  \item Handles global filtering.
    The formatter checks each message against the filters that are set up. Only a message that comes past the filter is sent to its 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 on. It prints an informative message, formats the message header, and discards the remainder of the data. It then 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 “Trace Formatting Routines” (page 366).

- Provides various modes of formatting.

The formatter provides the following three modes of formatting for trace data:

**Terse (one-line)** The least descriptive formatting mode used to get a summary of trace file contents in a single line. Additional options and flags control the behavior of terse formatting.

**Nice (detailed)** The most detailed level of formatting available. It attempts to decode every piece of data in the trace.

**Raw (hex-dump)** The default mode of formatting. It prints the trace data in hex format.

The formatter conveys the active formatting mode by enabling the appropriate flags when calling subformatter functions.

### Formatting Routines

The formatter provides two types of formatting routines you can use:

- **Utility routines** Help the subformatter obtain required information and display output of the trace and log message in a consistent manner.

- **Trace formatting routines** Help a driver subformatter completely format trace data by handling the link header information only, not including 802.2 information.

  By using the common functions, the underlying implementation of I/O can be changed easily. This enables easier porting, further performance enhancements, alternative output schemes, and so on.

#### Utility Routines

Table 14-1 lists the utility routines provided by the formatter. For exact syntax and further details, see each routine's manpage. The following functions reside in the `libfmtutil.so` library. Subsystems must not link to this library. All externals are resolved during dynamic loading at runtime.

<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 <code>tl_header_format</code> 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 <code>printf</code>.</td>
</tr>
</tbody>
</table>
### Table 14-1 Utility Routines (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tl_format_write</code></td>
<td>Writes the decoded buffer to stdout.</td>
</tr>
<tr>
<td><code>tl_get_line</code></td>
<td>Obtains a line from the filter file. This function is called by the subsystem's <code>subsys_N_get_options</code> function.</td>
</tr>
<tr>
<td><code>tl_get parms</code></td>
<td>Returns to the caller (typically the subformatter) a <code>ss_N_fmt_parms_type</code> data structure (see <code>ss_N_fmt_parms_type(9F)</code>), which contains all the information that a subformatter needs to operate. This function can be called anywhere within the subformatter.</td>
</tr>
<tr>
<td><code>tl_header_format1</code></td>
<td>Formats a single trace and log header. The format of the output conforms 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><code>tl_log_class</code></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 <code>tl_header_format1</code> function when printing the header.</td>
</tr>
<tr>
<td><code>tl_raw_format</code></td>
<td>Formats a trace or log message into both hexadecimal and printable ASCII characters. This function is typically used by the subformatter when it cannot further decode the information in the trace or log message.</td>
</tr>
<tr>
<td><code>tl_trace_kind</code></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 <code>tl_header_format1</code> function when printing the header.</td>
</tr>
</tbody>
</table>

### Trace Formatting Routines

The formatter core formats information from upper layer protocols such as IP, TCP, UDP, ARP, DUX, and NFS from traces taken at link layer. This makes it easier to analyze networking dialogs than examining raw hex data and manually determining what the protocols were sending.

In addition, the formatter filters the trace output; for example, to display only dialogs taking place with a TCP port. The filters include Ethernet type, 802.2 SAPS, IP addresses, UDP ports, and RFC information. For a complete list of filters supported, see `netfmt(1M)`.

This section describes the `netfmt` formatter routines of which new link products must take advantage. Link subformatters can use the following decoding routines to format link level packets:

- **set_up_***xxx***: Set up global protocol information for the traced packet, to be used later by the filtering and formatting routines. The *xxx* value can be 8022, link, ether, or ip.
- **filter_packet**: 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 in effect. The *xxx* value can be 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. The trace subformatter does not directly perform I/O, which is performed through the three formatting routines provided. These routines enable future changes to be made to the look of the formatted output without modifying the link subformatter code.

Table 14-2 lists each of the trace formatting routines provided by the formatter core. For exact syntax and further details, see each routine's manpage. The following functions reside in the `libnsfmt.so` library. Subsystems must not link to this library. All externals are resolved during dynamic loading at runtime.
Table 14-2 Trace Formatting Routines

<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 and 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 and PDU_OUT Ethernet packet.</td>
</tr>
<tr>
<td>set_up_link</td>
<td>Sets up global information for the data link layer only 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 cannot use this routine. Use it only when no link information is available (for example, when formatting a 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. Returns 0 if the packet is not to be displayed.</td>
</tr>
<tr>
<td>format_link_nice</td>
<td>Formats a packet to display upper layer information in detail. The subformatter calls this routine when nice format is enabled.</td>
</tr>
<tr>
<td>format_link_terse</td>
<td>Formats a packet to display upper layer information in a single line. The subformatter calls 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 calls this routine when raw format is enabled.</td>
</tr>
</tbody>
</table>

Examples

The following code snippet ow the enet subformatter handles trace data.

```c
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 the 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);
        /* Now extract MAC information from the data */
        trace_buffer = (struct ieee8023_hdr *)data_ptr;
        /* Make data_ptr point to the beginning of the 802.2 header.
```
* data_len is the length of buffer to which data_ptr points.
*/
if (trace_buffer->length < MIN_ETHER_TYPE) { /* IEEE packet */
    ether_type = 0;
    /* Both IEEE and ETHER header are 14 bytes long */
    data= data_ptr + 14;
    /* Point to LLC */
    /* Call setup routine to set up structures reflecting the 802.2 */
    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 */
    set_up_ether(data_ptr, data_len, trace_buffer->destaddr,
                 trace_buffer->sourceaddr, trace_buffer->length);
}
/* filter_packet() indicates whether the current packet meets the */
/* user-specified filter criteria. This routine uses the global info */
/* set up by set_up_8022/set_up_ether routines (IP address, */
/* Ether type, TCP port, and so on.) */
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 the 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() and format_link_raw() routines. If there */
/* is more information to pass about the link header, pass it as the */
/* iaddl_infoi parameter to the format_link_nice() and */
/* 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", "", "", ""))
            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 examples show the output for each format mode.

**Nice Format Output**

[The linktype parameter goes here]-vvvvv

---------------------- 802.3 ----------------------

[The line1 parameter goes here]-vvvvvvvvvvvvv

Source : 00-00-0c-00-06-31 [I] [Cisco ] LENGTH: 26
Dest   : 09-00-09-00-00-01 [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
Designing a Subformatter

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 can also provide filtering or formatting options.

This section describes the design of the function that is called in response to the formatter reading in a record containing data for a specific subsystem.

The subformatter must 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 subsystem ID. Formatting requirements for tracing are often different from logging. You must 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. Design the subformatter in relation to the types of information that come from the subsystem for logging. Providing minimal information, such as the logging event and a couple of data items, can 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 used by the ARPA logging subsystems.

Tracing information contains much more data, especially in the case of link-level packet tracing (PDU in or out tracing). The subformatter might need to know how the packet was constructed, which layer sits on top, and so on.

For tracing and logging, the subsystem must pass as little data as possible to output complete, useful information. The formatter passes flags to the subformatter that can control the format and degree of detail of the output.
Subformatter Responsibilities

Beyond transforming the data, the subformatter has the following responsibilities:

1. Performs 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 parses and determines 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 lowercase. The restrictions on the contents of the line are that it cannot exceed 2048 bytes, and it must contain only printable characters. The `tl_get_line` function must always be used to read options lines from the file.

   Subsystems can use this feature to alter the level of information (beyond terse and verbose), to include extra kinds of data, to provide extra filtering (for events or trace and log data that are not covered by the formatter’s global filtering functions), and so on.

2. Prints the header.

3. If console logging is on, formats a terse message. Otherwise, formats a message in accordance with the format flags.

4. Writes the formatted message.

   These tasks can be performed with the help of the utility functions listed in Table 14-1 (page 365).

Subformatter Requirements

The subformatter has the following requirements:

1. As mentioned in “Subformatter Registration with NetTL” (page 364), the `subsys_N_format` and optionally `subsys_N_get_options` routines must be supplied by the subsystem for the formatter to call when subsystem-specific actions need take place. Subsystem-specific actions include parsing filter files and formatting the subsystem’s trace or log data.

2. To use the generic formatter utility routines, the subformatter must 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.

Building and Installing the Subformatter

The shared library, typically named `libNfmt.so`, contains the subformatter routines created by compiling all modules with `cc +z` and linking them together using the options to `ld`.

```
-b +e subsys_N_format +e subsys_N_get_options
```

The `ld +e` command must be used to prevent symbol collisions among the different subformatter libraries.

The default path for the subformatter shared library is `/usr/lib/hpux32`. If the subformatter library is placed under a different directory, you must specify the absolute path in the `nettlgen.conf` configuration database.
Internationalization and Message Catalog Support

The formatter enables the subformatters 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 using the catopen and catclose calls, respectively.
2. The file descriptor returned by 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 calls catgets to retrieve messages from the message catalog. Subsystems must not open their own message catalogs or use multiple message catalogs.

The accepted way to use message catalogs is to use the catgets call and provide the English language string as the default to the call if the message catalog read fails. The call retrieves the same string from the default message catalog, typically located in /usr/lib/nls/msg/C/name.cat. The value for name is the name of the subsystem message catalog registered in the nettlgen.conf database.

To use message catalogs effectively, make 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 more information, see the ENET driver’s subformatter code in the DDK.

Because message catalogs can be altered for a given location, the subformatter must also include an identifying tag (such as FTAM 489) on the message that is not localized. Support personnel in a different location can then understand what is being logged without translating the text of the message.

Because the subformatter depends on message catalogs to provide the correct text for a log event, the version of the catalog is dependent on the version of the subsystem. Use the tl_check_cat_version function to check message catalog versions. See tl_check_cat_version(9F) in the HP-UX 11i v3 Driver Development Reference.

Configuring Subsystems into the System

Informing the tracing and logging facility about developed subsystems is complex. Subsystems must inform the tracing and logging facility of their existence at installation or update time.

Each fileset must have a Software Depot (SD) configure script. Tracing and logging use this independence to facilitate the configuration of subsystems into the nettl and netfmt commands.

The nettlconf script can configure the subsystem. See nettlconf(1M). Call nettlconf from within the configuration script during an SD update or installation. The nettlconf script configures the subsystem information and adds it to 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 the subsystems must configure include the following:

<table>
<thead>
<tr>
<th>Subsystem ID</th>
<th>Assigned to the subsystem by HP. See “Assigning a Subsystem ID” (page 355).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subsystem Mnemonic</td>
<td>The name by which the subsystem is identified in nettl and in the formatted header printed by netfmt. This string</td>
</tr>
</tbody>
</table>

372 Tracing and Logging in LAN Drivers
can consist of alphanumerics (beginning with a letter) and can contain underscores. Spaces are not allowed.

**Default Logging Class**
A mask containing the level of logging to be enabled when the logging facility starts. This level can be changed by subsequent calls to `nettl`.

**Subsystem Space Type**
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**
The subsystem formatter library that contains the subformatter functions.

**Subsystem Formatting Function**
The C function name used to call the function that supports formatting for the subsystem. This function must reside in the subsystem formatter shared library.

**Subsystem Options Function**
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 reside in the subsystem formatter shared library.

**Subsystem Group Name**
The name of the logical group to which the subsystem belongs, usually a product. This group name is included on the banner printed during formatting. Although this group name can be any ASCII string, it must contain the subsystem product name.

**Subsystem Formatter Message Catalog**
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`. It resides in the default NLS directory, `/usr/lib/nls/msg/C`. This can be specified 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` is specified as `/opt/xyz/lib/nls/%L/abc.cat`. 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 shipped message catalogs).

### Subformatter Configuration Script Example
The file set configuration script performs the configuration of all subsystems contained in the file set. The following code snippet shows an SDU control script to perform the configuration for example subsystems A, B, and C.

```bash
#!/usr/bin/posix/sh

###
# Product:
# Fileset: NETTL-MIN
```

Formatting Networking Trace and Log Messages 373
# configure
# @(#) $Revision: 1.1 $
#
# (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

. $UTILS
set_env

: ${FILESET:="NETTL-MIN"}
: ${NETTLCONF:="${SW_ROOT_DIRECTORY}usr/sbin/nettlconf"}
: ${NETFMT:="${SW_ROOT_DIRECTORY}usr/sbin/netfmt"}
if [ ! -x "$NETTLCONF" ] ; then
    echo "ERROR: Cannot find $NETTLCONF"
    exit 1
fi

# Subsystem A
$NETTLCONF -S -id 0 -name SUBSYSTEM_A -class 12 -kernel
  -lib libsubsystem_A.so -msg subsys_A_msg
  -fmtfn subsys_A_format -optfn subsys_A_get_options
  -group "SUBSYSTEM A Product"
  ||
exit 1 # nettlconf reports its own errors

# Subsystem B
$NETTLCONF -S -id 0 -name SUBSYSTEM_B -class 12 -kernel
  -lib libsubsystem_B.so -msg subsys_B_msg
  -fmtfn subsys_B_format -optfn subsys_B_get_options
  -group "SUBSYSTEM B Product"
  ||
exit 1 # nettlconf reports its own errors

# Subsystem C
$NETTLCONF -S -id 0 -name SUBSYSTEM_C -class 12 -kernel
  -lib libsubsystem_C.so -msg subsys_C_msg
  -fmtfn subsys_C_format -optfn subsys_C_get_options
  -group "SUBSYSTEM C Product"
  ||
exit 1 # nettlconf reports its own errors

... Other subsystem configurations

# Test the configuration file
cmd_output=`$NETFMT -pc /dev/null 2>&1`
cmd_result=$?
if [ $cmd_result -ne 0 ]
then
    # The configuration file caused an error
    echo "ERROR The $NETFMT command produced following error"
    echo "messages while verifying configuration:"
    echo "$cmd_output"
    exit 1
fi

exit

374 Tracing and Logging in LAN Drivers
Subsystem Installation Testing

You must perform complete installation testing on your subsystems. As described in the previous section, the network trace and log facility is configured at installation time by a registration process in the subsystems configuration 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 can be turned on for logging and tracing and have their records formatted appropriately.

The nettlconf command does not check the parameters 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 multiuser systems, workstations, and diskless clusters.

The registration scheme can break tracing and logging for all subsystems if the configuration becomes corrupt or if the information is invalid. Subsystems must test and review the procedures used to configure their subsystems into the network trace and log facility.

Testing Procedure

The subsystems perform the following steps in the target system where the driver subsystem is installed to ensure correct installation and configuration:

1. Install the subformatter library in the appropriate directory (default is /usr/lib/hpux32).
2. Configure subsystem information as explained in “Configuring Subsystems into the System” (page 372).
3. Stop and start nettl (see nettl(1M)).
   - Starting nettl must not result in error messages.
   - The nettl -status command displays the appropriate log class enabled for the subsystem as specified in the nettlgen.conf file.
4. Enable tracing, if applicable.
5. Generate log and trace messages.
6. Use the netfmt 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 whether the messages are displayed on the console.

Troubleshooting Installation and Configuration

The following common problems can make the configuration file 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 you use the ld +e option when creating the subformatter library.
- Symbols remain unresolved after netfmt has loaded the subformatter libraries for 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 HP as described in “Assigning a Subsystem ID” (page 355).
This chapter explains the interaction between the transport layer and the DLS providers to create an efficient data transfer mechanism between layers in a networking stack. To explain the interactions clearly, HP-DLPI is used as an example of a DLS provider that supports LAN class drivers. Other DLS providers support interface classes such as ATM, X.25, and HyperFabric.

Any DLS provider must adhere to the overall behavior of HP-DLPI when interacting with the transport layer. This includes the following behavior of DLPI 2.0 standard and HP-specific extensions when interacting with transport layer. In HP-UX 11i v2, the format of the data passed between the transport layer and the DLS providers was modified. Therefore, STREAMS modules pushed on a stream between transport (for example, IP) and DLS providers must change to function correctly. This includes any Native STREAMS DLPI network drivers and STREAMS modules ported from HP-UX 11i v1 to HP-UX 11i v3. The changes required for the STREAMS modules is discussed in this chapter.

This chapter addresses the following topics:

- “DLPI IOCTL and Primitives”
- “OOP Data” (page 379)

**DLPI IOCTL and Primitives**

HP-DLPI defines a set of *ioctls* for the in-kernel STREAMS DLS user (for example, IP module) to negotiate the driver features and set up a Fastpath. However, the information provided to the DLS user changes for HP-UX 11i v3. This interface was originally intended only for IP, but became DLS user-independent starting with HP-UX 11i v2.

**DL_IOC_DRIVER_OPTIONS**

IP (or any other in-kernel DLS user) uses the `DL_IOC_DRIVER_OPTIONS` ioctl to obtain additional information from DLPI regarding the drivers. HP-DLPI informs the DLS user of the special capabilities, which the DLS user can use to enhance performance. As a working example, the TCP/IP module is used to explain how these ioctls work.

While configuring an interface for IP, the IP sends the HP-DLPI an `M_IOCTL` message. The `M_IOCTL` message block contains an `iocblk` structure with `ioc_cmd` set to `DL_IOC_DRIVER_OPTIONS` and `ioc_count` set to (size of `driver_ops_t`). Following the `M_IOCTL` message block, pointed by `b_cont` of the `M_IOCTL` message block is an `M_DATA` message block containing a `driver_ops_t` structure.

Table 15-1 provides information on the structure.

<table>
<thead>
<tr>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>driver_ops_t</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Include File</th>
<th>&lt;sys/dlpi_ext.h&gt;</th>
</tr>
</thead>
</table>

| Purpose                  | DLPI/XPORT options negotiations structure. This structure is passed in as part of the `DL_IOC_DRIVER_OPTIONS` ioctl from transport. |

<table>
<thead>
<tr>
<th>Members</th>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>driver_ops_type</code></td>
<td><code>uint32_t</code></td>
<td>Features supported by transport. The features are bitwise flags.</td>
</tr>
<tr>
<td></td>
<td><code>driver_ops_type_1</code></td>
<td><code>uint32_t</code></td>
<td>Reserved. Must be set to 0 (zero).</td>
</tr>
<tr>
<td></td>
<td><code>driver_ops_type_2</code></td>
<td><code>uint32_t</code></td>
<td>Reserved. Must be set to 0 (zero).</td>
</tr>
</tbody>
</table>

Table 12-6 (page 314) provides information on features supported by the transport layer.
The transport layer uses the `driver_ops_type` field to pass a bitmask of flags indicating the driver capabilities that it is asking about. HP-DLPI looks at the bitmask passed by the transport layer and compares the flags with the driver features. The negotiated features are set in the `driver_ops_type` field of the `driver_ops_t` structure originally sent by transport (such as IP). For example, `DRIVER_CKO_IN` can be set by DLPI to say that the driver supports inbound the CKO feature. See Table 12-6 (page 314) for information on other features. Flags of all the features that are not supported or understood by the DLS provider must be disabled in the `driver_ops_type` element.

Some of the options negotiated using this ioctl can require communication of additional information in the data path. The additional information is called Out-Of-Packet (OOP) data. The OOP data precedes the packet payload and is used during the packet processing.

**DL_IOC_HDR_INFO ioctl**

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 it 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. After the connection is established, many connection-oriented protocols such as TCP have identical information in every `DL_UNITDATA_REQ`. To avoid constructing a packet header every time, a DLS provider can support an ioctl that permits the transport layer to request a packet header template during configuration of TCP/IP on a per-interface basis. The packet header template enables that transport layer to add the packet header before sending the packet to a DLS provider. This mechanism is called Fastpath.

Fastpath is negotiated when the transport layer sends an `M_IOCTL` type message with `ioc_cmd` set to `DL_IOC_HDR_INFO`. The `b_cont` field of the `M_IOCTL` message block links to an `M_DATA` type message block that contains a `dl_unitdata_req_t` structure, which is used to create the `DL_UNITDATA_REQ` primitive. Figure 15-1 shows the Fastpath negotiation request message format.

**Figure 15-1 Fastpath Negotiation Request Message**

```
<table>
<thead>
<tr>
<th>M_IOCTL</th>
<th>b_cont</th>
<th>M_DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL_IOC_HDR_INFO</td>
<td></td>
<td>DL_UNITDATA_REQ</td>
</tr>
</tbody>
</table>
```

For a driver that does not support Fastpath, HP-DLPI returns a `M_IOCNAK` message. In this case, the transport layer must use only the `DL_UNITDATA_REQ` primitive for data transfer.

A Native STREAMS DLPI network interface driver must implement this ioctl to support Fastpath.

In response to the `DL_IOC_HDR_INFO` ioctl, a DLS provider returns the packet header template, which contains the OOP data template and the link level header template with information filled in. The response is an `M_IOCACK` ioctl as shown in Figure 15-2. The second message block contains the `DL_UNITDATA_REQ` passed to DLS provider. The third message block contains an `M_DATA` message block with the Link Layer Header template created for that particular stream.

**Figure 15-2 Fastpath Negotiation Acknowledgement Message**

```
<table>
<thead>
<tr>
<th>M_IOCACK</th>
<th>b_cont</th>
<th>M_DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL_IOC_HDR_INFO</td>
<td></td>
<td>DL_UNITDATA_REQ</td>
</tr>
<tr>
<td>OOP header (if any) + Link Layer Header</td>
<td>b_cont</td>
<td></td>
</tr>
</tbody>
</table>
```

The transport layers can use the OOP data template part of the packet header template to communicate additional information to be used for packet processing. This information is not
sent out on the wire as-is, but is used by the DLS provider to improve the overall performance of the stack. For more information on the OOP data, see “OOP Data” (page 379).

Other DLS providers must follow the schematics of the packet header negotiations ioctl as done by HP-DLPI.

**DL_HP_NOTIFY_EVENT_REQ Primitive**

Transport layers that negotiate Fastpath must request notification of link DOWN and UP events generated by the physical interface. The physical interface generates link DOWN and UP events when operating parameters such as MAC address and MTU change. The change in interface operating parameters invalidates the negotiated Fastpath OOP and link level packet header. To request notification of link DOWN and UP events, the transport layer must use the DL_HP_NOTIFY_EVENT_REQ primitive. The primitive is contained in an M_PROTO message block with the dl_hp_notify_event_req_t structure. The dl_event field of the dl_hp_notify_event_req_t structure must be set to DL_HP_LINK_STATE_CHANGE. DLS providers must keep track of this and notify the transport layer of the link DOWN and UP events. The specification of this primitive is available in the DLPI Programmer’s Guide. For more information on link DOWN and UP notifications, see “DL_LINK_UP_IND and DL_LINK_DOWN_IND Primitives” (page 379).

**DL_LINK_UP_IND and DL_LINK_DOWN_IND Primitives**

These primitives link state notification indications that must be sent upstream by DLS providers in an M_PROTO message block. The associated structures for these indications are the dl_link_down_ind_t and dl_link_up_ind_t structures, depending on whether it is a DL_LINK_DOWN_IND or DL_LINK_UP_IND indication. These indications are sent whenever the link state of the physical interface changes.

On receipt of the DL_LINK_DOWN_IND, the transport interfaces associated with this interface must be marked as DOWN. When a DL_LINK_UP_IND is received, the transport layer must redo the options negotiations and Fastpath negotiations to get new supported features and the link layer header because these can change.

**OOP Data**

OOP data is additional information that the transport layer and DLS providers can communicate in each outbound and inbound M_DATA packet. During the packet header negotiation, DLS providers create a Link Layer Header template and OOP data template. The OOP data template precedes the Link Layer Header template. Figure 15-3 shows the OOP Data Tuple Header.

**Figure 15-3 Data Tuple**

```
<table>
<thead>
<tr>
<th>Cko_info_t</th>
<th>Type</th>
<th>Size</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Packet Header Template
The OOP data consists of a **OOP data header** and a series of **OOP data tuples**. Every OOP data tuple is a combination of **Type-Size-Data** fields, where **Type** and **Size** are two bytes each. The enumerated type for the OOP data types is defined in `dl_hp_oop_type_t(9S)`. The **Type-Size** part of the OOP data tuple is the header part of each OOP data tuple and is defined as a structure in `dl_hp_oop_hdr_t(9S)`. A specific type of OOP might contain associated data. See `dl_hp_oop_type_t(9S)` for associated data.

All OOP data starts with the `cko_info_t` data structure. This structure is an OOP data header for the OOP data. It also holds CKO information for the transport layer, which interacts with drivers that support CKO. For drivers that do not support CKO, but support other features that require communication of other OOP data, this structure acts as the OOP Data Header.

Following all other OOP data types is a special OOP data type called **End-of-Options** (EOO). The presence of this type means that this is the last OOP data tuple in the OOP data. The EOO type has a valid OOP data tuple header with a **Size** field, which gives the size of the data part of the EOO type. However, the data part of EOO does not contain any valid information.

Not all packet header templates created by DLS providers contain an OOP data template. For example, HP-DLPI creates the OOP data template based on the features that the driver supports in addition to the capabilities of HP-DLPI. If neither the driver nor HP-DLPI supports any features that require OOP data to be passed between the layers, the packet header template includes the link level header template only.

After the packet header negotiation is complete, the transport layer can send out an `M_DATA` packet without the `M_PROTO` message block `DL_UNITDATA_REQ`. All `M_DATA` packets not preceded by `M_PROTO` must use the **Packet Header Template** created by the DLS providers.

During outbound processing for an `M_DATA` `mblk`, the HP DLS provider might modify the contents of the `mblk`, including the OOP area and the MAC/LLC headers. If the upper layers need to keep a reference to this `mblk`, they must use the STREAMS `copyb` or `copymsg` services to create a new copy of the `mblk` contents, including the OOP template. Then they can pass the contents to the HP DLS provider.

---

**CAUTION:**

Do not use the STREAMS `dupb` or `dupmsg` services as they create references to the original `mblk` data. Passing this data by reference makes it vulnerable to being overwritten by the HP DLS provider.

The format of the OOP data is easy to understand and simple to read, and helps the module to extract the payload from the packet. The presence of OOP data is signalled by the presence of the **b_flag** bit `DL_HP_OOP_PRESENT` in the `M_DATA` message block. If the **b_flag** is present, the packet has the leading OOP data.

Each OOP data is a combination of three values:

- **Type**  The type of information present (2 bytes)
- **Size**  The size of the data (2 bytes)
- **Data**  The data present

If the `DL_HP_OOP_PRESENT` flag is set, the first 20 bytes of data serves two purposes. The first 20 bytes of information is of type `cko_info_t`. It contains the offset in the first four bytes (`cko_offset` element of `cko_info_t`) to the beginning of the packet payload. The rest of the 20 bytes contains CKO data. Following the 20 bytes of OOP header is the OOP data. Figure 15-4 shows a sample packet with OOP data.
The \texttt{cko_offset} value for a packet is the offset to the transport layer header. A STREAMS module that must get to the packet payload immediately can use the \texttt{cko_offset} value in the beginning of the data. The \texttt{cko_offset} value added to the \texttt{b_rptr} always points to the transport data payload (for example, for a TCP/IP packet, it points to the IP header) both in the inbound and outbound data transmission. Also, the beginning of the message block data is guaranteed to be 4 bytes aligned. The module can consider the beginning of \texttt{b_rptr} as a \texttt{uint32_t} value without understanding the \texttt{cko_info_t} structure.

**NOTE:** The message block containing the OOP and LLC header can be separate from the packet payload message block.

Packets without OOP data do not have the \texttt{DL_HP_OOP_PRESENT} flag set.

Various OOP types are defined in the \texttt{dl_hp_oop_type_t} structure. None of the OOP types are supported between HP-DLPI and the transport, but the transport layer that does Fastpath negotiation must not assume that the header template created by HP-DLPI contains only MAC and LLC header. It might contain an OOP data template, which is preallocated by HP-DLPI. It can then use it to send OOP data to the driver without allocating an additional buffer in the Fastpath. The structure provides this information though most of the OOP types are valid when packets are transmitted or received by transport. This requirement is valid for any DLS provider on HP-UX.

If a DLS provider supports only CKO, the Packet Header Template must contain the EOO OOP type following the CKO information of the packet.

Table 15-2 provides \texttt{dl_hp_oop_type_t} structure information.

### Table 15-2 Enumerated Type

<table>
<thead>
<tr>
<th>Name</th>
<th>dl_hp_oop_type_t</th>
</tr>
</thead>
<tbody>
<tr>
<td>Include File</td>
<td>&lt;sys/dlpi_ext.h&gt;</td>
</tr>
<tr>
<td>Purpose</td>
<td>The OOP types supported.</td>
</tr>
</tbody>
</table>
Table 15-2 Enumerated Type (continued)

<table>
<thead>
<tr>
<th>Members</th>
<th>Name</th>
<th>Value</th>
<th>Size of Data (bytes)</th>
<th>Associated Data</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL_HP_OOP_PAD</td>
<td>0x0001</td>
<td>Variable</td>
<td>None</td>
<td>Padding Type.</td>
<td></td>
</tr>
<tr>
<td>DL_HP_OOP_CKO</td>
<td>0x0002</td>
<td>Not valid between DLPI and the transport Layer.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DL_HP_OOP_VLAN_TAG</td>
<td>0x0003</td>
<td>4</td>
<td>None</td>
<td>The 2-byte VLAN information in outbound packets.</td>
<td></td>
</tr>
<tr>
<td>DL_HP_OOP_TOS</td>
<td>0x0004</td>
<td>4</td>
<td>None</td>
<td>The 1-byte IP precedence (Type of Service, or ToS) information on inbound packets that a DLS user (for example, IP stack) can use.</td>
<td></td>
</tr>
<tr>
<td>DL_HP_OOP_HDR_LENGTH</td>
<td>0x0005</td>
<td>Not valid between DLPI and the transport Layer.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DL_HP_OOP_EOO</td>
<td>0x00ff</td>
<td>Variable</td>
<td>None</td>
<td>EOO delimiter that separates the OOP and the payload.</td>
<td></td>
</tr>
</tbody>
</table>

During Packet Header Template creation, if DLPI created the OOP data template, all the types have the DL_TYPE_DATA_INVALID bit set. When the transport layer fills an OOP Data Tuple in the outbound path, it must reset the invalid bit before sending the data to DLPI. For more information, see “DL_IOC_HDR_INFO ioctl” (page 378).

The transport layer must not assume that the valid types are always communicated. The presence of OOP data and its validity depends on various factors; for example, whether the driver and transport layer support and understand the features. The transport layer or any STREAMS module must always check the validity (by checking the DL_HP_TYPE_DATA_INVALID bit) of a type of data before using it.

The CKO data does not follow typical OOP Data Tuple format. The presence of a cko_info_t structure in the OOP data passed does not mean it contains valid CKO information. The validity of CKO information is specified by the cko_type element of cko_info_t. If the cko_type is set to CKO_ALGO_NONE, there is no valid CKO information. By default, HP-DLPI sets the value to CKO_ALGO_NONE when creating the header template. In the outbound path, the transport layer sets it to a valid value if CKO information is present. Table 15-3 lists different CKO types.

Table 15-3 CKO Structure

<table>
<thead>
<tr>
<th>Name</th>
<th>cko_info_t</th>
<th>Include File</th>
<th>&lt;net/cko.h&gt;</th>
<th>Purpose</th>
<th>Structure that contains the CKO information.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Members</th>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>cko_offset</td>
<td>uint</td>
<td>Offset to the packet payload.</td>
</tr>
<tr>
<td></td>
<td>cko_type</td>
<td>uint</td>
<td>CKO types supported.</td>
</tr>
<tr>
<td></td>
<td>cko_sum</td>
<td>uint</td>
<td>Checksum.</td>
</tr>
<tr>
<td></td>
<td>cko_start</td>
<td>uint</td>
<td>Start calculating the checksum from this offset.</td>
</tr>
<tr>
<td></td>
<td>cko_insert</td>
<td>uint</td>
<td>Insert point into the packet.</td>
</tr>
</tbody>
</table>

Table 15-4 lists the CKO types.

382 OOP and Transport Ioctls
Table 15-4 CKO Types

<table>
<thead>
<tr>
<th>CKO Type in &lt;net/cko.h&gt;</th>
<th>Value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CKO_INSERT</td>
<td>0x00000001</td>
<td>IP requests that the driver insert the checksum in the packet.</td>
</tr>
</tbody>
</table>
| CKO_ALGO_FAILED        | 0x00000002 | Checksum verification failed in the device for this packet. In addition to this flag, one or more of the following flags must be set:  
  - CKO_ALGO_TCP  
  - CKO_ALGO_UDP  
  - CKO_ALGO_IP |
| CKO_ALGO_PASSED        | 0x00000004 | Checksum verification passed in the device for this packet. In addition to this flag, one or more of the following flags must be set:  
  - CKO_ALGO_TCP  
  - CKO_ALGO_UDP  
  - CKO_ALGO_IP |
| CKO_ALGO_NONE          | 0x00000008 | Checksum is not present.                                      |
| CKO_ALGO_OTHER         | 0x00000010 | Reserved.                                                     |
| CKO_ALGO_UDP           | 0x00000020 | UDP checksum.                                                |
| CKO_ALGO_TCP           | 0x00000040 | TCP checksum.                                                |
| CKO_ALGO_IP            | 0x00000080 | IP checksum.                                                 |

As shown in Figure 15-2, during the Packet Header Template negotiation, in the third message block HP-DLPI provides a set of flags in the b_flag element of the message. The flags set in this element must be used by the transport layer whenever the Packet Header Template created by HP-DLPI is used to send out M_DATA packets. The flag that HP-DLPI sets is used to mark whether the packet header template created by HP-DLPI contains an OOP data template.

This is required when working with the TCP/IP implementation on HP-UX. The TCP/IP layer might not be aware of the OOP data template in the Packet Header Template if it has no information to send to DLS providers as OOP data in the outbound packet. TCP/IP does not set the DL_HP_OOP_PRESENT flag in the Fastpath packet when it uses the Packet Header Template created by DLS providers. A DLS provider must set the DL_HP_OOP_PRESENT flag in the b_flag element of the third message block. This flag is set when the Packet Header Template created has an OOP data template. The value of the DL_OOP_PRESENT flag is defined in Table 15-5.

This flag must also be used by STREAMS modules and transport layer to identify packets with OOP data in both the inbound and outbound paths.

Table 15-5 shows a sample flag definition.

Table 15-5 DL_HP_OOP_PRESENT Flag Definition

<table>
<thead>
<tr>
<th>b_flag (defined in &lt;sys/dlpi_ext.h&gt;)</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL_HP_OOP_PRESENT</td>
<td>OOP data is present in the packet.</td>
</tr>
</tbody>
</table>

Outbound Processing

In the outbound processing path, the transport layer can send down any packet with the Packet Header Template prepended. The steps in the following section must be followed to properly traverse the packet header template. The valid OOP data for transmission between DLS providers and the transport layer is defined in Table 15-2. The transport layer can walk through the OOP data template and provide the information that needs to be sent down to the DLS providers.
Algorithm

After receiving the Packet Header Template, the DLS user must follow these steps:

1. Fill in the CKO information in the cko_info_t structure if it has CKO information to pass.
2. Set the cko_type field to something other than CKO_ALGO_NONE.
3. Search for the type in the OOP data template. When it finds a known type that requires transport to provide data [Type & DL_HP_TYPE_MASK], move to the data area of the OOP Data Tuple, place the data, and reset the DL_HP_TYPE_DATA_INVALID flag.
4. Set the b_flag element of the M_DATA message block that contains the template to the b_flag provided by DLS provider during the packet header negotiation.

**NOTE:** The complete OOP and LLC header must always be in one mblk.

Inbound Processing

In the inbound path, the DLS provider must place the OOP data in the Packet Header Template before passing the packet to the transport layer. The DLS provider must also set the b_flag of the M_DATA message block to inform the transport layer that OOP data is present in the packet.

Algorithm

1. Check if b_flag of the mblk_t has the DL_HP_OOP_PRESENT flag set. If set, proceed to the next step. Otherwise, no OOP data is present. The message block only contains the packet payload.
2. If DL_HP_OOP_PRESENT is set, find the cko_info_t structure pointed to by b_rptr and check if cko_type is a value other than CKO_ALGO_NONE. If it is a valid CKO type, get the information from the cko_info_t structure. The STREAMS module can also get the cko_offset, which is the offset to the packet payload.
3. Move the b_rptr past the cko_info_t structure and read the Type field of OOP Data Tuple. If the type is unknown, skip past the unknown type size received from the Size value of the OOP Data Tuple. If a known OOP data type is encountered, read it.
4. If DL_HP_OOP_EOO is encountered, this is the end of OOP data. Use the cko_type read in step 2 to go to the packet payload.

STREAMS Modules

Between the transport layer and a DLS provider, you can push STREAMS modules to serve various purposes, such as filtering, load balancing, and firewall. Because any packet can traverse through the stream on which a module has been pushed, the module must understand the format of the packets to function properly.

Most of the modules that are pushed between DLS provider and the transport layer do not make use of the OOP data, and process the packet payload only. For example, a filter module might only process the IP data that is being passed.

The format of the OOP data provides a straightforward way to find the payload without traversing the OOP data. If a module does not need OOP data, it must complete the following steps:

1. For each packet, check whether the DL_OOP_PRESENT flag is set in the b_flag field.
2. If set, cast the data pointed to by b_rptr of the mblk to uint32_t to get the offset to the payload. The first 32 bits contain the first element of the cko_info_t structure, which is present if the DL_HP_OOP_PRESENT flag is set.
3. Add the offset value read from step 2 to b_rptr to go to the payload. For example, it points to the IP header in both the inbound and outbound path for the IP transport layer.
4. If the DL_OOP_PRESENT flag is not set, there is no OOP data in the packet. The beginning of the mblk (b_rptr) points to the transport layer header (for example, IP header).
NOTE: STREAMS modules must not make any assumption about the alignment of the IP header.

Packet Trains

HP-UX 11i v3 supports fragment trains in the outbound path. A fragment train is constructed with the individual fragments of an IPv4 or IPv6 datagram. The fragment trains are linked by `b_cont` element of the `mblk_t` structure. The start of a new fragment is specified by the presence of the `DL_HP_OOP_PRESENT` flag in the `b_flag` element. Figure 15-5 shows a fragment chain.

**Figure 15-5 Fragment Chain**

STREAMS modules can read the `DL_IOC_DRIVER_OPTIONS` ioctl to determine whether packet trains will be transmitted on a particular stream. If the train feature is negotiated, they must walk through the data packets that are being transmitted to get to each of the packet fragments.
16 Supporting the HP SMH NIC Tool in LAN Drivers

HP System Management Homepage (HP SMH)\(^1\) is a single-system management solution for HP-UX systems. It comprises a number of tools for various system administration tasks, including the Network Interfaces Configuration (NIC) tool.

This chapter describes how to create the driver shared library to add the support for SMH NIC tool, so that the IHV LAN driver instances can be managed using SMH NIC tool.

This chapter addresses the following topics:

- “NIC Tool Overview”
- “How the NIC Tool Driver Shared Library Works” (page 388)
- “Creating the NIC Tool Driver Shared Library” (page 393)

\(^1\) System Administration Manager (SAM) is deprecated in the 11i v3 release of HP-UX. HP System Management Homepage (HP SMH) is the recommended tool for managing HP-UX. HP SMH provides Graphical User Interface (GUI), Text User Interface (TUI), and Command-Line Interface (CLI) for managing HP-UX. It can be accessed by using the `smh` command (`/usr/sbin/smh`).

NOTE: The terms SMH NIC tool, NIC tool, and NCweb are used interchangeably in this chapter. All these terms refer to the HP SMH Network Interfaces Configuration tool.

NIC Tool Overview

The NIC tool enables a system administrator to configure and manage the network interfaces on a system.

To start the NIC tool, complete the following steps:

1. Log on to the HP System Management Homepage
   
   For information on how to logon, see the *HP System Management Homepage User Guide* at: http://www.docs.hp.com

2. Click **Tools**. The Tools page is displayed.

3. Click **Network Interfaces Configuration**. The Network Interfaces Configuration page is displayed.

4. Click **Network Interface Cards**. The HP-UX Network Interfaces Configuration (NIC) tool page is displayed.

Figure 16-1 shows the NIC tool main page. The NIC tool comprises different tabs for managing the network interfaces. Depending on your system configuration, you may see a different set of tabs and lists of network interfaces. In Figure 16-1, the NIC and VLAN tabs are visible, and the NIC tab is the active tab. The NIC tab is the only tab that can support IHV LAN drivers, provided the driver delivers the shared library per the guidelines provided in this chapter.

The following sections discuss how the NIC tool driver shared library works. It also describes how to create this shared library to provide the SMH NIC tool support in network drivers, specifically the NIC tab.
How the NIC Tool Driver Shared Library Works

The NIC tool shared library is the layer between the SMH NIC tool and a network interface driver.

**NOTE:** The SMH NIC tool shared library and the NIC tool shared library refer to the same shared library.

Figure 16-2 shows the SMH NIC tool architecture.

**Figure 16-2 SMH NIC Tool Architecture**

When the user interacts with the HP SMH NIC tool, a series of events occur. Table 16-1 (page 389) describes the user actions and the events that occur. Figure 16-3 (page 390) shows a high level flow diagram of how the driver shared library works when the user interacts with the NIC tool.
<table>
<thead>
<tr>
<th>User Action</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selects <strong>NIC</strong> (Network Interface Cards) tab</td>
<td>List of network interfaces on the system is displayed in the List area of the page.</td>
</tr>
<tr>
<td></td>
<td>Tasks relevant for the subsystem are displayed in the Tasks area of the page.</td>
</tr>
<tr>
<td>Selects an interface in the displayed list of network interfaces</td>
<td>Details of the selected interface are displayed at the bottom of the page, that is, at the end of the list.</td>
</tr>
<tr>
<td></td>
<td>Tasks relevant for the selected interface are displayed in the Tasks area of the page.</td>
</tr>
<tr>
<td>Selects a task (View/Modify NIC Attributes)</td>
<td>The View/Modify NIC Attributes page showing the attributes that can be modified is displayed.</td>
</tr>
<tr>
<td>Modifies attributes that are marked Refresh on Change</td>
<td>The View/Modify NIC Attributes page with the changed data is displayed.</td>
</tr>
<tr>
<td>Clicks the <strong>Preview</strong> button</td>
<td>The View/Modify NIC Attributes page showing the commands for the task, at the bottom of the page is displayed.</td>
</tr>
<tr>
<td>Clicks the <strong>Cancel</strong> button</td>
<td>The task is cancelled and the main page showing the list of interfaces and the details is displayed.</td>
</tr>
<tr>
<td>Clicks the <strong>OK</strong> button</td>
<td>The task is executed and a message maybe displayed based on whether the task was executed with success or failure.</td>
</tr>
</tbody>
</table>
The initial event of displaying the list of network interfaces in the NIC tab does not involve the driver shared library. To display the list of interfaces on the system, the SMH NIC tool calls the nwmgr common services shared library. The nwmgr common services shared library exports a set of services to the SMH NIC tool and to other shared libraries.

The nwmgr common services shared library obtains the list of all interfaces including the native STREAMS DLPI interfaces retrieved from HP-DLPI and provides it to the SMH NIC tool. No calls are made to the driver shared library to get the list of network interface instances. For more information on nwmgr, see Chapter 13 (page 337).

The events that follow after the list of interfaces is displayed involve the driver shared library.
The NIC tool driver shared library does the following:

- Provides data for displaying the details of the interface and the tasks associated with the interface, when the user selects an interface
- Provides data for displaying the task form, when the user selects a task
- Provides data for displaying updated fields, after user changes any field that is marked Refresh on Change
- Provides data for displaying messages, based on whether the task was executed successfully
- Provides data for displaying the command equivalent of the selected task, when the user clicks Preview

The following sections explain how a driver shared library works when the user interacts with the SMH NIC tool.

- Providing data for displaying the details of the interface and the tasks associated with the interface

  When a user selects a network interface listed in the NIC (Network Interface Cards) tab, the SMH NIC tool attempts to load the driver specific shared library. The SMH NIC tool looks for the shared library with a specific name at a known location. The SMH NIC tool can either succeed or fail to locate the driver shared library.

  If the SMH NIC tool succeeds in locating the driver shared library, it loads the driver shared library and invokes the `drivername_get_if_tasks` and `drivername_get_if_details` APIs in the driver shared library. The data returned by these APIs is used to update the NIC tab page, particularly the Details of Interfaces area in the page.

  If the SMH NIC tool fails to find a shared library, it calls the APIs exported by the `nwmgr` common services shared library and displays the retrieved information. In such a case, the SMH NIC tool does not provide the user the View/Modify NIC Attributes menu in the Tasks area, though it enables users to view and configure network level attributes.

- Providing data for displaying the task page, when the user selects a task

  When the user clicks an action item in the Tasks area to perform a task, the NIC tool invokes the `drivername_get_task_info` API for that task. The resulting `task_info` structure is used to generate a new HTML page for that task.

- Providing data for displaying updated fields

  If the user changes an attribute that is marked with the `NA_FLAG_REFRESH_ON_CHANGE` flag while interacting with a task page, the SMH NIC tool front end (see Figure 16-1 (page 388)) ascertains whether the attribute value has changed from the time that page was generated. If the value has changed, the SMH NIC tool invokes the `drivername_refresh_task_info` API. The resulting `task_info` is used to generate a new HTML page.

- Providing data for displaying messages

  When the user interacts with a task page and clicks OK, the SMH NIC tool loads the driver shared library and invokes the `drivername_do_task` API. This API executes the task. The task outcomes are indicated by the return value from the API.

  typedef enum {
    NM_DO_TASK_RET_OK = 0,
    NM_DO_TASK_RET_MSG,
    NM_DO_TASK_RET_RETRY,
    NM_DO_TASK_RET_ERR_FATAL,
    NM_DO_TASK_RET_ERR_RETRY,
    NM_DO_TASK_RET_OK_NOMSG
  } netmod_do_task_ret_t;

  The return values indicate whether the task was executed with success or failure. The return value is either 0, 1, 2, or 5 if the task is successful. The return value is 3 or 4 if there is failure.
in executing the task. The return value is determined by the type of success or failure. Table 16-2 (page 392) explains the return values.

**Table 16-2 Return Values and Task Outcomes**

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null (0)</td>
<td>The task is completed successfully. If the nti_msgs field in the task_info is NULL, the user is directed to the main page of the tab.</td>
</tr>
<tr>
<td>Not null (1, 2, 3, or 4)</td>
<td>If the nti_msgs field is not NULL, a new page is generated with the ns_display_name field from each of the messages and an OK button. The user is expected to browse the messages and then click OK to return to the main page of the tab.</td>
</tr>
<tr>
<td>1</td>
<td>The task is completed successfully. The SMH NIC tool displays a warning or informative message.</td>
</tr>
<tr>
<td>2</td>
<td>The task is completed successfully. The SMH NIC tool displays a verification message. The user is allowed to perform the task again. The return value is 2 in the following case: The user has chosen to only verify the task for potential failures and errors by selecting the Verify-Only option in the task form. After the verification message is displayed, the user can choose to execute the task by not selecting the Verify-Only option.</td>
</tr>
<tr>
<td>3</td>
<td>The task failed to execute. The failure is fatal. For example, the selected interface is deleted from the system. In the case of fatal errors, the driver shared library provides the error message and the SMH NIC tool displays the error message in a separate page. This message has an OK button, similar to the message displayed for Success with Message outcome (Return code 1).</td>
</tr>
<tr>
<td>4</td>
<td>The task failed to execute. The failure is not fatal. The user can attempt to perform the task again. An example of a reason for such a task failure is the occurrence of a transient memory allocation error. For a retry-able error, the driver shared library provides the error message and the SMH NIC tool displays the error message in the same page at the top. In the case of a retry-able error, the nti_msgs field of out_task is treated as if it contains error messages in the ns_display_name fields. If the out_task is not created by copying the in_task, the driver shared library sets the out_task to NULL and indicates the failure as a retry-able error. When this occurs, the driver shared library cannot communicate an error message to the API caller. The caller is expected to assume that a return code of Failure with Retryable Error (Return value 4) together with a NULL out_task indicates a memory allocation failure. The API caller must allow the user to perform the task again.</td>
</tr>
<tr>
<td>5</td>
<td>The task is completed successfully. If the return value is 5, the SMH NIC tool does not display any message.</td>
</tr>
</tbody>
</table>

- Providing data for displaying the command equivalent of the selected task
  
  If the user clicks Preview while interacting with a task page, the NIC tool loads the shared library and invokes the drivername_get_cmd_preview, unless the task_info has set the NTI_FLAG_DELEGATE_PREVIEW flag. If the flag is set, the NIC tool generates the command preview. The generated command preview is displayed to the user in the task page, allowing the user to execute the task.

  At any point, the user can click Cancel to cancel a task and return to the main page of the selected tab. When the user clicks Cancel the SMH NIC tool invokes the drivername_get_if_tasks and drivername_get_if_details APIs in the driver shared library, thereby cancelling the
action. The data returned by these APIs is used to update the NIC tab page, particularly the Details of Interfaces section.

Creating the NIC Tool Driver Shared Library

To provide complete SMH NIC tool support in LAN drivers, network driver developers must create the NIC tool driver shared library. This section addresses the following topics:

- Requirements for the Driver to Work With the NIC Tool
- Developing the Shared Library (page 393)
- Building the Shared Library (page 395)
- Installing the Shared Library (page 395)

Requirements for the Driver to Work With the NIC Tool

In order for your network drivers to work with the SMH NIC tool, ensure that the following requirements are fulfilled:

- The network interface driver must be of class lan. For more information on writing a driver, see Chapter 4 (page 77).
- The network interface driver must register each instance with HP-DLPI during the NIC initialization, which is normally done in the driver init function. For more information, see Chapter 11 (page 221) for HP-DLPI based drivers and Chapter 12 (page 293) for Native DLPI drivers.
- The network interface driver must supply the driver shared library and install it per the guidelines. For information on installing the shared library, see “Installing the Shared Library” (page 395). The SMH NIC tool interacts with the network interface driver through this driver shared library to retrieve information, which is displayed by the SMH NIC tool.

The advantage of supplying the shared library is that the SMH NIC tool can retrieve driver-specific information for a network interface driver. This enables the user to configure link level attributes. However, it is optional for IHV network driver developers to supply the driver shared library.

If a network interface driver fulfils only the first two requirements, the NIC tool displays information about the interfaces belonging to that driver in the list of interfaces. The users can view and configure any network level attributes.

Developing the Shared Library

The HP SMH tool supports both HP-DLPI and Native STREAMS DLPI IHV network drivers. For more information on writing HP-DLPI network drivers, see Chapter 11 (page 221). For more information on writing Native STREAMS DLPI drivers, see Chapter 12 (page 293).

The HP-UX 11i v3 Driver Development Kit (DDK) contains two fully functional sample SMH NIC tool driver shared libraries for use with the HP-DLPI (IELAN) and Native STREAMS DLPI (ENET) sample network drivers that are also included in the DDK.

To obtain a fully functional sample SMH NIC tool driver shared library, download the HP-UX 11i v3 Driver Development Kit available at: www.hp.com/go/hpux_ddk. After installing the HP-UX 11i v3 DDK, go to the misc/ncweb directory to obtain the NCweb/SMH NIC tool shared libraries for the following sample drivers:

ENET Native STREAMS DLPI network driver
IELAN Non-Native HP-DLPI based network driver
HP recommends that you use one of these shared libraries as the base for your driver’s shared library. Consider the following guidelines while modifying the SMH NIC tool driver shared library:

- Prefix all the functions with the name of your network driver. This avoids symbol name clashes. For example, all the function names for the ENET driver’s SMH NIC tool shared library have the enet_ prefix.
- Include the netmod.h header file that is provided with the DDK. This header file defines the nwmgr and SMH APIs. Do not modify the header file.

The APIs – data structures and function prototypes – are exported through the netmod.h header file. The HP-UX 11i v3 Driver Development Reference contains the manpages for all the APIs that are necessary for developing a SMH NIC driver shared library. For more information on the APIs, see the HP-UX 11i v3 Driver Development Reference available at: www.hp.com/go/hpux_ddk

Table 16-3 lists the APIs used by the SMH NIC tool.

<table>
<thead>
<tr>
<th>API Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>drivername_get_if_tasks</td>
<td>Returns the list of tasks that the user can select only after selecting an interface. For example, the user can select the task View/Edit only after selecting an interface. This API returns the same data for a given context no matter how often it is called.</td>
</tr>
<tr>
<td>drivername_get_if_details</td>
<td>Returns the list of attribute groups that must be displayed in the Details of Interface for a selected interface in the main page of the tab. This API returns the same data for a given context no matter how often it is called.</td>
</tr>
<tr>
<td>drivername_get_task_info</td>
<td>Returns the task information for a selected task. This API returns the same data for a given context no matter how often it is called.</td>
</tr>
<tr>
<td>drivername_do_task</td>
<td>Executes the specified task, after considering qualifiers such as Verify-Only. This API is NOT idempotent.</td>
</tr>
<tr>
<td>drivername_get_cmd_preview</td>
<td>Returns the Command Preview string for a given task. This is required only if the library supports a task for which the NTI_FLAG_DELEGATE_PREVIEW flag is not set.</td>
</tr>
<tr>
<td>drivername_refresh_task_info</td>
<td>Refreshes the task information. This is required only if the subsystem specifies the Refresh bit for any group in any task information it provides to SMH.</td>
</tr>
</tbody>
</table>

Table 16-4 lists the shared data structures used by the SMH NIC tool.

<table>
<thead>
<tr>
<th>Shared Data Structure Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>netmod_task_info_t</td>
<td>Encapsulates a task that the API caller application can display.</td>
</tr>
<tr>
<td>netmod_task_info_flags_t</td>
<td>A bit mask whose fields can be OR’d together. The bits are set by the subsystem. The API caller can only read these flags.</td>
</tr>
<tr>
<td>netmod_group_t</td>
<td>Represents a collection of related data, such as attributes, MIB statistics fields, the list of interfaces needed for creating and modifying virtual interfaces, or qualifier attributes.</td>
</tr>
<tr>
<td>netmod_group_type_t</td>
<td>Represents the type of contents of a group.</td>
</tr>
<tr>
<td>Shared Data Structure Name</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>netmod_group_flags_t</td>
<td>A bit mask whose fields can be OR’d together. The bits are set by the subsystem. The API caller can only read these flags.</td>
</tr>
<tr>
<td>netmod_table_t</td>
<td>A list of interfaces in tabular format. If <code>ng_group_type</code> in <code>netmod_group_t</code> is set to <code>NG_TYPE_IF_LIST</code>, <code>ng_data</code> points to the data structure of this type.</td>
</tr>
<tr>
<td>netmod_row_t</td>
<td>Represents data for each interface listed in the interfaces table. The data includes fields needed for virtual groups, such as membership flags. However, these fields are not relevant for the <code>get_first_page_data</code> API.</td>
</tr>
</tbody>
</table>
| netmod_range_table_t             | This structure addresses the following scenario. If there is a quantity R whose values are within a range (minimum and maximum), the values might be integers or floats. The set of values of R is divided into N ranges, each with a start and end, such that the N ranges together include every value of R without duplicates. That is, the ranges do not overlap and also do not have any gaps between them. There is another quantity Q that is dependent on the value of R. For each range of R, quantity Q needs to be set to a value. There might be more than one such dependent quantity. There might be zero or more attributes that are related to R and its dependents in some way. The number of ranges, N in this scenario, and the number of dependent quantities, D in this scenario, are determined by the subsystem library. The user specifies the following:  
  • Ending values for each range of R  
  • The value of each dependent quantity for each range |
| netmod_freefunc_t                | A type definition for a free-function pointer. Subsystems might define a free-function for each API and pass them to the SMH NIC tool. The NIC tool must invoke this function if the tool is using the data obtained through that API. |
| netmod_tab_code_t                | Represents the integer codes for the various tab areas in the SMH NIC tool, such as NIC, RDMA, APA and VLAN. When the SMH NIC tool invokes an API that has a tab code parameter, the NIC tool must pass the code for the tab from which the API is called. When an application other than SMH NIC tool uses an API that has a tab code parameter, the tab code must be specified as NONE. |

**Building the Shared Library**

The sample SMH NIC tool driver shared libraries contain a `Makefile` file that detects the underlying platform and selects the correct options to compile and build the shared libraries. To compile and build the shared library, execute the following command in the source file folder:

```
$ make
```

On PA-RISC systems, a 32-bit shared library is built. On Integrity systems, a 64-bit shared library is built.

**Installing the Shared Library**

To install the built shared library, execute the following command:

```bash
Creating the NIC Tool Driver Shared Library  395
```
$ make install

The Makefile copies the shared library to the appropriate directory, depending upon the platform, and creates all the needed symbolic links.

To uninstall the shared libraries, execute the following command:

$ make uninstall
This chapter provides an overview of the mass storage I/O stack in HP-UX. This chapter addresses the following topics:

- “Mass Storage Stack Features”
- “Overview of the HP-UX Mass Storage I/O Stack” (page 397)
- “Major Data Structures” (page 400)
- “I/O Flow” (page 402)
- “SCSI Addressing Paradigm” (page 402)
- “Probe Functions” (page 403)
- “Interface Driver Migration to HP-UX 11i v3” (page 404)

**Mass Storage Stack Features**

The mass storage stack in HP-UX 11i v3 supports enhanced virtualization and adaptability of storage as follows:

- Persistent device special files are not tied to the physical hardware path to a disk or tape drive, but instead map to the device's unique worldwide identifier (WWID). Thus, the device file is unchanged if a LUN is moved from one HBA to another, moved from one switch or hub port to another, or presented with a different target port to the host.

- Multipathing is managed transparently by the Mass Storage Stack. If a LUN has multiple hardware paths (known as lunpaths), I/O requests are distributed across all available lunpaths with a choice of load balancing algorithms. If a lunpath fails, the Mass Storage Stack automatically disables the failed lunpath and I/O continues on all available enabled lunpaths. Any failed or nonresponsive lunpaths are monitored, so that when a failed lunpath recovers, it is automatically and transparently reincorporated into any load balancing. New LUNs and lunpaths are also automatically discovered and added to load balancing.

  The persistent device special file serves as a single access point for all the lunpaths. If there are any changes to the LUN’s pathing such as addition, removal, or modification of a lunpath, applications using the persistent device special file are not affected, if at least one lunpath is still active.

- The I/O system automatically recognizes and adapts to asynchronous device changes to the LUN’s pathing, size, and block size.

- The HP-UX SCSI stack supports conventional SCSI-3 addressing; it is not limited to a SCSI-2 addressing paradigm.

- Scalability limits have been lifted. For example, the number of I/O buses is not limited by minor number formats.

**Overview of the HP-UX Mass Storage I/O Stack**

Figure 17-1 illustrates the Mass Storage I/O Stack in HP-UX, its key components, and the key modules it interacts with.
From the top of the diagram, the Mass Storage I/O Stack includes the following:

**User Applications or I/O Commands**
These lie outside the kernel and access the SCSI subsystem through kernel interfaces such as upper layer modules, raw device access, diagnostics, or pseudodrivers.

**Upper Layer Modules (ULMs)**
These are kernel modules above the SCSI stack. They consist mainly of volume managers and file systems.

**SCSI Class Drivers (SCSI device drivers)**
At the top of the SCSI subsystem are the class drivers, which control the individual SCSI LUNs for a given class of device, such as disk or tape devices. A generic class driver, called the SCSI pass-through driver (esctl), can be used by SCSI-specific applications, commands, or upper level modules to build SCSI commands directly. It then sends them to a device for vendor-specific or special control of a device or for application control of a device class that is not supported by a specific class driver. Existing class drivers for HP-UX include the SCSI disk driver (esdisk) and SCSI tape driver (estape). The SCSI pass-through driver is described in esctl(7) in the HP-UX Reference.

**SCSI Services**
Below the class drivers is a layer called the SCSI Services, which provides connection interface services between the SCSI class drivers and interface drivers. The SCSI Services layer defines the entry points...
SCSI Interface Drivers
(SCSI transport drivers)

At the bottom of the SCSI subsystem are the interface drivers, which control the hardware that transports the SCSI protocol from the host system to the end SCSI device (LUN). They typically directly control a host bus adapter (HBA), and directly access their hardware through Programmed I/O (PIO), by reading and writing registers on the HBA, or by the initiation of Direct Memory Access (DMA) to cause the HBA to transfer data to and from host memory. In HP-UX 11i, parallel SCSI (pSCSI), Serial Attached SCSI (SAS), and Fibre Channel (FC) interface drivers control pSCSI, SAS, and FC HBAs. For information on writing a SCSI interface driver, see Chapter 18 (page 405).

Kernel Services

To the side of the SCSI subsystem are WSIO and kernel services, and the additional SCSI services described above, which drivers can use. WSIO is the Device Driver Environment (DDE) CDIO that defines and supports the entry points and configuration services used by the SCSI class and interface 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. The additional SCSI Services provide SCSI subsystem-specific DMA services that interface drivers are encouraged to use. Drivers also need a number of kernel services such as timer and memory allocation services. A summary of kernel services is listed as follows. For detailed discussions on WSIO and how a driver fits into the WSIO installation and configuration sequence, see Chapter 1 (page 29), Chapter 2 (page 33), Chapter 3 (page 69), Chapter 4 (page 77), and Chapter 8 (page 183). For detailed discussions on kernel services, WSIO services, and SCSI services, see Chapter 1 (page 29), Chapter 2 (page 33), Chapter 4 (page 77), and Chapter 8 (page 183).

ULM Services

These functions asynchronously notify upper level managers when their underlying storage changes, such as when a LUN changes its size, its block size, or its available paths.

Multipathing Services

These services handle the configuration, detection and management of multiple paths to a SCSI LUN. They include recognition and grouping of multipathed devices, and load balancing of I/O requests through available LUN paths. In addition, they provide a plug-in interface to support active-passive devices.

I/O Services

These services enable drivers to issue SCSI command blocks (CDBs). For class drivers, they support load balancing and path selection, and SCSI control block allocation and management. For interface drivers, they support tag management and DMA mapping, including special mappings for cards that only allow 32-bit DMA.
In addition, they process common SCSI ioctl requests such as SIOC_IO and SIOC_IO_EXT.

**Miscellaneous Services**

For class drivers, these provide private services for locking, logging, and thread management. For interface drivers, they provide public services to allocate and use a critical region for synchronization, schedule and reliably untimeout timers, generate traces on controller and LUN objects, queue requests on a particular thread, and manage linked lists.

**Node Services**

WSIO provides functions for interface drivers to use to create and manage I/O tree nodes, properties, and callbacks invoked when certain events occur on the node. These services include creation and configuration of mass storage I/O nodes, registration of I/O node properties, and management of interactions with the persistent data store.

**Target Manager Services**

These services manage SCSI target paths. They enable interface drivers to register target paths (as a result of a probe operation) and unregister them (as a result of online deletion or DLKM unload). If an interface driver asynchronously detects a target path change, such as a target going offline or online, or a port ID change, it can notify the target manager, which fans the event onto the relevant lunpaths. In addition, this manager is responsible for discovering and registering the LUNs and lunpaths underneath each registered target path. Interface drivers can also register legacy hardware paths for targets and LUNs, and query previously registered target and lunpaths. Finally, the target manager synchronizes bus device resets to avoid redundant resets.

**Controller Services**

These services are similar to the target manager services, but apply to SCSI controllers. They include controller registration and unregistration, bus reset synchronization, and services to enable interface drivers to access the list of SCSI controllers registered for that driver.

**Major Data Structures**

The SCSI subsystem uses certain data structures to define interfaces between and pass data across the different layers of the SCSI stack. Within these data structures, some fields are managed by a particular layer, and other layers must treat them as read-only. Figure 17-2 illustrates those data structures and their connectivity.
There are more data structures within each type of driver. Those presented here are global and are necessary to describe the flow of I/O requests through the SCSI stack. For more information about each data structure, see its manpage in the *HP-UX 11i v3 Driver Development Reference.*

**I/O buffer header**  
*(struct buf, or bp)*  
This structure, the header for buffers in the buffer pool, is used to describe a block I/O request. I/O requests are passed to the interface driver as a `buf` structure. It includes a pointer to the data to be transferred, the data direction, and data counts.

It also contains queue pointers. Queues manage I/O flow control and ordering. Buffers can be queued at all levels of the SCSI subsystem: in the class driver (to avoid exceeding a LUN’s queue depth), in SCSI services (to await resources or perform retries), or in the interface driver (to maintain ordering). For more information, see `buf(9S).`
**SCSI control block**
(escsi_scb_t, or scb)

This structure holds SCSI-specific information associated with an I/O request, such as the SCSI command bytes, the completion status, sense data, residues, and the time limit for the I/O request. It is typically created by the class driver and updated by the interface driver. The buffer header contains a pointer to the scb. For more information, see scb(9S).

**SCSI interface switch**
(escsi_ifsw_t)

This structure defines the interface driver entry points and attribute values. SCSI Services uses it to interact with interface drivers. In either its driver_attach or driver_init routine, the interface driver allocates and initializes this structure, then passes it to SCSI Services as part of registering its controller. For more information, see escsi_ifsw(9S).

---

**I/O Flow**

This section shows an example of how I/O requests flow through the mass storage stack. It shows how the data structures and interfaces work between the class driver, SCSI Services, and interface driver. Because the focus is on driver interaction, the functionality within SCSI Services is simplified. Figure 17-3 shows the steps for a block I/O request.

**Figure 17-3 I/O Flow Diagram**

1. An upper layer module, such as LVM, sends a request to the class driver through its driver_strategy entry point in the bdevsw table, passing in a buf structure.
2. The driver_strategy entry point calls SCSI services, which can queue the bp pending necessary resources to enable the request to be sent to the interface driver.
3. SCSI Services queues the buffer on the interface driver's input queue, and calls the driver_if_start entry point in the interface driver's switch structure. This informs the interface driver that a bp has been queued on the driver’s select queue for processing.
4. The interface driver’s driver_if_start entry point unqueues the I/O request and retrieves the SCSI control block from it. It then initiates the I/O on the HBA.
5. Once the I/O request completes, the interface driver is notified. It logs SCSI completion and sense status in the SCSI control block, updates completion information in the buffer header, then invokes the callback function in the scb (scb->cbfn) to signal that the I/O request is complete.

**SCSI Addressing Paradigm**

In HP-UX 11i v3, the HP-UX mass storage stack supports two different addressing paradigms: SCSI-2 (legacy) and SCSI-3. SCSI-2 addressing only supports up to eight LUNs per target and
up to 16 targets per controller. SCSI-3 addressing supports a much larger addressing model in which the number of target paths per controller or lunpaths per target is not limited by the SCSI stack, only by device, controller, or transport protocol addressing restrictions.

In releases prior to HP-UX 11i v3, mass storage devices were represented with SCSI-2 addressing. The names of device special files, the encoding of their minor numbers, and the path representation in the system’s I/O tree were all based on three fixed fields of controller (8 bits), target (4 bits), and LUN (3 bits). This encoding limited the system to 256 distinct buses and 32768 distinct lunpaths. It also created a different device special file for each lunpath, so a multipathed LUN had multiple device special files, one for each lunpath. Interface drivers that supported SCSI-3 devices worked around the SCSI-2 addressing restriction by creating virtual controllers, virtual targets, and virtual LUNs in the I/O tree.

Beginning with HP-UX 11i v3, the mass storage stack supports SCSI-3. That is, it supports the use of SCSI transport-specific target port names and target port identifiers. As a result, interface drivers no longer need to perform SCSI-3-to-SCSI-2 mapping schemes to report their discovered devices and have them incorporated into the I/O tree. However, to provide backward compatibility support for legacy device files which use SCSI-2 addressing, HP-UX native interface drivers must provide a legacy hardware path build function (an entry point in escsi_ifsw_t). After probing a target path, SCSI Services calls this routine. It creates the list of legacy target and LUN hardware paths associated with the probed target path and the discovered lunpaths.

**NOTE:** It is optional for third party interface drivers to support legacy hardware path building.

**Probe Functions**

Each part of the mass storage stack must probe for SCSI targets and LUNs.

**SCSI Class Driver**

SCSI class drivers configure newly discovered LUNs and lunpaths.

**SCSI Services**

SCSI Services registers targets and LUNs with the I/O infrastructure. SCSI Services also probes for LUNs and lunpaths beneath discovered targets, collects SCSI inquiry information, and notifies the class drivers when new LUNs and lunpaths appear.

**SCSI Interface Driver**

The interface driver discovers targets under its associated SCSI controller and registers them with SCSI Services. It provides a controller probe routine (driver_if_ctlr_probe) in its interface switch structure, which it registers during the driver's driver_install or driver_attach entry point. It also defines a node callback routine in the d_drv_cb field of its dvr_ops_t structure.

The controller probe routine facilitates the discovery of targets and LUNs and their instantiation in the I/O tree. SCSI Services invokes it to probe the controller and discovers targets under it, typically using a transport-specific mechanism. When the interface driver discovers a new target path, either due to a probe or an asynchronous mechanism, the driver registers the discovered target paths with SCSI Services. In turn, SCSI Services registers those target paths with the I/O infrastructure and probes for LUNs beneath each target. The list of LUNs discovered is passed to the interface driver's legacy hardware path build routine, if one was registered in the escsi_ifsw_t, to create legacy hardware paths in the I/O tree.

A new variant of probing introduced in HP-UX 11i v3, SCAN_ALL, scans for all objects under a given node. When the I/O infrastructure performs a SCAN_ALL operation on a controller, it invokes the interface driver's callback routine. The callback routine forwards the probe to SCSI Services, which calls the driver's controller probe routine.
Interface Driver Migration to HP-UX 11i v3

Support of the enhanced mass storage stack introduces several changes in the SCSI layer data structures and external interfaces provided to mass storage drivers. To operate with the new mass storage stack, HP-UX drivers written for a previous release require some modification. The HP-UX 11i v2 to HP-UX 11i v3 I/O Driver Migration Guide describes the migration process in detail, but the following list gives a brief overview of the changes:

- Class drivers and interface drivers must support the enhanced switch structures, which supersede the existing scsi_ifsw structure.
- The mechanism for operations such as opening, closing, and resetting a LUN has changed. Instead of synchronous calls to the interface driver, the I/O system now performs asynchronous calls through the interface switch. The interface driver asynchronously invokes a callback function on completion. The completion status of an asynchronous call occurs asynchronously from the original call, thus a callback function is needed. The driver_if_start entry point in HP-UX 11i v2 and earlier releases was an example of an asynchronous call.
- Interface drivers now register their HBA controller and target devices with the SCSI stack during driver initialization and probe operations. Each interface driver instance registers the controller it is operating on and the target devices visible to that controller.
- Interface drivers must provide a controller probe routine, which is called by SCSI services to probe the controller and discover targets under it. There is no longer a default parallel SCSI-specific probe routine.
- Interface drivers must support asynchronous event notification to report controller and target path related events to the SCSI stack, which then notifies other layers of the I/O system. Such events include the discovery of new targets and targets going online or offline.
- Drivers must support updated I/O completion status values, prioritized I/O requests, and additional locking policies.
Writing a SCSI Interface Driver

This chapter provides information necessary to design and develop SCSI interface drivers consistent with the driver development steps outlined in Chapter 4 (page 77). First, read chapters 1–8 of this manual. SCSI interface drivers manage interface cards (HBAs) that attach to a mass storage interconnect, including Parallel SCSI (pSCSI) and Fibre Channel (FC). Currently only these two interconnect types are supported for third party driver development in HP-UX, but the information in this chapter is applicable to other SCSI transports as well, such as SAS or iSCSI. SCSI interface drivers are WSIO interface drivers as defined in “Driver Types” (page 33).

This chapter addresses the following topics:

- “External Interfaces to a SCSI Interface Driver”
- “SCSI Interface Driver Development ” (page 413)

IMPORTANT: This chapter supports development of Parallel SCSI and Fibre Channel interface drivers only.

External Interfaces to a SCSI Interface Driver

The external interfaces available to a SCSI interface driver fall into four categories:

**Entry Points**

Driver-exported functions that the SCSI services layer and other modules (WSIO or KRS) can use to interact with SCSI interface drivers and macros.

**Service Functions**

Functions and macros exported by other modules, such as SCSI Services or WSIO Services used by SCSI interface drivers.

**External Data Structures**

Modules used by entry points and service functions to complete specific tasks.

**Header Files**

Files containing service function definitions, external data structures, and supporting definitions and constants.

The external interfaces to SCSI interface drivers can be mandatory, conditional, or optional:

- Mandatory external interfaces are required by all SCSI interface drivers.
- Conditional external interfaces are mandatory when specific conditions exist.
- Optional external interfaces are not mandatory or conditional.

This section provides a short description of the SCSI Services interfaces available to an HP-UX 11i v3 SCSI interface driver. Detailed information on these external interfaces is provided in the HP-UX 11i v3 Driver Development Reference.

**Entry Points**

There are mandatory, conditional and optional entry points to a SCSI interface driver.

**Mandatory Entry Points**

All SCSI interface drivers must provide the following entry points:

- `install` Driver installation
- `attach` HBA controller attachment (claiming)
- `init` HBA controller initialization
- `I/O node callbacks` Controller and target node callback
- `if_lpt_open` LUN path open interface
- `if_lpt_close` LUN path close interface
Conditional Entry Points

The following entry points are required if specific conditions exist:

- **if_ioctl**
  Interface driver-specific support for ioctls issued on LUN device files. This entry point is required to support legacy ioctls that were supported on the LUN-level legacy device files and to support services management functions such as `SIOC_DEL_IFLPT`.

- **if_leg_hwp_build**
  Legacy hardware path build interface. This entry point must be provided by SCSI interface drivers that support legacy hardware paths and device special files.

- **load, unload**
  Dynamic driver install and uninstall. These two entry points must be provided for dynamic loading and unloading of the driver through the HP-UX DLKM interface.

- **drv_event_handler**
  HA event handler. This entry point must be provided for support of any of the HA events such as OLR suspend and resume.

Service Functions and Macros

SCSI Services provide the following service functions and macros available for use by SCSI interface drivers:

- Registration and Probe Services
- Asynchronous Event Services
- Interface Handle Lookup Services
- Tag Management Services
- DMA Mapping Services
- Tracing Services
- SCSI Command Services
- Linked List Services

Each of these services are discussed briefly below. Additional details on each of the individual interfaces are provided in the *HP-UX 11i v3 Driver Development Reference*.

Registration and Probe Services

The following registration and probe services are available to SCSI interface drivers:

- **escsi_ctlr_reg**
  Registers a controller with SCSI Services.

- **escsi_tgt_reg**
  Registers a target with SCSI Services.

- **escsi_ctlr_unreg**
  Unregisters a controller with SCSI Services.

- **escsi_tgt_unreg**
  Unregister a target with SCSI Services.

- **escsi_if_ctlr_prb_cbfn**
  Controller probe completion callback function.

- **escsi_ctlr_node_cb**
  Controller I/O node callback function.

- **escsi_tgt_node_cb**
  SCSI Services target node callback function.

- **escsi_reg_leg_bus**
  Registers a legacy bus with SCSI services.

- **escsi_reg_leg_tgt**
  Registers a legacy target path with WSIO and SCSI Services.

- **escsi_get_inq_serial**
  Gets LUN inquiry data and serial number. Used by the `if_leg_hwp_build` entry point.
The following asynchronous event service is available to SCSI interface drivers:

- **escsi_if_aen**: Sends an asynchronous event notification to SCSI Services.

### Interface Handle Lookup Services

The following interface handle lookup services are available to SCSI interface drivers:

- **escsi_ifctlr_get**: Gets an interface-specific controller structure handle.
- **escsi_iftgt_get**: Gets an interface-specific target path structure handle.
- **escsi_iflpt_get**: Gets an interface-specific LUN path structure handle.
- **escsi_iobj_get**: Gets a pointer to an I/O object.
- **escsi_get_addr**: Gets an instance address for a specified LUN path.

### Tag Management and Allocation Services

The following tag management and allocation services, and macros are available to SCSI interface drivers:

- **escsi_tag_alloc**: Allocates a tag from SCSI services.
- **escsi_tag_free**: Frees a tag back to SCSI services.
- **ifsw->if_flags**: Specifies the scope of the tags. The scope specifies an object that the generated tags are guaranteed to be unique to. The supported scopes are I/O object, target path, or LUN path:
  - **IF_IOBJ_TAGS**: Specifies I/O object scope tags in the `if_flags` field of `escsi_ifsw_t`.
  - **IF_TP_TAGS**: Specifies target path scope tags in the `if_flags` field of `escsi_ifsw_t`.
  - **IF_LPT_TAGS**: Specifies LUN path scope tags in the `if_flags` field of `escsi_ifsw_t`.
- **if_max_tags**: Specifies the maximum number of concurrent tags that can be generated. SCSI Services limits the maximum number of concurrent tags generated to 1024.
- **if_tag_size**: Specifies the number of bits that SCSI Services can use for a tag. The maximum tag size is a 64-bit signed quantity. The definition of an invalid tag is -1, which represents a lack of free tags. SCSI Services generates tags with the specified width.

The HP-UX 11i v3 SCSI stack provides tag management services that are called by SCSI interface drivers to allocate tags for I/Os. A SCSI interface driver can allocate a tag while starting an I/O and free that tag when the I/O is completed. Tag resources are managed by SCSI services, but allocating and freeing tags is the responsibility of each SCSI interface driver. Tag allocation is not done by default in the SCSI services I/O data path, because some SCSI interface drivers do not require or use tags.

SCSI interface drivers specify the nature and size of tags through the `escsi_ifsw_t` structure registered with SCSI services. The fields in the `escsi_ifsw_t` structure specify the type of tags required by interface drivers. The attributes and management information for interface driver tags is encapsulated in a tagpool object. Depending on the scope of the tags, the controller manager, target path manager, or LUN path manager allocates a tag object. The LUN path manager initializes the tagpool pointer in the LUN path to the appropriate tagpool object.

If a SCSI interface driver requires tag allocation, it must set `ifsw->if_flags` to **IF_IOBJ_TAGS**, **IF_TP_FLAGS**, or **IF_LPT_FLAGS**. In addition, the SCSI interface driver must specify the number of concurrent tags required (`if_max_tags`) and the desired width of the tags (`if_tag_size`).
If a SCSI interface driver does not require tag allocation, it must not set any of the scope flags in `if_flags` and it must set `if_max_tags` to 0.

SCSI Services uses a tag stack and a rolling counter to generate tags of the desired width. The lower 10 bits of the tag are generated by the tag stack. The rest of the bits are generated through a rolling counter mechanism. A maximum of 1024 tags are supported by tag allocation services. If there are no free tags available, an invalid tag value of -1 is returned by tag management services.

SCSI Services provides the `escsi_tag_alloc` and `escsi_tag_free` service functions to allocate and release (free) tags.

A SCSI interface driver that has specified a tag scope in `escsi_ifsw_t` can allocate tags by calling `escsi_tag_alloc` while starting an I/O, and free tags by calling `escsi_tag_free` during I/O completion. SCSI Services does not synchronize concurrent calls to tag allocation and free routines. The SCSI interface driver is responsible for serializing concurrent calls to `escsi_tag_alloc` and `escsi_tag_free` for the scope that the those tags are unique to. This is done by holding an interface driver-specific lock while making these calls.

**DMA Mapping Services**

The following DMA mapping service functions are available to SCSI interface drivers:

- `escsi_dma_map`: Performs DMA mapping for I/O buffer.
- `escsi_dma_unmap`: Performs DMA unmapping for I/O buffer.
- `escsi_dma32_map`: Performs 32-bit DMA mapping for I/O buffer.
- `escsi_dma32_unmap`: Performs 32-bit DMA unmapping for I/O buffer.

HP-UX 11i v3 SCSI Services provides common routines called by SCSI interface drivers to map and unmap data buffers that DMA data transfer takes place on. SCSI Services provides the following service functions for mapping and unmapping buffers:

- `escsi_dma_map`: Performs `dma_sync`, provides DMA mapping services, and performs buffer alignment as appropriate.
- `escsi_dma_unmap`: Unmaps DMA mapping and performs required DMA syncs to prevent stale data.

DMA mapping services use a `escsi_dma_parms_t` structure. This structure is associated with I/O and contains stateful information about DMA mapping for I/O, for example, the scatter list and the gather list. This information is used during DMA sync and DMA unmapping of I/O. The `escsi_dma_parms` structure holds data used by SCSI DMA services to manage DMA for I/Os. A SCSI interface driver is expected to allocate an `escsi_dma_parms` structure for each I/O that is set up for DMA. This structure must not be reused for the next I/O, until the previous I/O has been unmapped.

Before calling `escsi_dma_alloc`, a SCSI interface driver must allocate the memory required for the scatter or gather list and specify its pointer in the `iovec_array` field in `escsi_dma_parms`. SCSI Services uses the `iovec_array` field to store DMA mapping for I/Os. SCSI interface drivers can specify the memory size required for `iovec` as part of `if_scb_size` in `escsi_ifsw_t`, which is allocated as part of the `if_scb` allocation by SCSI Services.

DMA mapping services perform the following alignment handling before mapping:

1. If the SCSI interface driver does not specify any alignment requirement, a default cacheline boundary alignment is done for read I/Os. Write I/Os are not aligned in this case.
2. If the SCSI interface driver specifies an alignment requirement, then both read and write I/Os are aligned if necessary. The interface driver can request that the data be aligned either on a cacheline boundary or page boundary.

While calling the `escsi_dma_map` service, an interface driver can specify flags in the `dma_flags` parameter to indicate the type of DMA mapping required by the driver. These flags include:
DMA_REQ_DEFAULT_ALIGN
Default alignment policy. The default policy is to only align read requests on a cacheline boundary. Write requests are not aligned. Do not use the default alignment if the SCSI interface driver required ifsw->if_beg_align and ifsw->if_end_align in versions prior to HP-UX 11i v3.

DMA_REQ_CACHE_ALIGN
Indicates that the beginning and ending address must be cacheline aligned for all I/Os (reads and writes).

DMA_REQ_PAGE_ALIGN
Indicates that the beginning and ending address must be page aligned for all I/Os (reads and writes).

DMA_REQ_IOVEC_SPLIT4GB
Indicates that an iovec that straddles the 4 GB address space must be split.

SCSI interface drivers managing cards that support only 32-bit DMA can request SCSI Services to allocate memory that is within the 32-bit DMA range. For drivers that must be allocated 32-bit DMA capable data buffers, SCSI services provides the following functions:

escsi_dma32_map
Allocates a 32-bit DMA capable buffer, copies information for writes to the allocated buffer, and maps the allocated buffer for DMA.

escsi_dma32_unmap
Unmaps the 32-bit DMA capable data buffer, copies information back to the I/O buffers for reads, and deallocates the 32-bit DMA capable data buffer.

Tracing Services
HP-UX 11i v3 SCSI Services provides common tracing macros for tracing useful data and code execution paths in SCSI interface drivers. These macros are invoked by SCSI interface driver functions to trace important code paths and data values. The trace macros enable SCSI Services to trace 64-bit values representing the data values being handled in the context of the function invoking the macros.

Tracing can be done on a controller or at a LUN object level. Interface drivers are expected to use the controller trace buffer for their tracing. To use the tracing service, a SCSI interface driver must specify IF_CTLR_TRACING in the if_flags field of the escsi_ifsw_t structure. The driver must also specify the trace buffer size (in bytes) required in the trc_len field of the escsi_ifctrlr_attr_t structure passed during controller registration.

The trace information is stored in a circular buffer attached to the SCSI controller (or LUN) object. SCSI Services provides the following tracing macros:

ESCSI_TRC(__tbuf, __nparms, __v1, __v2, __v3)
Traces up to three 64-bit unsigned values.

ESCSI_TRC_VERBOSE(__tbuf, __nparms, __v1, __v2, __v3)
Conditional macro traces up to three 64-bit unsigned values. This takes effect only when the _ESCSI_DEBUG conditional compilation switch is enabled. Use this for verbose tracing in the debug kernel.

The values can be integers or pointers. The __tbuf parameter specifies the controller trace buffer where the trace information is logged. The __nparms parameter specifies the actual number of values to be traced. The 64-bit values to be traced are passed in __v1, __v2, and __v3. For example, if the caller must trace just one value, then the macro can be called with __nparms as 1, __v1 as the value to be traced, and __v2, and __v3 as zeros.

Use the following function to allocate and initialize (or resize) a trace buffer:
escsi_trc_init
Initializes a trace buffer structure.

Use the following function to deallocate a trace buffer:
escsi_trc_uninit
Uninitialize a trace buffer structure.
SCSI Command Services

The `escsi_cmdx_ext` service enables interface drivers to issue SCSI commands such as an INQUIRY or REPORT LUNS to devices, while transparently providing SCSI Services support for tagged versus untagged devices, queue depth, and other device level issues. The interface driver can use this service to issue asynchronous I/Os that provide a callback function to notify the caller of I/O completion, instead of waiting or blocking for the I/O completion.

The `escsi_cmdx_ext` service enables an interface driver to issue commands (CDBs) directly to a LUN and as an option, to specify the LUN path used to issue the I/O. The `escsi_cmdx_ext` takes a `cdbinfo` structure pointer as an argument. The caller is expected to allocate and initialize the `cdbinfo` structure. The caller can issue path specific I/Os or LUN I/Os (I/Os that can be delivered to the LUN on any available I/O through any available HBA or target port accessible to the LUN), and can specify whether the I/O is synchronous or asynchronous. For synchronous I/Os, the call returns after completion of the I/O. For asynchronous I/Os, the caller specifies a callback function to be called after the I/O completes.

The `cdbinfo` structure, `escsi_cdbinfo_t`, contains the information necessary to issue the I/O. This interface allows various modules in the mass storage stack (SCSI Services, class drivers, and interface drivers) to issue SCSI CDBs directly to a device. The `escsi_cdbinfo_t` structure contains details such as the SCSI command to be issued, the `buf` structure associated with the I/O, the number of retries, the timeout value, the LUN or LUN path information, and the error code.

The following services are provided to allocate, initialize or reinitialize, and free `cdbinfo` structures:

- `escsi_cdbinfo_alloc` Allocates a `cdbinfo` structure and the associated `buf` structure, and initializes the `cdbinfo` structure with the information that is passed in.
- `escsi_cdbinfo_init` Reinitializes a `cdbinfo` structure.
- `escsi_cdbinfo_free` Frees a `cdbinfo` structure and its associated `buf` structure.

For more information on the `escsi_cmdx_ext` service, see the HP-UX 11i v3 Driver Development Reference.

Linked List Services

SCSI Services provides a predefined set of macros for linked list management. The purpose of these macros is to simplify and standardize the management of doubly linked lists in the SCSI stack. These macros provide services to manage a doubly linked circular list, with a header element pointing to its head and tail.

The basic building block of a linked list is the `escsi_list_t` structure:

```c
typedef struct escsi_list {
    struct escsi_list *flink; /* forward pointer */
    struct escsi_list *blink; /* back pointer */
    void *parent; /* pointer to parent structure */
} escsi_list_t;
```

The following macros manage linked lists that contain `escsi_list_t` elements:

- Get the next element or the previous element to a given list element or header:
  - `ESCSI_NEXT(__elem)`
  - `ESCSI_PREV(__elem)`
- Initializes a linked list header to point to itself:
  - `ESCSI_INIT_HDR(__hdr)`
- Initialize linked list element pointers:
  - `ESCSI_INIT_ELEM(__elem)`
- Set or get the parent pointer of a linked list element:
ESCSI_GET_PARENT(__elem)
ESCSI_SET_PARENT(__elem, __parent)

- Confirm whether a linked list is empty or not:
  - ESCSI_IS_EMPTY(__hdr)
  - ESCSI_IS_NOT_EMPTY(__hdr)

- Queue a linked list element at the tail or head of the queue:
  - ESCSI_ENQ_AT_TAIL(__hdr, __elem)
  - ESCSI_ENQ_AT_HEAD(__hdr, __elem)

- Queue an element before or after a specified linked list element:
  - ESCSI_ENQ_BEFORE(__next_elem, __elem)
  - ESCSI_ENQ_AFTER(__prev_elem, __elem)

- Append or prepend one list to another list:
  - ESCSI_PREPEND_LIST(__hdr1, __hdr2)
  - ESCSI_APPEND_LIST(__hdr1, __hdr2)

- Move the list pointed to by one header to another header:
  - ESCSI_MOVE_LIST(__from_lh, __to_lh)

- Unqueue a linked list element:
  - ESCSI_DEQ(__elem)

- Unqueue an element from the head or tail of a linked list:
  - ESCSI_DEQ_FROM_HEAD(__hdr, __elem)
  - ESCSI_DEQ_FROM_TAIL(__hdr, __elem)

- Quickly unqueue an element from the head or tail of a linked list:
  - ESCSI_DEQ_FROM_HEAD_FAST(__hdr, __elem)
  - ESCSI_DEQ_FROM_TAIL_FAST(__hdr, __elem)

The escsi_is_on_q linked list service function is available to SCSI interface drivers to confirm whether a given element is on a specified linked list.

External Data Structures

The following SCSI Services data structures are used in SCSI interface drivers:

**Controller/Target Registration**

- `escsi_ifctlr_reg_t` Controller registration structure
- `escsi_ifctlr_attr_t` Controller attributes
- `escsi_ifctlr_stat_t` Controller statistics
- `escsi_ifctlr_type_t` Protocol types: physical or logical HBA
- `escsi_addr_t` SCSI instance address
- `escsi_name_t` SCSI component name (for example, target port name and target node name)

- `escsi_iftgt_reg_t` Target registration structure
- `escsi_iftgt_attr_t` Target attributes
- `escsi_iftgt_stat_t` Target statistics
- `escsi_reg_leg_tgt_t` Legacy target path registration
- `escsi_reg_leg_LUN_t` Legacy LUN path registration
- `escsi_get_leg_LUN_info_t` Get address, inquiry data, and serial number of a LUN port
escsi_ifsw_t       SCSI interface switch structure
escsi_if_flags_t   Flags for tag management and tracing

Lunpath Open/Close
escsi_ifspoc_ws_t  SCSI LUN path open and close arguments

I/O Path
escsi_iobj_t       SCSI I/O object
escsi_scb_flags_t  scb structure definition
escsi_scb_t        SCSI command block structure

Task Management
escsi_task_t       Task management function definitions (for example, ABORT_TASK_SET)
escsi_tm_arg_t     SCSI task management arguments

Asynchronous Event Notification
escsi_async_evt_t  Public asynchronous event notification definitions
escsi_iobj_cpu_t   Parameter of the ESCSI_IOBJ_CPU_REG operation
escsi_if_iobj_t    Parameter of the ESCSI_IOBJ_CPU_REG operation
escsi_aen_t        Asynchronous event notification structure

DMA Services
escsi_dma_req_flags_t  DMA alignment and I/O vector split requirements
escsi_dma_parms_t       DMA mapping parameters

SCSI Command Services
escsi_cdbinfo_t       Information to issue a SCSI command
esctl_io_t            I/O request specifics
escsi_cdbinfo_flags_t  Flags specifying how SCSI services is to handle I/O

Tracing
escsi_trc_rec_t       Trace record structure
escsi_trc_buf_t       Trace buffer structure

Miscellaneous
escsi_list_t         Doubly linked list element
escsi_type_t         SCSI object types (for example, controller, target path, LUN path, and LUN)
escsi_event_t        SCSI events

Header Files
An interface driver can obtain all the SCSI Services interfaces by including the following header file:
<io/escsi_services.h>
The following two necessary additional header files are included by escsi_services.h:
<io/escsi_common.h>
<io/escsi_macros.h>
SCSI Interface Driver Development

SCSI interface drivers operate within the HP-UX WSIO infrastructure and the SCSI stack, interfacing with HBA hardware (or a virtual HBA interface) at the bottom side of the driver, and with SCSI Services at the top side of the driver, as described in Chapter 17 (page 397). WSIO provides the general infrastructure for driver installation and configuration, and also provides the interfaces needed for hardware to access PIO and DMA. SCSI services provide connection interface services between SCSI class drivers and SCSI interface drivers, and additional services for probing, flow control, and other SCSI-related activities.

Like other WSIO drivers, a SCSI interface driver must be:
1. Installed (or loaded)
2. Configured (attached) to a particular HBA port (controller)
3. Initialized

These three steps are described for WSIO drivers in Chapter 4 (page 77). SCSI interface driver-specific details are addressed in the following sections of this chapter.

After initialization, a SCSI interface driver registers its HBA controller with SCSI Services and probes for targets and LUNs through that HBA controller. The probing sequence and associated registration requirements are expained in “Controller Registration and Probing ” (page 421). Next, or in conjunction with probing, the interface driver can receive opens, closes, reads, writes, ioctls, and other commands or SCSI task management requests. I/O completions and other hardware notifications are received in the driver through its Interrupt Service Routine (ISR), which is registered by the driver during attach or initialization. Certain high availability (HA) events can also be supported in a SCSI interface driver by using the drv_event_handler interface, introduced in “Conditional Entry Points” (page 406). For information about this interface, see “Driver Installation and Uninstallation” (page 413) and “High Availability Event Handling” (page 444).

The aspects of a SCSI interface driver discussed above provide for the basic operation of the driver. The remainder of this chapter contains a detailed presentation of these topics as follows:

- Driver installation (install or load) and uninstallation (unload)
- HBA controller configuration (attach) and initialization (init)
- Controller registration and probing (ctlr_node_cb, if_ctlr_probe, tgt_reg_done_cb, if_leg_hwp_build)
- SCSI open and close handling (if_lpt_open and if_lpt_close)
- SCSI request handling (if_start, if_task_mgmt, if_ioctl, d_ioctl)
- SCSI completion handling (ISR and internal error conditions)
- Asynchronous event handling
- HA events (drv_event_handler)

NOTE: The SCSI interface driver controls a SCSI initiator port, which is called a SCSI HBA controller in HP-UX 11i v3. The interface driver also discovers and registers target paths, which in SCSI are known as an I-T nexus.

Driver Installation and Uninstallation

A WSIO driver must have a driver_install routine, a driver_load routine, or both. The install routine statically installs the driver into the kernel and the load routine dynamically loads the driver into the kernel using the DLKM interface.

During install the following tasks are completed:
• Any driver global data is initialized.
• The driver’s attach routine links into the global attach list.
• If the driver supports any HA events (for example, OL*) it also calls
  `wsio_install_drv_event_handler` to register its HA event handler.

The following pseudocode shows the basic elements of an install routine for the `xyz` driver:

```c
int (*xyz_saved_attach)();  /* Saved function pointer */
extern int (*pci_attach)(); /* Global attach list */
void xyz_install()
{
    /* Link attach routine into global attach list */
    xyz_saved_attach = pci_attach;
    pci_attach = xyz_attach_linked;

    /* Register driver with WSIO */
    wsio_install_driver(&xyz_wsio_info);

    /* Register HA event handler */
    wsio_install_drv_event_handler(&xyz_wsio_info, xyz_event_handler);
}
```

`xyz_attach_linked` is the driver function actually linked into the global attach list, with the following pseudocode implementation:

```c
xyz_attach_linked( id, isc )
{
    /* Call the actual attach routine */
    xyz_attach(id, isc);
    /* Call the next driver in the list */
    return xyz_saved_attach(id, isc);
}
```

`xyz_wsio_info` is a static global data structure in the driver that contains pointers to driver information structures of type `drv_info_t`, `drv_ops_t`, and `wsio_drv_data_t`. The following pseudocode illustrates the implementation of these structures for SCSI interface drivers, and identifies mandatory flags and values:

```c
#define XYZ_DRIVER_NAME "xyz"
#define XYZ_DRV_FLAGS ( DRV_SAVE_CONF /* Mandatory */
| DRV_MP_SAFE /* Mandatory */
| DRV_SCAN /* Mandatory */
| DRV_DEFER_SCAN /* Optional */
| DRV_CHAR ) /* Mandatory */
```

The `DRV_CHAR` flag must be set to support the creation of a controller device special file (DSF) of type character. For parallel scanning of the HBA controller, the `DRV_SCAN` and `DRV_DEFER_SCAN` flags must be set in the `drv_info_t` structure.

```c
static drv_info_t xyz_info = {
    XYZ_DRIVER_NAME, /* Name of the driver */
    "ext_bus", /* Class of the driver */
    /* Note: SCSI interface drivers newly developed for
    * HP-UX 11iv3 must adopt the "escsi_ctlr" class. */
    XYZ_DRV_FLAGS, /* Driver flags */
    -1, /* Block major device number */
    -1, /* Character major device number */
    NULL, /* Drivers set to NULL */
    NULL, /* Drivers set to NULL */
```

414 Writing a SCSI Interface Driver
The block major number is ignored because the HBA controller is a character device. The character major number is set to -1 to get a dynamic major number.

```c
/* cdevsw entry points for HBA controller device files */
static drv_ops_t xyz_ops = {
  xyz_open,   /* d_open     Mandatory */
  xyz_close,  /* d_close    Mandatory */
  NULL,       /* d_strategy */
  NULL,       /* d_dump    */
  NULL,       /* d_psize   */
  NULL,       /* d_mount   */
  NULL,       /* d_read    */
  NULL,       /* d_write   */
  xyz_ioctl,  /* d_ioctl    Mandatory */
  NULL,       /* d_select  */
  NULL,       /* d_option1 */
  NULL,       /* reserved1 */
  NULL,       /* reserved2 */
  xyz_drv_cb, /* drv_ops callback Mandatory */
  NULL,       /* reserved4 */
  C_ALLCLOSES /* d_flags    Mandatory */
| C_MGR_IS_MP /*            Mandatory */
| C_OPAQ_DEV  /*            Mandatory */
};
```

```c
static wsio_drv_data_t xyz_data = {
  "scsi",          /* drv_path */
  T_INTERFACE,     /* driver type (defined in conf.h) */
  DRV_CONVERGED,   /* driver flags */
  NULL,            /* must be set to NULL */
  NULL,            /* must be set to NULL */
};
```

```c
static wsio_drv_info_t xyz_wsio_info = {
  &xyz_info,
  &xyz_ops,
  &xyz_data,
  WSIO_DRV_CURRENT_VERSION
};
```

For more information about `driver_install`, `wsio_install_driver`, and related data structures, see Chapter 5 (page 137).

A driver that supports DLKM must provide load and unload entry points. The load routine is called instead of the install routine if the driver is dynamically loaded into the kernel. The load routine allocates global resources required by the driver, registers the driver with WSIO, and registers the driver's attach routine. A dynamically allocated `drv_info_t` structure is passed to the load routine, replacing the statically defined structure. Most of the initialization code can be shared between the install and load routines, with the exception of the code to register the attach routine.

**NOTE:** Drivers that support both static and dynamic loading must provide both an `install` routine and a `load` routine.

The unload routine frees all resources allocated by the driver and unregisters the driver from WSIO. It must ensure that all the interfaces controlled by it are quiesced.

The following pseudocode illustrates the basic elements of the load and unload routines.

```c
int xyz_load(drv_info_t *drv_infop)
{
  xyz_wsio_info.drv_info = drv_infop;
```

SCSI Interface Driver Development  415
/* Register driver with WSIO */
if(wsio_install_driver(&xyz_wsio_info) == 0) {
    return ENXIO;
}
if(wsio_install_drv_event_handler(&xyz_wsio_info,
    xyz_event_handler) != WSIO_OK) {
    wsio_uninstall_driver(&xyz_wsio_info);
    return ENXIO;
}
if(mod_wsio_attach_list_add(MOD_WSIO_PCI, xyz_attach, "xyz"
    xyz_attach, xyz_wsio_info.drv_info->name) != WSIO_OK) {
    wsio_uninstall_drv_event_handler(&xyz_wsio_info,
        xyz_event_handler);
    wsio_uninstall_driver(&xyz_wsio_info);
    return ENXIO;
}
.
.
.
return 0;
}

int
xyz_unload(drv_info_t *drv_infop)
{
    /* If the driver is busy, return EBUSY */
    mod_wsio_attach_list_remove(MOD_WSIO_PCI, xyz_pci_attach);

    /* Quiesce the driver and the hardware it controls. */
    .
    .
    .

    /* Release any resources allocated by the load routine. */
    .
    .
    .
    wsio_uninstall_drv_event_handler(&xyz_wsio_info,
        xyz_event_handler);
    wsio_uninstall_driver(&xyz_wsio_info);
    return 0;
}

HBA Controller Configuration and Initialization

The driver_attach routine is called by WSIO to attach a driver to an HBA controller. An ID for the HBA controller is passed to the attach routine. The ID is the first four bytes of the PCI configuration space for PCI HBAs, which the driver must match against its list of supported HBAs. If the ID matches one of the driver's supported HBAs, the driver must call wsio_claim_node to claim the controller. Otherwise, the driver must return without claiming. In either case, the driver must call the next attach routine in the attach chain (see the previous example). If the HBA controller is supported by the driver, but an error that prevents it from operating occurs after it is claimed using wsio_claim_node, the driver must set the INIT_ERROR flag in the isc->if_info->flags field.

The following pseudocode illustrates the implementation of an attach routine for the xyz SCSI interface driver.

NOTE: Only some of the error cases are handled.

int
xyz_attach(
    uint32_t id,
    struct isc_table_type *isc)
{
int status;
int intr_cnt = 1;
intr_loc_t *loc_list;
kmem_arena_attr_t attr;
kmem_handle_t xyz_lisc_arena;
mpt_isc_t *lisc = NULL;
wsio_addr_handle_t cfg_handle = NULL;
wsio_reg_node_t node_info;
wsio_ret_code_t ret;
wsio_reg_info_t *registers;

/* Check for the supported HBA controllers */
if ( ! XYZ_SUPPORTED_IDS(id) ) {
    /* HBA not supported by this driver */
   return WSIO_OK;
}

/* Assign HBA controller localities (cell-based system optimization) */
status = wsio_intr_req_loc(isc, intr_cnt, WSIO_INTR_CARD_LOC);
if (status == WSIO_OK) {
    status = wsio_intr_get_loc(isc, &intr_cnt, &loc_list);
}

/* Allocate the local isc structure (lisc) */
kmem_arena_attr_init(&attr, sizeof(kmem_arena_attr_t));
attr.kat_flags = KAT_MULTICACHE_SIZE;
if (clm_ok == WSIO_OK) {
    attr.kat_loc = loc_list[0].loc;
}
xyz_lisc_arena = kmem_arena_create((size_t)sizeof(xyz_isc_t),
    "XYZ_ISC", &attr, 0);
lisc = kmem_arena_alloc(xyz_lisc_arena, M_WAITOK);

/* Initialize the lisc */
bzero(lisc, (size_t)sizeof(xyz_isc_t));
lisc->xyz_lisc_arena = xyz_lisc_arena;
isc->if_isc = (void *) lisc;
lisc->isc = isc;

/* Map PCI configuration space */
ret = wsio_map_cfg_handle(isc, &cfg_handle);
if (ret != WSIO_OK) {
    xyz_attach_cleanup(isc, lisc);
    return WSIO_ERROR;
}
lisc->cfg_handle = cfg_handle;

/* Claim the HBA controller with WSIO */
node_info.version = WSIO_DRV_CURRENT_VERSION;
node_info.drv_info = &xyx_wsio_info
node_info.type = T_INTERFACE;
node_info.cb_func = xyz_ctlr_node_cb;
node_info.cb_arg = (void *)isc;
node_info.events_mask = CB_SCAN_ALL | CB_MK_DEV | CB_GET_DEVS;
node_info.name = XYZ_DRIVER_NAME;
(void) bcopy((char *)&id, (char *)&(node_info.id[0]), sizeof(id));
node_info.desc = "4Gb Fibre Channel";
node_info.isc = isc;
ret = wsio_claim_node(isc->card_node, &node_info);

/* Create the health property on the HBA controller node */
status = prop_create(isc->card_node,
    P_NO_EXPORT | P_NOTIFY | P_NO_COPY | P_NO_SLEEP,
    "health", HEALTH_ONLINE, strlen(HEALTH_ONLINE)+1);
/* Obtain and map the MMIO (Memory Mapped) Registers */
registers = wsio_get_all_registers(isc);
if (registers == NULL)
    .
    .
ret = wsio_map_reg(isc, &registers[0]);

/* Make sure the WSIO INIT_ERROR flag is cleared */
((struct wsio_if_info *)(isc->if_info))->flags &= ~INIT_ERROR;

/* Register init routine (checking the gfsw first) */
if (isc->gfsw == NULL)
    /* Error out */
    CONNECT_INIT_ROUTINE(isc, xyz_init);

/* Register supported HA events (suspend/resume for OLR) */
xyz_event_mask = (WSIO_EVENT_SUSPEND | WSIO_EVENT_RESUME);
status = wsio_reg_drv_capability_mask(isc, mpt_event_mask);
status = wsio_set_parm(isc, WSIO_HW_SUSPEND_TIMEOUT, (void *)xyz_tmo);
status = wsio_set_parm(isc, WSIO_HW_RESUME_TIMEOUT, (void *)xyz_tmo);

/* HBA controller successfully claimed */
return WSIO_OK;
}

The node_info passed to wsio_claim_node contains cb_func, cb_arg, and events_mask fields, which are used to provide driver callbacks when certain events occur on the HBA controller I/O tree node. The following event opcodes are mandatory for SCSI interface drivers:

- **CB_SCAN_ALL**
- **CB_MK_DEV**
- **CB_GET_DEVS**

The **CB_SCAN_ALL** event, in conjunction with the DRV_SCAN and DRV_DEFER_SCAN drv_info flags, initiates scanning (probing) of the HBA controller for targets and LUNs. See “Controller Registration and Probing” (page 421). The **CB_MK_DEV** and **CB_GET_DEVS** events are used to create dev_t for the HBA controller. In **CB_MK_DEV**, the driver creates the dev_t without any device options. **CB_GET_DEVS** creates all the dev_t for a given device, including those with device-specific options. Many interface drivers do not have options for their controller-level dev_t. Thus the same dev_t is created for both events. For the **xyz_ctlr_node_cb** pseudocode implementation details, see “Controller Node Callback” (page 424).

### Controller I/O Node Health Attribute

The SCSI interface driver creates and updates the HEALTH attribute of the controller I/O node to reflect the current HEALTH status of the controller. In the attach entry point, the driver must create the health attribute and set it to HEALTH_ONLINE by calling prop_create. When the controller state changes, the driver must update the HEALTH attribute accordingly by calling prop_modify with the P_NOTIFY flag set so GIO can generate an EVM event.

Table 18-1 lists the semantics of the I/O node health attributes.

<table>
<thead>
<tr>
<th>I/O Node Health Attribute</th>
<th>SCSI Interface Driver Controller Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>HEALTH_ONLINE</td>
<td>Link is online.</td>
</tr>
<tr>
<td>HEALTH_OFFLINE</td>
<td>Link is dead (offline exceeding time threshold).</td>
</tr>
<tr>
<td>HEALTH_BROKEN</td>
<td>PCI error, firmware is dead.</td>
</tr>
<tr>
<td>HEALTHLIMITED</td>
<td>(Optional) The link is performing suboptimally.</td>
</tr>
</tbody>
</table>
Table 18-1 I/O Node Health Attributes (continued)

<table>
<thead>
<tr>
<th>I/O Node Health Attribute</th>
<th>SCSI Interface Driver Controller Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>HEALTH_TESTING</td>
<td>Loopback or other forms of level 0 diagnostic test are performed on the controller.</td>
</tr>
<tr>
<td>HEALTH_DISABLED</td>
<td>A PCI online replacement suspend operation was performed on the controller, or the controller was disabled by an HBA-specific command; for example, <code>fcmsutil disable</code>.</td>
</tr>
<tr>
<td>HEALTH_STANDBY</td>
<td>(Optional)</td>
</tr>
</tbody>
</table>

HBA Controller Initialization

After successfully claiming the HBA controller and returning from the attach routine, WSIO calls the driver’s initialization routine, which is registered during the attach routine. The initialization routine initializes the HBA controller, allocates necessary data structures (including shared DMA structures), sets up interrupts, and so on. The HBA controller becomes operational when the initialization routine is completed. If the controller is not operational or cannot be made operational, the driver must fail the initialization routine by returning `WSIO_ERROR`. HBA controller registration is also performed in the initialization routine, as noted in the following pseudocode. HBA controller registration is described in “Controller Registration” (page 422).

```c
int
xyz_init(
    struct isc_table_type *isc)
{
    xyz_isc_t *lisc = isc->if_isc;
    xyz_hba_t *hba_info = lisc->hba_info;
    char desc_buf[96];

    /* Set wsio_inst and clear driver statistics */
    lisc->wsio_inst = wsio_isc_to_instance(isc, NULL);
    bzero(&lisc->stat, (size_t) sizeof(xyz_stat_t));

    /* Create the arenas used in the xyz driver */
    if( (xyz_arena_create(lisc)) != XYZ_SUCCESS ) {
        xyz_init_cleanup(isc, NULL);
        return WSIO_ERROR;
    }

    /* Allocate and activate interrupt objects */
    status = wsio_intr_alloc(isc, (wsio_drv_isr_t) xyz_isr,
        (uintptr_t) lisc, (uint64_t) 0, &lisc->wsio_intr_line);
    status = wsio_intr_set_irq_line(isc, lisc->wsio_intr_line,
        (intptr_t) WSIO_IRQ_LINE_AUTO, (uint64_t) 0L);
    status = wsio_intr_activate(isc, lisc->wsio_intr_line);

    /* Allocate DMA handles and set DMA attributes */
    hba_info->ctrl_dma_handle = wsio_allocate_dma_handle(isc);
    hba_info->io_dma_handle = wsio_allocate_dma_handle(isc);
    /* Examples of DMA attribute settings */
    /* Turn off prefetch on control structures */
    status = wsio_set_dma_attributes(isc, hba_info->ctrl_dma_handle,
        WSIO_DMA_ATTR_PREFETCH, 0L);
    /* Set aggressive prefetch on I/O buffer structures */
    status = wsio_dma_set_device_attributes(isc, NULL,
        WSIO_DMA_ATTR_PREFETCH, 2L);
    /* Set 64-bit DMA */
    status = wsio_dma_set_device_attributes(isc, NULL,
        WSIO_DMA_ATTR_ADDR_WIDTH, 64L);
    /* Set DMA Bandwidth attribute to 4096 to improve throughput */
    status = wsio_dma_set_device_attributes(isc, NULL,
        WSIO_DMA_ATTR_BANDWIDTH, (wsio_dma_attr_param_t)4096);
}
```

SCSI Interface Driver Development 419
/* Set ioscan description string */
(void)snprintf(desc_buf, 96, "My U320 SCSI 2-port Adapter");
status = prop_modify(isc->card_node, 0, "description", desc_buf,
(size_t)(strlen(desc_buf) + 1UL));

/* Initialize HBA controller hardware */
xyz_init_hba_hw(lisc);
isc->my_address = lisc->scsi_parms->scsi_initiator_id;

/* Calculate max_io_size (hardware or scatter-gather-list limit)*/
lisc->if_max_io_size = (NBPG * hba_info->max_sge_nos;

/* Register the HBA controller with the SCSI Services. */
status = xyz_ctlr_reg(lisc);

/* Do other registrations with the SCSI Services and kmio. */
status = xyz_misc_reg(lisc);

/* Initialize and start the driver’s watchdog timer. */
/* Show usage of kernel timer interfaces here. */
xyz_init_timers(lisc);
xyz_start_wdog_timer(lisc, (int(*)(caddr_t)xyz_wdog_timer,
(caddr_t)lisc, XYZ_TIMER_TICKS);

/* Allocate target structures needed in if_ctrl_probe. */
xyz_tgt_alloc(lisc);

/* HBA controller successfully initialized. */
return (WSIO_OK);
}

Parallel SCSI Parameter Settings

Parallel SCSI requires some state information about HBA controllers to persist across system reboots. This state information must be retrieved during initialization. HBA controller state information resides in the I/O card firmware, the system firmware, or both, depending on the platform. On PA-RISC platforms, SCSI parameters are set through processor-dependent code (PDC) and the card is not initialized with these values. The driver programs the card with the SCSI parameters to match the values set through the PDC. A SCSI interface driver must access the PDC to get the SCSI parameters. To protect drivers from possible 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 `SCSI_GET_INITIATOR_PARAMS` macro, do not make any modifications to it.

Include the `scsi_params_macro.h` file in the DDK to access this macro.

On Integrity systems, SCSI parameters are kept in NVRAM on the HBA and can be retrieved by the driver and programmed into the operating parameters of the HBA controller.

The `SCSI_GET_INITIATOR_PARAMS` macro is called with three arguments: a pointer to the `isc` structure, a place holder for the initiator ID, and a place holder for the SDTR period. The initiator ID also must be set in `isc->my_address`. The `SCSI_GET_INITIATOR_PARAMS` macro
works with the sample SCSI interface driver in the DDK. It must be modified to work with a different driver.

⚠️ **CAUTION:** If this macro is called from within a DLKM driver, the static `device_path` structure must be dynamically allocated from equivalently mapped memory to work properly.

## Controller Registration and Probing

The controller registration and probing sequence begins during the attach and initialization process. It uses a sequence of asynchronous callbacks as the sequence progresses. Figure 18-1 illustrates the overall sequence.

**Figure 18-1 Controller Registration and Probing Sequence**

1. From the `attach` routine, the SCSI interface driver calls `wsio_claim_node` to claim the HBA controller and register the controller I/O node callback.
2. WSIO invokes the driver's initialization routine, which in turn calls `escsi_ctlr_reg` to register the controller.
3. The controller probing sequence follows, beginning with WSIO invoking the interface driver's controller I/O node callback with the `CB_SCAN_ALL` event to scan the controller for targets and LUNs.
4. The driver's controller node callback calls `escsi_ctlr_node_cb` to do initial SCSI Services processing.
5. The driver is called back in its `if_ctlr_probe` routine.
6. The driver issues transport-specific probes to discover targets, each of which are registered by calling `escsi_tgt_reg`.
7. When all of the targets have been probed, and the ones that are found have been registered, the driver calls `escsi_if_ctlr_prb_cbfn` to indicate the controller probe is complete.
Each of the steps in the controller registration and probing sequence is described in detail, with code snippets showing example implementations, in the following sections:

- “Controller Registration” (page 422)
- “Controller Node Callback” (page 424)
- “Target Registration Information” (page 426)
- “Controller Probing” (page 427)
- “Controller and Target Unregistration” (page 429)
- “Target Path I/O Node Callback” (page 430)
- “Legacy Hardware Path Support” (page 431)
- “Controller Management LUN Support” (page 433)
- “Asynchronous Device Discovery” (page 433)

Controller Registration

A SCSI interface driver must register its controller with SCSI Services during initialization. It does this by initializing an `escsi_ifctrlr_reg_t` structure and passing it to `escsi_ctlr_reg`. During registration, the SCSI stack expects controller attributes to be set by the interface driver in the `escsi_ifctrlr_attr_t` structure, which is linked to `escsi_ifctrlr_reg_t`. The interface driver must also initialize the `escsi_ifsw_t` switch table structure with information about driver entry points that can be called by SCSI services.

The `escsi_ifctrlr_reg_t` structure is shared by both the SCSI interface driver and SCSI Services. Most of the fields are input to SCSI Services during registration. One field, `addr.c`, is output from SCSI Services back to the driver. Another field, `trc_info`, is for use by SCSI Services for controller-level tracing. The `attr` and `stat` fields point to structures of type `escsi_ifctrlr_attr_t` and `escsi_ifctrlr_stat_t`. The content of `escsi_ifctrlr_attr_t` and `escsi_ifctrlr_stat_t` is owned by the SCSI interface driver, but it can be referred to by SCSI Services. The driver must update these structures continuously to keep the attributes and statistics current.

SCSI interface drivers can belong to different driver classes; for example, `fc`, `ext_bus`, and `escsi_ctlr`. For this reason, SCSI services cannot rely on the driver class instance number to identify a controller. Two SCSI interface drivers belonging to two different driver classes can have the same instance number. When SCSI interface drivers invoke `escsi_ctlr_reg`, SCSI Services assigns a unique controller software identifier, which is returned to the SCSI interface driver in `escsi_ifctrlr_reg_t->addr.c` if the registration succeeds. This identifier is unique across all controllers on a given system. However, it is not persistent across system reboots and must not be used for any form of persistent identification, such as syslog error logging, `dev_t` encoding, or device special file encoding. The controller software identifier can be used by the SCSI interface driver to specify the controller instance in an `escsi_addr_t` structure. This is used to obtain an `escsi_ifctrlr_reg_t`, `escsi_iftgt_reg_t`, or private interface LUN path structure instance using the `escsi_ifctrlr_get`, `escsi_iftgt_get`, or `escsi_iflpt_get` services.

The `num_iobj` field in `escsi_ifctrlr_attr_t` specifies the number of I/O resource objects to be instantiated for the controller. An I/O object (`iobj`) encapsulates all I/O related resources used in the data path. It is accessed as the handle to start I/O. This model supports drivers that use a traditional single interrupt per HBA controller, and drivers that use multiple queue pairs and a corresponding number of MSI/MSI-x vectors per HBA controller. When a SCSI interface driver calls `escsi_tgt_reg`, it specifies the number of I/O objects it needs in `num_iobj`. Based on this value, SCSI Services creates the required number of I/O objects for the controller managed by that driver. A driver that does not support the multiple queue pair model (uses the traditional single interrupt per HBA controller), must set `num_iobj` to 1.

The following pseudocode provides an example of controller registration for a parallel SCSI driver:
int
xyz_ctlr_reg(
    xyz_isc_t *lisc)
{
    int status;
    escsi_ifsw_t *ifsw = &(lisc->ifsw);
    escsi_ifctlr_reg_t *ifctlr;

    /* Allocate the ifctlr structure and substructures. */
    lisc->ifctlr = kmem_arena_varalloc(lisc->xyz_ctlr_arena,
        (size_t)sizeof(escsi_ifctlr_reg_t), M_WAITOK);
    /* Allocate the controller attribute structure. */
    lisc->ifctlr->attr = kmem_arena_varalloc(lisc->xyz_ctlr_arena,
        (size_t)sizeof(escsi_ifctlr_attr_t), M_WAITOK);
    /* Allocate the controller statistics structure. */
    lisc->ifctlr->stat = kmem_arena_varalloc(lisc->xyz_ctlr_arena,
        (size_t)sizeof(escsi_ifctlr_stat_t), M_WAITOK);
    /* Zero the newly allocated structures. */
    bzero(lisc->ifctlr, (size_t)sizeof(escsi_ifctlr_reg_t));
    bzero(lisc->ifctlr->attr, (size_t)sizeof(escsi_ifctlr_attr_t));
    bzero(lisc->ifctlr->stat, (size_t)sizeof(escsi_ifctlr_stat_t));

    /* Initialize the ifsw structure. */
    ifsw->if_flags = IF_CTLR_TRACING;
    ifsw->if_tm_dsbl = 0; /* Support all task management operations. */
    ifsw->if_max_tags = 0; /* Create tags internally, */
    ifsw->if_tag_size = 0;
    ifsw->if_scb_size = sizeof(xyz_lsp_t) + 8;
    ifsw->if_scb_bzero_size = ifctlr->ifsw->if_scb_size;
    ifsw->if_max_io_size = lisc->if_max_io_size;
    ifsw->if_timeout_msecs = 0; /* Zero latency for pSCSI. */
    ifsw->if_lpt_open = xyz_if_lpt_open;
    ifsw->if_lpt_close = xyz_if_lpt_close;
    ifsw->if_start = xyz_if_start;
    ifsw->if_task_mgmt = xyz_if_task_mgmt;
    ifsw->if_ioctl = xyz_if_ioctl;
    ifsw->if_ctlr_probe = xyz_if_ctlr_probe;
    ifsw->if_leg_hwp_build = xyz_leg_hwpath_build;

    /* Initialize the ifctlr structure. */
    ifctlr = lisc->ifctlr;
    ifctlr->ifsw = &(lisc->ifsw);
    ifctlr->version = ESCSI_IFCTLR_REG_VER;
    ifctlr->isc = lisc;
    ifctlr->if_ctlr = lisc;

    /* Initialize ifctlr attributes */
    ifctlr->attr->iport_name.iov_len = 0; /* Controller WWN, n/a for pSCSI */
    ifctlr->attr->port_id = isc->my_address; /* SCSI Initiator ID */
    ifctlr->attr->protocol = PARALLEL_SCSI;
    ifctlr->attr->driver_name = XYZ_DRIVER_NAME;
    ifctlr->attr->driver_desc = XYZ_DRIVER_DESC;
    ifctlr->attr->driver_rev = XYZ_DRIVER_REVISION;
    ifctlr->attr->fw_rev = lisc->fw_version_str;
    ifctlr->attr->cur_speed = XYZ_RATE_TO_STR(lisc->scsi_parms->speed);
    ifctlr->attr->max_speed = XYZ_MAX_SPEED;
    ifctlr->attr->addnl_trc_rec = 0;
    ifctlr->attr->num_iobj = 1; /* XYZ card does not support MSI-X */
    ifctlr->attr->capability = (ESCSI_BOOT_CAPABLE);
    ifctlr->attr->type = ESCSI_PHYSICAL; /*Physical controller present*/

    /* Register the HBA controller with SCSI Services. */
    status = escsi_ctlr_reg(lisc->ifctlr);
    if (status != ESUCCESS) {
        return (XYZ_FAILURE);
    }

    /* Register the controller’s WSIO instance number as a legacy bus
    with SCSI Services if the driver is supported on pre-HP-UX 11iv3 */
    status = escsi_reg_leg_bus(lisc->wsio_inst);
    /* Status of EOVERFLOW here means that the wsio_inst exceeds the
    legacy bus limit of 255. EOVERFLOW can therefore be ignored and

SCSI Interface Driver Development 423
NOTE: The if_scb_bzero_size must be kept small because zeroing the if_scb structure is in the performance path. Fields of the if_scb structure that are always initialized during I/O processing do not need to be zeroed.

For drivers other than parallel SCSI, additional attr fields need to be initialized. For example, the HBA controller world-wide name (WWN), where applicable, needs to be set in the iport_name field. Two capabilities are defined for the capability field, ESCSI_BOOT_CAPABLE and ESCSI_DUMP_CAPABLE. However, dump interfaces are not available to third-party interface drivers; you can specify only ESCSI_BOOT_CAPABLE.

After registering the controller, the driver must register its interrupt CPU bindings with SCSI Services. See “I/O Object Registration” (page 443). The SCSI interface driver must also set up a kmio_handle for the controller instance for performance measurement. These registrations are shown in the following pseudocode:

```c
int xyz_misc_reg(xyz_isc_t *lisc)
{
    int status;
    intptr_t cpu_id;
    escsi_aen_t aen;
    escsi_iobj_cpu_t iobj_cpu;
    km_info_t km_info;
    int major_num = xyz_wsio_info.drv_info->c_major;
    uint32_t my_dev = makdev(major_num << 24, |
                              M_INSTANCE24_TO_MINOR(lisc->wsio_inst));

    /* Register the interrupt CPU binding with SCSI Services. */
    /* Obtain the CPU assignment and notify SCSI Services. */
    status = wsio_intr_get_assigned_cpu(lisc->wsio_intr_line, &cpu_id);
    iobj_cpu.cpu_id = (spu_t)cpu_id;
    iobj_cpu.iobj_id = 0;
    aen.version = ESCSI_AEN_VER;
    aen.addr.c = lisc->ifctlr->addr.c;
    aen.addr.t = aen.addr.l = 0;
    aen.evt = ESCSI_IOBJ_CPU_REG;
    aen.evt_data = (intptr_t)&iobj_cpu
    status = escsi_if_aen(&aen);

    /* Save for later use (lisc->iobj is of type escsi_iobj_t). */
    lisc->iobj = escsi_iobj_get(lisc->ifctlr, 0);

    /* Allocate a kmio_t structure for this controller instance. */
    my_dev = (major_num << 24) |
              M_INSTANCE24_TO_MINOR(lisc->wsio_inst);
    km_info.collate_cb = NULL;
    km_info.dev = xyz_dev;
    km_info.dev_type = D_CHR;
    km_info.km_token_type = KM_DEV;
    lisc->kmio_handle = KM_io_alloc(&km_info);
    KM_io_first_open( (kmio_t *) lisc->kmio_handle );

    return (XYZ_SUCCESS);
}
```

Controller Node Callback

After the attach and initialization have completed successfully, the driver is invoked in its controller node callback function with event opcode CB_SCAN_ALL to begin the controller probe...
sequence, scanning the controller for targets and LUNs. This is the callback function registered
with `wsio_claim_node` in the `attach` routine.

Upon receipt of the controller node CB_SCAN_ALL, the driver must call the SCSI Services controller
node callback, `escsi_ctlr_node_cb`. Next, SCSI Services does some processing and calls back
into the driver on its `ifsw->if_ctlr_probe()` entry point to probe the controller for its targets.

The following pseudocode provides an example of the controller node callback including
implementation of the CB_MK_DEV and CB_GET_DEVS events.

```c
int
xyz_ctlr_node_cb(
    void *handle,
    void *arg,
    io_events_t event,
    uintptr_t event_info)
{
    io_make_dev_t          *make_devp;
    io_dev_info_t          *dev_infop_array;
    io_dev_arr_t           *dev_arryp;
    struct isc_table_type  *isc;
    xyz_isc_t              *lisc;

    switch (event) {

        case CB_SCAN_ALL:
            return(escsi_ctlr_node_cb(handle, event, event_info));

        case CB_MK_DEV:
            spinlock(xyz_global_lock);
            isc  = (struct isc_table_type *)arg;
            lisc = (xyz_isc_t *)isc->if_isc;
            if (lisc == NULL)
                return GIO_ERROR;
            make_devp = (io_make_dev_t *)event_info;
            if (make_devp->dev_type != D_CHR)
                return GIO_E_INVAL_ARGS;
            make_devp->dev = makedev(xyz_wsio_info.drv_info->c_major,
                    M_INSTANCE24_TO_MINOR(lisc->wsio_inst));
            spinunlock(xyz_global_lock);
            return GIO_SUCCESS;

        case CB_GET_DEVS:
            spinlock(xyz_global_lock);
            isc  = (struct isc_table_type *)arg;
            lisc = (xyz_isc_t *)isc->if_isc;
            if (lisc == NULL)
                return GIO_ERROR;
            dev_arryp = (io_dev_arr_t *)event_info;
            dev_infop_array = dev_arryp->array;
            if (*(dev_arryp->count) < 1) {
                *(dev_arryp->count) = 1;  /* Set how many are needed */
                dev_infop_array[0].dev = makedev(xyz_wsio_info.drv_info->c_major,
                        M_INSTANCE24_TO_MINOR(lisc->wsio_inst));
                dev_infop_array[0].dev_type = D_CHR;
                dev_infop_array[0].options = 0;
                *(dev_arryp->count) = 1;
                spinunlock(xyz_global_lock);
                return GIO_SUCCESS;
            }
            /* Only one dev_t per HBA controller */
            dev_infop_array[0].dev = makedev(xyz_wsio_info.drv_info->c_major,
                    M_INSTANCE24_TO_MINOR(lisc->wsio_inst));
            dev_infop_array[0].dev_type = D_CHR;
            dev_infop_array[0].options = 0;
            *(dev_arryp->count) = 1;
            spinunlock(xyz_global_lock);
            return GIO_SUCCESS;

        default:
```
printf("xyz_ctlr_node_cb: unsupported event \%ld\n", event);
}

return GIO_E_NOT_SUPPORTED;
}

The driver's if_ctlr_probe routine is invoked next in the probe sequence. It probes for target devices and determines their availability. Each time this routine is called, the targets must be probed again to validate that they are still available, or to detect new targets that have come online. Reprobing of targets previously discovered can be done with a lighter weight probe to validate their continued availability; for example, a simple INQUIRY command, rather than the entire DV process in parallel SCSI.

---

**TIP:** HP recommends a light weight reprobe to minimize overall scanning overhead, where appropriate.

---

**Target Registration Information**

The following data structures must be allocated and initialized for target registration:

- escsi_iftgt_reg_t: Main target registration structure
- escsi_iftgt_attr_t: Target attribute structure
- escsi_iftgt_reg_t: Target statistics structure

When a new target is discovered, it must be registered with SCSI Services. This requires an escsi_iftgt_reg_t structure and associated substructures. Because escsi_iftgt_reg_t is asynchronous, allocations done during a probe must be done with M_NOWAIT; however, these allocations can fail. The failures must be handled and retried to avoid ioscan NO_HW and other anomalous conditions. To ease the implementation of if_ctlr_probe, a driver can preallocate all necessary target structures at initialization, as noted in the initialization routine pseudocode. See “HBA Controller Initialization” (page 419).

---

**NOTE:** HP does not recommend the preallocation approach for drivers that have large and indefinite numbers of targets, for example Fibre Channel. These interface drivers must do M_NOWAIT allocations of target structures as they are discovered.

---

This implementation of target structure allocations and initializations is illustrated in the following example:

```c
int xyz_tgt_alloc(  
    xyz_isc_t *lisc)
{
    xyz_tgt_t *tgt;
    escsi_iftgt_reg_t *iftgt;
    escsi_iftgt_attr_t *my_attr;
    escsi_iftgt_stat_t *my_stat;

    for (tgt_id = 0; tgt_id <= XYZ_MAX_TGT_ID; tgt_id++)
    {
        tgt = &lisc->target[tgt_id];
        tgt->lisc = lisc;
        tgt->tgt_id = tgt_id;
        /* Allocate iftgt_reg and associated substructures. */
        iftgt_reg = kmalloc(sizeof(escsi_iftgt_reg_t), ...)
        my_attr = kmalloc(sizeof(escsi_iftgt_attr_t), ...)
        my_stat = kmalloc(sizeof(escsi_iftgt_stat_t), ...)
        if any of the three are null {
            /* Free any that did get allocated and return error. */
        }
        /* Initialize iftgt_reg and associated structures. */
    }
}  
```

---

426 Writing a SCSI Interface Driver
iftgt = tgt->iftgt_reg;
bzero the three structures
iftgt->attr = my_attr;
iftgt->stat = my_stat;
iftgt->version = ESCSI_IFTGT_REG_VER;
iftgt->if_tgt = tgt;
iftgt->if_tgt_reg_cbfm = (escsi_cbfunc_t)xyz_tgt_reg_done;
iftgt->addr.c = lisc->ifctlr->addr.c;
/* Target node callback function. */
iftgt->wsio_cb_func = xyz_tgt_node_cb;
iftgt->arg = NULL;
iftgt->events_mask = CB_SCAN_ALL | CB_DESTROY;
/* New hardware path elements */
iftgt->hwp.num_hwp_elms = 1;
iftgt->hwp.addr[0] = tgt->tgt_id;
/* Target attributes */
iftgt->attr->port_id = tgt->tgt_id;
iftgt->attr->iobj_id = 0;
iftgt->attr->protocol_rev = "";
/* Initialize other fields in the target structure. */
.
.
.
}
return XYZ_SUCCESS;
}

The hwp field in the escsi_iftgt_reg_t structure is designed to allow for drivers and protocols with widely varying sizes of the target hardware path element information. In the preceding xyz_tgt_alloc parallel SCSI example, only one 64-bit element is needed. This is the case for any protocol or situation where the target information is less than or equal to 64 bits in size. If the hardware path information is more than 64 bits, hwp accommodates a maximum of eight chunks of 64 bits or less. Except for the last chunk, all chunks are registered by SCSI Services as transparent I/O nodes. The last chunk of the target path hardware path information is represented by an I/O node owned by the SCSI Services estp driver.

The iobj_id field in the escsi_iftgt_attr_t structure binds the discovered target path to an iobj (corresponding to an interrupt source). The value of iobj_id can range from 0 to (num_iobj - 1), where num_iobj is the value specified in escsi_ifctlr_attr_t during controller registration. For more information, see “I/O Object Registration” (page 443).

Controller Probing

The if_ctlr_probe function launches target probe operations. The target probe operations complete asynchronously. For example, domain validation on parallel SCSI involves a sequence of INQUIRY commands to the target. When a target probe operation completes, the driver must call escsi_tgt_reg to register the target. The escsi_iftgt_reg_t target registration structure specifies a tgt_reg_done callback function that SCSI Services calls after target registration is completed. The following pseudocode examples show the if_ctlr_probe function, target registration, the tgt_reg_done callback, and signaling for target probe completion.

```c
void
xyz_if_ctlr_probe(
void *if_ctlr)
{
    xyz_iisc_t *lisc = (xyz_iisc_t *)if_ctlr;
    spinlock(hba_info->hba_lock);

    if (lisc->state == XYZ_SUSPENDED || lisc->state == XYZ_SUSPENDING) {
        /* Suspended; signal error completion to escsi. */
        spinunlock(hba_info->hba_lock);
        escsi_if_ctlr_prb_cbfm(lisc->ifctlr, EIO);
    }
```
/* Because the probe sequence is asynchronous keep a count of
  * how many probes are left to know when probe done. */
   lisc->tgt_probe_cnt = XYZ_MAX_TGT_ID + 1;

   /* Launch target probes (domain validation in u320 scsi). */
   for (tgt_id = 0; tgt_id <= XYZ_MAX_TGT_ID; tgt_id++) {
      xyz_tgt_t *tgt = &lisc->target[tgt_id];
      if (tgt_id == isc->my_address) {
         /* If this is the initiator id, register it to allow
            for discovery of pSCSI "controller management LUNs". */
         spinunlock(hba_info->hba_lock);
         escsi_tgt_reg(lisc->iftgt_reg);
         spinlock(hba_info->hba_lock);
      } else {
         xyz_tgt_probe(tgt); /* Launch target probe */
      }
   }
   spinunlock(hba_info->hba_lock);
   return;
}

For information on registering the initiator ID as a target, see “Controller Management LUN Support” (page 433).

Upon successful completion of a protocol-specific target probe sequence, the interface driver must call escsi_tgt_reg and pass the previously allocated and initialized iftgt_reg structure:

   escsi_tgt_reg(lisc->iftgt_reg);

**NOTE:** To properly display targets in ioscan and other utilities, the driver must register each successfully probed target on every invocation of if_ctlr_probe, even if the target was previously registered.

Upon successful target registration, SCSI Services sets the target path I/O node in the node field of the escsi_iftgt_reg_t structure, and sets the target path instance number in addr.t. Then the driver dynamically updates associated target attributes and statistics throughout the life of the target.

After target registration is processed, SCSI Services calls back to the driver with the tgt_reg_done callback.

```c
void
xyz_tgt_reg_done(
   escsi_iftgt_reg_t *iftgt_reg,
   int errno)
{
   xyz_tgt_t *tgt = &lisc->target[tgt_id];
   escsi_iftgt_reg_t *iftgt = tgt->iftgt_reg;
   escsi_ifctrlr_reg_t *ifctrlr = lisc->ifctrlr;
   spinlock(hba_info->hba_lock);
   if (errno == ESUCCESS) {
      if (!tgt->registered) {
         /* Newly discovered target. Increment statistics. */
         ifctrlr->stat->num_tgt_ports++;
         tgt->registered = TRUE;
      }
   }
   /* Decrement target probe count. */
   lisc->tgt_probe_cnt--;
   if (lisc->tgt_probe_cnt == 0) {
      spinunlock(hba_info->hba_lock);
   }
}
```
NOTE: The driver must call `escsi_if_ctlr_prb_cbfn` after all target probing and registration for a given controller probe is complete.

The driver must also handle all target probe failures as illustrated in the following example.

```c
void xyz_tgt_probe_error(
    xyz_tgt_t *tgt)
{
    xyz_tgt_t *tgt = &lisc->target[tgt_id];
    uint8_t all_done_needed = FALSE;
    uint8_t tp_offline_needed = FALSE;

    spinlock(hba_info->hba_lock);
    /* If going from target online to offline, notify SCSI Services. */
    if (tgt->registered) {
        tgt->registered = FALSE;
        tp_offline_needed = TRUE;
    }
    /* Decrement target probe count. */
    lisc->tgt_probe_cnt--;
    if (lisc->tgt_probe_cnt == 0) {
        all_done_needed = TRUE;
    }
    spinunlock(hba_info->hba_lock);
    if (all_done_needed) {
        /* Notify SCSI Services that probe is complete. */
        escsi_if_ctlr_prb_cbfn(lisc->ifctlr, ESUCCESS);
    }
    if (tp_offline_needed) {
        /* Send ESCSI_TP_OFFLINE event to SCSI Services. */
        aen.evt = ESCSI_TP_OFFLINE;
        aen.version = ESCSI_AEN_VER;
        aen.addr = ld->iftgt_reg->addr;
        aen.evt_data = NULL;
        (void) escsi_if_aen(&aen);
    }
}
```

NOTE: The `ESCSI_TP_OFFLINE` target path offline event must only be sent during target probing by interface drivers that can asynchronously detect when a target comes back online.

For each registered target path, SCSI Services discovers LUN paths and LUNs associated with the target path, without further involvement by the interface driver.

Controller and Target Unregistration

Targets must not be unregistered because of a probe failure of a previously registered target. Instead, an event must be sent to SCSI Services indicating that the target has gone offline. See the previous example. Similarly, targets must not be unregistered during PCI online replacement, suspend, or resume operations. However, there are three circumstances in which a driver must unregister targets and controllers:

- PCI online deletion of the HBA controller
- Unload of a SCSI interface driver through DLKM
- A CB_DESTROY event on the target node callback
As part of a PCI online deletion or unload, all target paths must be unregistered first, then the controller must be unregistered. The CB_DESTROY event, which occurs as a result of an rmsef command, requires the unregistration of a specific target.

A SCSI interface driver takes the following steps to unregister target paths and to unregister a controller:

1. For each registered target path, unregister the target path by calling escsi_tgt_unreg.
2. If target path unregistration succeeds, deallocate target path data structures allocated by the SCSI interface driver during registration.
3. Unregister the controller from SCSI services by calling escsi_ctlr_unreg.
4. If controller unregistration succeeds, deallocate controller structures allocated by the SCSI interface driver during registration.

Controller and target unregistration calls do not fail unless an invalid parameter is passed.

Target Path I/O Node Callback

Probing can be driven at the target path level, for example, when an ioscan -H is performed, in which case the driver’s target node callback function is called. As with the controller node callback, the driver must call the SCSI Services controller node callback escsi_tgt_node_cb to handle the discovery of LUN paths and LUNs associated with the target path, without further involvement of the interface driver.

Pseudocode for the target node callback and for handling the CB_DESTROY event on the target node is provided in the following example.

NOTE: CB_DESTROY occurs, for example, when an rmsef -H is performed.

CB_SCAN_ALL and CB_DESTROY are mandatory events for interface drivers on target nodes.

int xyz_tgt_node_cb(
    void *handle,
    void *arg,
    io_events_t event,
    uintptr_t event_info)
{
    switch (event) {
        case CB_SCAN_ALL:
            return(escsi_tgt_node_cb(handle, event, event_info));

        case CB_DESTROY: {
            xyz_tgt_t *tgt;
            escsi_addr_t tp_addr;
            escsi_type_t tp_type;
            escsi_iftgt_reg_t *iftgt_reg;

            /* Obtain the target structures for this node. */
            ret = escsi_node_to_addr(handle, &tp_addr, &tp_type);
            if (ret == ESUCCESS) {
                ret = escsi_iftgt_get(&tp_addr, (ulong_t)0, NULL,
                    ESCSI_GET_THIS, ESCSI_ADDR_LOOKUP, &iftgt_reg);
            }
            if (ret != ESUCCESS)
                return GIO_ERROR;
            tgt = iftgt_reg->if_tgt;

            /* Shut down the target. */
        }

        default:
            printf("xyz_tgt_node_cb: unsupported event %ld\n", event);
    }
Legacy Hardware Path Support

Upon completing a target probe, SCSI Services calls the `if_leg_hwp_build` entry point in the SCSI interface driver to build the legacy hardware paths associated with the probed target and discovered LUN paths. This provides backward compatibility support for legacy LUN path device files and hardware paths for OS versions prior to HP-UX 11i v3.

If the legacy bus node corresponds to the controller I/O node (for example, parallel SCSI), there is no need to register the legacy bus with WSIO. The legacy bus registration with SCSI Services following the controller registration in the driver initialization context suffices. However, if the legacy bus is a virtual bus (for example, Fibre Channel), it must be registered with both WSIO and SCSI Services in the context of `if_leg_hwp_build` by calling `wsio_reg_legacy`. See the following pseudocode example for implementation details.

When `REPORT LUNS` is not supported by a target, SCSI Services assumes the target supports SCSI-2 8 bit LUN IDs. SCSI Services creates 64-bit LUN IDs, where the 8-bit SCSI-2 LUN IDs are stored in the second most significant byte of each 64-bit LUN ID with an addressing method of 0 as follows:

<table>
<thead>
<tr>
<th>byte 7</th>
<th>byte 6</th>
<th>byte 5</th>
<th>byte 4</th>
<th>byte 3</th>
<th>byte 2</th>
<th>byte 1</th>
<th>byte 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LUN ID</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

In this case, the interface driver converts the 64-bit LUN IDs received from SCSI services into 8-bit LUN IDs by extracting the 8-bit LUN ID from the second most significant byte of the 64-bit LUN ID.

When several LUN addressing methods are used in the LUN ID list, use only the LUN addressing method used by the driver in HP-UX 11i v2 or earlier.

```c
void xyz_leg_hwp_build (  
    escsi_ifctlr_reg_t *ifctlr_reg, /* Controller registration structure. */  
    escsi_iftgt_reg_t  *if_tgt,     /* Target registration structure. */  
    uint64_t            num_LUNids, /* Number of LUN IDs in the list. */  
    escsi_LUNid_t      *LUNid_list) /* List of 64-bit LUN IDs. */  
){  
    xyz_isc_t *lisc = (mpt_isc_t *)ifctlr_reg->if_ctlr;  
    void *ctrl_node = lisc->isc->card_node;  
    wsio_legacy_info_t dom_leg_info;  
    wsio_legacy_info_t bus_leg_info;  
    
    1) Parse the list of LUN IDs to select the LUN addressing method.  
       For example, pre-11iv3 HP Fibre Channel interface drivers  
       used the following priority order:  
       (1) Volume set addressing (flat space addressing in SAM-3)  
       (2) Logical unit addressing  
       (3) Peripheral device addressing  
       
       for (i = 0; i < num_LUNids; i++) {  
           addressing = XYZ_LUN_ADDR_TYPE(LUNid_list[i].LUNid);  
           if (addressing == XYZ_VOLUME_SET_ADDRING) {  
               break;  
           }  
       }  
    
    If volume set addressing was found, use it. Otherwise use the  
    addressing method found on the last LUN in the list.  

    2) Build legacy T_INTERFACE hw_path based on the topology of the  
       target. Eg. for FC Fabric topologies the target's NPortID is 
```
a 24-bit value made up of \texttt{byte}_2 \texttt{byte}_1 \texttt{byte}_0 as follows:

\begin{itemize}
  \item Domain = \texttt{byte}_2
  \item Area = \texttt{byte}_1
  \item Port = \texttt{byte}_0
\end{itemize}

In pre-11iv3 HP FC drivers, each of these three bytes was used as successive hw_path elements, with the domain element being a T\_INTERFACE node. In this case therefore the domain node needs to be registered, as shown below.

\begin{verbatim}
bzero((caddr_t) &hw_path, sizeof(hw_path_t));
hw_path.addr[++hw_path.last_index] = domain;
io_init_hw_path(&hw_path, IO_TREE_LEGACY);
/* Register it with wsio */
rval = io_node_to_hw_path(ctrl_node, NULL, &hw_path);
bzero((caddr_t) &dom_leg_info, sizeof(wsio_legacy_info_t));
dom_leg_info.type = T\_INTERFACE;
dom_leg_info.drv_info = &xyz_fcp_wsio_info
dom_leg_info.name = "fcp";
dom_leg_info.id[0] = domain;
bzero(&dom_leg_info.id[1], sizeof(dom_leg_info.id) - 1);
bcopy(&hw_path, &dom_leg_info, sizeof(hw_path_t));
rval = wsio_reg_legacy(ctrl_node, &dom_leg_info);

/* Add the FC area and port elements to the hw path */
hw_path.addr[++hw_path.last_index] = area;
hw_path.addr[++hw_path.last_index] = port;
\end{verbatim}

3) Walk through the LUN id list and register each corresponding legacy bus, tgt, and LUN with the SCSI Services.

\begin{verbatim}
i = 0;
while (i < num_LUNids) {
  - get LUN id element from LUNid_list parameter

  - If LUN id addressing method does not match the selected addressing method, skip it.

  - If needed, invoke escsi_get_ing_serial() to get the Inquiry data and serial number of the LUN id.

  - Parse LUN id to identify (virtual) SCSI-2 busses/tgt/LUNs in a transport specific manner and initialize leg_bus_id, leg_tgt_id and leg_LUN_id accordingly.

  Suppose, for example, that the following mapping scheme was used in 11iv3 for Volume Set addressing:

  \begin{verbatim}
  7 6 5 4 3 2 1 0
  0 1| bus id
  ---------
  0 1| tgt id|LUN id
  ---------
  \end{verbatim}

  In this case the legacy bus, target, and LUN would be extracted from the two most significant bytes of the 64-bit LUN id.

  - Register the legacy bus, target, and LUN with SCSI Services by calling escsi_reg_leg_bus(), escsi_reg_leg_tgt(), and escsi_reg_leg_LUN(), respectively. Must call wsio_reg_legacy to register the legacy bus prior to calling escsi_reg_leg_bus,

}\end{verbatim}
Controller Management LUN Support

Certain transports such as parallel SCSI use a controller management LUN to issue SCSI commands to the initiator ID. This management LUN is typically a Processor peripheral device type. For example, parallel SCSI uses this management LUN to support the NILE protocol, which sends SCSI commands to manage the parallel SCSI Proliant system controllers. For example, power, fans, and device activity.

A management LUN device special file is only created for transports with the following characteristics:

1. Register a target path using `escsi_tgt_reg` corresponding to the target port ID matching the controller port ID stored in `isc->my_address`.
2. Successfully respond to the probing of LUN 0 beneath this target path by SCSI Services. This requires the following:
   a. A successful `ifs->if_lpt_open()` on the controller – target port – LUN 0 (C-T-L) nexus.
   b. A successful response to an INQUIRY command sent to LUN 0.

Additional probing I/Os such as EVPD INQUIRY pg0x80 and EVPD INQUIRY pg0x83 can fail.

If the above requirements are met, `insf` automatically creates a LUN device file corresponding to the management LUN. The management LUN is managed by the pass-through class driver.

If the interface driver doesn't support such management LUNs, `isc->my_address` must be set to 0xFF.

Asynchronous Device Discovery

The HP-UX 11i v3 SCSI stack supports asynchronous device discovery. Asynchronous device discovery enables automatic recognition and configuration of new devices that are added when a system is up and running. SCSI interface drivers can receive asynchronous notifications from a name server during topology changes (for example, RSCN in Fibre Channel fabric topologies) can initiate the discovery of new targets by calling the `wsio_async_scan` interface. This interface invokes the `if_ctlr_probe` routine that discovers and registers new targets. Newly discovered targets are probed for LUNs by SCSI services, and device files for those LUNs are created automatically.

SCSI Interface Driver Switch Table

The `escsi_ifsw_t` SCSI interface driver switch table structure contains information about driver entry points that can be called by the SCSI Services. SCSI interface drivers must provide the following mandatory entry points in the `escsi_ifsw_t` structure:

- `if_lpt_open` LUN path open interface
- `if_lpt_close` LUN path close interface
- `if_start` I/O start interface
- `if_task_mgmt` Task management interface
- `if_ctlr_probe` Controller probe interface
- `if_leg_hwp_build` Legacy hardware path build interface. This entry point must be provided by interface drivers that support legacy hardware paths and device files.
- `if_ioctl` Interface driver-specific support for ioctls issued on LUN device files. This entry point is required to support services management operations, such as LUN path deletion using the `SIOC_DEL_IFLPT` ioctl. It is also required to support legacy interface driver-specific ioctls on legacy device files.
If the driver sets the if_lpt field in escsi_ifspoc_ws_t to a driver-internal structure during its if_lpt_open routine, this linkage must be cleaned up during LUN path deletion. This can occur in the context of an rmaf of a LUN path, which results in SCSI Services invoking the if_ioctl entry point in the corresponding interface driver with the SIOC_DEL_IFLPT ioctl command.

In addition to the mandatory entry points, the escsi_ifsw_t structure includes the following fields for SCSI interface drivers:

- **if_flags**: Tag scope and tracing hints.
- **if_tm_dsbl**: Logical OR operation of task management opcodes not supported by the driver. The ABORT_TASK_SET opcode is mandatory and must not be set in this field.
- **if_max_tags**: Maximum number of concurrent qtags allowed. 0 means no limit in the interface driver. SCSI Services applies a maximum limit of 1K tags.
- **if_tag_size**: Tag size supported by the number of bits to be used. 0 means that no tags are allocated for the driver by SCSI Services.
- **if_scb_size**: Interface-specific scb structure size.
- **if_scb_bzero_size**: Number of bytes from the start of scb->if_scb that must be zeroed by SCSI Services. This value must be less than or equal to the if_scb_size value.
- **if_max_io_size**: Maximum I/O size (in bytes) supported by the HBA controller.
- **if_timeout_msecs**: Time required on this transport type for an I/O to reach the target. For transports with negligible latency, set this field to 0.
- **if_max_conc_ios**: Maximum number of I/Os that can be sent by SCSI Services to the I/O object select_q. If this is 0, SCSI Services does not apply flow control.

**SCSI Open and Close Handling**

SCSI interface drivers must provide a LUN path open interface function in the if_lpt_open entry point in the escsi_ifsw_t switch table structure. SCSI interface drivers must also provide a LUN path close interface function in the if_lpt_close entry point in the escsi_ifsw_t structure.

**Lunpath Open Interface**

SCSI Services uses the if_lpt_open interface to open a LUN path associated with a SCSI interface driver. The if_lpt_open function is asynchronous, and requires a callback to SCSI Services when the LUN path open is complete.

SCSI Services calls the LUN path open interface for every LUN path associated with a target path. On the first open on a target, which occurs after a target is initially registered during a LUN path probe initiated by SCSI Services, the SCSI interface driver must establish an I-T nexus with the target. For subsequent opens, it can increment an internal reference count.

The parameter passed to if_lpt_open is of type escsi_ifspoc_ws_t, which contains the escsi_addr_t of the LUN path to be opened. The driver must pass this value to the SCSI Services escsi_ifctlr_get and escsi_iftgt_get interfaces to get interface-specific data structures associated with this LUN path.

The LUN ID value in escsi_addr_t.l is always in the SCSI-3 64-bit LUN format. SCSI interface drivers that use SCSI-2 LUN ID formats must extract the SCSI-2 LUN ID byte from the second most significant byte of the SCSI-3 LUN ID, as described in “Legacy Hardware Path Support” (page 431).
SCSI interface drivers can allocate an interface-specific LUN path structure for each LUN path during the LUN path open, and place it in the if_lpt field of the escsi_ifspoc_ws_t structure to streamline the I/O path. When an I/O is sent to the LUN path, SCSI Services passes back the if_lpt pointer in the scb structure. This pointer can be used by a SCSI interface driver to access other interface-specific structures in the if_start path. It is not mandatory for SCSI interface drivers to use an if_lpt structure. But if it is not used, the SCSI interface driver must set if_lpt to NULL.

SCSI Services provides a completion callback function in the escsi_ifspoc_ws_t structure. This callback function must be invoked by the SCSI interface driver after the LUN path open operation is complete. The parameters of this callback function are escsi_ifspoc_ws_t->cbfn.cb_arg and the errno value indicating the status of the LUN path open operation.

To provide maximum parallelism in the SCSI stack, interface drivers must return as soon as possible from asynchronous entry points, including if_lpt_open, if_lpt_close, and if_task_mgmt, doing any extended processing asynchronous to the calling thread. SCSI interface drivers must not block in these entry points.

Lunpath Close Interface

The if_lpt_close interface is used by SCSI Services to close a LUN path associated with a SCSI interface driver. The if_lpt_close function is asynchronous, and requires a callback to SCSI services when the lunpath close is complete. The comments in “Lunpath Open Interface” (page 434) regarding the asynchronous if_lpt_open, not blocking, the use of the callback, and the use of the escsi_addr_t structure, also apply to if_lpt_close. During the last lunpath close on a target path, a SCSI interface driver must close the I-T nexus with the target.

SCSI Request Handling

This section describes the I/O start interface, high priority I/Os, the task management interface, the ioctl interfaces, and boot support for SCSI interface drivers.

I/O Start Interface

SCSI interface drivers must provide an I/O start interface routine in the if_start entry point of the escsi_ifsw_t structure. if_start is called by SCSI Services to issue I/Os (SCSI commands) to a LUN on a target path of the HBA controller. I/Os are issued to the interface driver from select_q, which is a queue of scb structures (escsi_scb_t). Multiple I/Os can be queued in select_q before the if_start routine is called. This enables multiple I/Os to be issued to the interface driver with a single call to if_start.

The input parameter for the if_start function is the escsi_iobj_t structure. The escsi_iobj_t structure contains the select_q and a pointer to the interface-specific controller structure (called lisc in this chapter). The escsi_iobj_t structure also contains a spinlock that synchronizes access to the select_q between SCSI Services and the interface driver. SCSI Services provide the ESCSI_IOBJ_LOCK and the ESCSI_IOBJ_UNLOCK macros to lock and unlock escsi_iobj_t while accessing the select_q. The interface driver must hold the escsi_iobj_t lock while removing I/Os from the select_q.

When one or more new I/Os are added to the an empty select_q, SCSI Services ensures that the if_start routine is called. If an interface driver leaves I/Os behind on the select_q after completing if_start, (for example, if no internal resources are available), the SCSI interface driver must check select_q again in its I/O completion path to start pending I/Os.

The timeout for each I/O is specified in the scb->max_msecs field. This timeout period refers to the time allowed for the I/O once it is issued to the HBA. However, if an I/O has a B_PFTIMEOUT flag set in its buf structure, the time must be measured from the time the driver received the request in select_q. The timeout period starts while the I/O waits for resources to become available.
in the interface driver. If the \texttt{scb->max\_msecs} field is specified as 0, the SCSI interface driver is not required to time that I/O.

High Priority I/Os

High-priority I/Os are not subject to flow control on the LUN path. The SCSI stack indicates high-priority I/Os to interface drivers through the \texttt{SCBF\_HIPRI} scb flag. SCSI interface drivers must give priority to these I/Os when allocating resources and issuing I/Os to the target. Some examples of high-priority I/Os are target/LUN probe related I/Os and I/Os issued to a LUN in the context of a LUN path open operation. High-priority I/Os must be handled ahead of other internally queued I/Os (whether on the \texttt{select\_q} or on driver internal queues) to avoid various responsiveness and scanning issues during congested conditions.

SCSI Services queues high-priority I/Os at the head of \texttt{select\_q}. If a SCSI interface driver is starting I/Os directly from \texttt{select\_q}, the priority is automatically guaranteed by the SCSI interface driver. If the SCSI interface driver maintains its own internal queue to start I/Os, it must guarantee prioritized handling. To guarantee prioritized handling of high-priority I/Os, a SCSI interface driver must keep its internal queue prioritized or maintain a separate queue specifically for high-priority I/Os. A SCSI interface driver is not required to process high priority I/Os in the order they are received in \texttt{select\_q}, provided all high-priority I/Os are started before any regular I/Os are started. When a SCSI interface driver removes I/Os from \texttt{select\_q}, it can add high-priority I/Os to the head of its internal queue and regular I/Os to the tail of its internal queue.

Task Management Interface

When upper layers require a task or some group of tasks to be aborted or terminated, a task management function is issued by way of the \texttt{if\_task\_mgmt} entry point. This entry point is used by SCSI Services to issue task management functions to the LUNs or targets associated with the controller. The following task management functions can be issued by SCSI Services:

- **\texttt{ABORT\_TASK\_SET}**: Aborts all I/Os on the specified I-T nexus. The driver must issue the \texttt{ABORT\_TASK\_SET} task management function to a specified LUN path, and abort all I/Os in its internal queues that are associated with the specified target path.

- **\texttt{CLEAR\_TASK\_SET}**: Aborts all I/Os on the specified I-T nexus, and can also abort I/Os on other initiators for that target depending on the state of the task set type (TST) field in the Control Mode page of the device. The driver must issue the \texttt{CLEAR\_TASK\_SET} to the specified LUN path and abort all I/Os in its internal queues that are associated with the target path.

- **\texttt{LUN\_RESET}**: Resets a specified LUN that is associated with a specified target, and aborts all I/Os from all initiators at the LUN. The driver must issue the \texttt{LUN\_RESET} to a LUN on the specified LUN path. It must also abort all I/Os to the LUN that are present in its internal queues.

- **\texttt{WARM\_TARGET\_RESET}**: Resets a specified target and aborts all I/Os from all initiators on the specified target. The I-T nexus (for example, target login) of the initiators associated with the specified target is not affected. This is similar to the SAM-2 TARGET RESET.

- **\texttt{COLD\_TARGET\_RESET}**: Resets a specified target and sends a transport-specific third party logout command to that target. All I-T nexuses for the specified target are terminated. All I/Os from all initiators currently associated with the target are aborted.

- **\texttt{BUS\_RESET}**: Resets the bus connected to the controller. This function is applicable only for bus-based transports such as parallel SCSI. This results in

436 Writing a SCSI Interface Driver
SCSI interface drivers can specify unsupport for specific task management operations with the if_tm_dsbl field in the escsi_ifsw_t SCSI interface switch table. The if_tm_dsbl field represents a logical OR operation value of task management codes that are not supported by the SCSI interface driver. Before issuing task management requests, SCSI Services checks the if_tm_dsbl field to confirm that the required task management command is supported by the SCSI interface driver. The ABORT_TASK_SET function is mandatory and must be supported by all SCSI interface drivers. SCSI interface drivers must also support the other task management functions listed above, unless the transport protocol or HBA hardware does not provide support for these task management functions. If any of the task management functions are not supported, the SCSI interface driver must use the if_tm_dsbl field to specify those functions as unsupported.

The task management interface is asynchronous. SCSI Services provides a callback function that must be called by the interface driver after a task management operation completes. Any I/Os aborted as a result of a task management operation that did not successfully complete before being aborted must be failed back to SCSI Services with the status SCTL_IO_ABORTED.

On any of the previous task management functions, the SCSI interface driver must examine its internal queue of pending I/Os and send errors to those I/Os affected by the task management operation. The SCSI interface driver must examine each I/O by comparing the lun_id, if_lpt, or if_tp fields in the scb to determine if it is impacted. The driver must send an error back to the scbs destined for the LUN path or target path being impacted.

When task management functions are issued, SCSI interface drivers must do the following (SCSI Services expects this behavior):

• Scan the select_q and any driver internal queues of pending I/Os, and send errors back to all I/Os for the LUN or target path, depending on the scope of task management.
  — For LUN_RESET, send errors back to I/Os on the lunpath.
  — For ABORT_TASK_SET, CLEAR_TASK_SET, WARM_TARGET_RESET, and COLD_TARGET_RESET, send errors back to I/Os on the target path.
  — For BUS_RESET, send errors back to all I/Os on the HBA controller.

• Issue the task management function and other operations as required to the device; for example, target logout for COLD_TARGET_RESET.

• If the task management response from a device indicates success, the SCSI interface driver must examine its internal queue of active I/Os, and abort all I/Os issued for the LUN or target path depending on the scope of task management.

• If a task management command fails, the SCSI interface driver must terminate the I-T nexus with the device and send an error back to all active I/Os on that target path, after a protocol-specified time period has passed, or after establishing a new I-T nexus.

• After all active I/Os relating to a particular task management command function have had errors sent back, the SCSI interface driver must invoke the task management callback function. The result of the task management operation is passed back to SCSI Services as an errno parameter to the callback function.

If the task management function fails, the SCSI interface driver takes a higher level, transport appropriate, error recovery path to ensure that the I/Os associated with the failed task management command are aborted at the target end.

For session-oriented SCSI transports, if a task management command fails, the SCSI interface driver must log out the I-T nexus and abort the I/Os in the task set in an explicit manner. The driver ensures that I/Os are explicitly aborted at the target end, before sending an error back to the I/Os. The driver issues a transport-specific logout (for example, LOGO/PLOGI for Fibre Channel) and send an error back to the I/Os only upon completion of the LOGO or PLOGI. If LOGO or PLOGI are unsuccessful, the SCSI interface driver must hold onto the I/Os until a
transport-defined safe time period has elapsed after the last response received from the target. Only then can it send errors to the I/Os. This safe time period (for example, 2*RA_TOV for Fibre Channel) is required to guarantee that the data is not present as ghost I/O in the interconnect fabric, loop, or bus after a host failover in an HA cluster.

SCSI interface drivers must guarantee that the task management callback function is called only after all pending and active I/Os for the specified task had errors sent back to SCSI Services. If the task management command fails, and the SCSI interface driver delays I/Os by a protocol-specified safe time period, the callback function must not be called until all delayed I/Os have errors sent back.

**ioctl Interfaces**

There are two ioctl interfaces into the SCSI interface driver:

- **ifsw->if_ioctl** Invoked for interface driver specific handling of ioctls received on LUN level device special files, for example, /dev/rdsk/c0t1d0 or /dev/rdisk/disk12.

- **cdevsw->d_ioctl** Invoked for ioctls received on HBA controller device files, for example, /dev/fcd0, instance 0 of the fcd driver’s HBA controllers.

The if_ioctl entry point is registered with SCSI Services during controller registration. The d_ioctl entry point is registered at driver installation through the drv_ops_t structure that is passed to wsio_install_driver.

**if_ioctl Entry Point**

The ifsw->if_ioctl entry point is used to handle legacy ioctls issued to legacy LUN path device special files that need interface driver support. All the legacy ioctls supported by SCSI interface drivers in ifsw->if_ioctl are still supported. SCSI Services selects the interface driver instance corresponding to the legacy LUN path device special file and forwards the legacy ioctls to the selected SCSI interface driver instance through the ifsw->if_ioctl entry point. For example:

- **Parallel SCSI interface drivers must support the following p SCSI-specific ioctls in the if_ioctl entry point:**
  - SIOC_RESET_BUS
  - SIOC_GET_TGT_P ARMS
  - SIOC_GET_BUS_P ARMS
  - SIOC_GET_TGT_LIM ITS
  - SIOC_GET_BUS_LIM ITS
  - SIOC_SET_TGT_LIM ITS
  - SIOC_SET_BUS_LIM ITS
  - SIOC_RESET_DEV
  - DIOC_RSTCR

- **FibreChannel drivers must support the following ioctls in the if_ioctl entry point:**
  
  - SIOC_GET_PORT_WWN Get the target port WWN of the target path corresponding to a given legacy LUN path device special file.
  - SIOC_GET_WWN Get the target node WWN of the target path corresponding to a given legacy LUN path device special file.

The ifsw->if_ioctl entry point is also used to handle management operations that SCSI Services forwards to SCSI interface drivers. For example, SIOC_DEL_IFLPT must be supported if the interface driver has set a non-NULL if_lpt during if_lpt_open. The SIOC_DEL_IFLPT ioctl enables an interface driver to delete a driver private LUN path object associated with a LUN path that is being deleted.
If the interface driver does not recognize an ioctl value, the `ifsw->if_ioctl` entry point must return -1. When SCSI Services receives an unrecognized ioctl, it forwards the ioctl first to the interface driver through its `ifsw->if_ioctl` entry point. If `ifsw->if_ioctl` returns -1, SCSI Services forwards the ioctl to the class driver using its `dd_ioctl` entry point.

**d_ioctl Entry Point**

The `d_ioctl` entry point is used as follows:

- To support legacy ioctls that were traditionally issued to the LUN path device file so they can also be issued on the controller device file where applicable. Interface driver supported legacy ioctls can be deprecated with respect to LUN-level device files in the future and must therefore be supported in `d_ioctl` wherever possible. Hence, legacy ioctls handled in the `if_ioctl` entry point must also be handled in the driver’s `d_ioctl` entry point. For example, parallel SCSI interface drivers must support the pSCSI-specific ioctls mentioned in “ioctl Interfaces” (page 438) on both the `if_ioctl` and `d_ioctl` interface.

- To support new ioctls for SCSI interface drivers. New ioctls for SCSI interface drivers cannot be issued to the new LUN device special files, because the new LUN ioctls are not be forwarded to interface drivers through the `if_ioctl` entry point. Instead, all interface driver ioctls are handled in the driver `d_ioctl` entry point. New ioctls required for boot support fall into this category, as described in “Boot Support” (page 439).

Table 18-2 summarizes the usage of the `if_ioctl` and `d_ioctl` entry points.

<table>
<thead>
<tr>
<th><code>if_ioctl</code></th>
<th><code>d_ioctl</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Legacy LUN path device file ioctls supported in pre-HP-UX 11i v3 through the <code>if_ioctl</code> entry point.</td>
<td>Legacy LUN path device file ioctls supported in the pre-HP-UX 11i v3 <code>if_ioctl</code> entry point.</td>
</tr>
<tr>
<td>SCSI Services management operations, for example, SIOC_DEL_IFLPT.</td>
<td>The ioctls required to support the <code>scsimgr</code> utility.</td>
</tr>
</tbody>
</table>

**Boot Support**

Interface drivers that must support boot devices through their HBA controllers must implement the following ioctls in their `d_ioctl` entry point. The `d_ioctl` entry point handles ioctls carried out on the controller `/dev/xyz` device file.

- `IOC_EFI2HW` Integrity Platform EFI to Hardware Path Conversion
- `IOC_HW2EFI` Integrity Platform Hardware to EFI Path Conversion
- `IOC_EFI2STR` Integrity Platform EFI Path to String Conversion

These ioctls convert Integrity firmware-specific boot paths to HP-UX I/O tree `hw_paths` and vice versa. Boot support is only provided on Integrity systems. It is not available on PA-RISC systems.

The following header file must be included for support of these ioctls:

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

This header file includes the following platform-specific header file:

```c
#include <machine/sys/kern_drv_iface_pdk.h>
```

These two header files are only shipped on Integrity systems.

The following three header files are also required:

- `efibind.h`
- `efidef.h`
- `efidevp.h`

You can download these EFI header files from the EFI industry web site at:

You can download a compressed file containing a distribution of files. Extract `efidef.h` and `efidevp.h` from the INC directory, and `efibind.h` from the INC/ipf directory. Then place the three files in `/usr/include/sys` on your development system. Add the following field to the end of the `EFI_DEV_PATH` union in the `efidevp.h` file to work with HP-UX:

```
char    max[1024]
```

The following pseudocode describes the boot conversion ioctls. The three ioctls must be under `#ifdef __ia64`.

```c
int32_t
xyz_ioctl (    
    dev_t dev,    
    int cmd,    
    caddr_t pdata,    
    int flags)  
{
    xyz_isc_t *lisc;
    mpt_isc_t *lisc_list;
    struct isc_table_type *isc;
    xyz_hba_t *hba_info;

    /* Obtain the lisc corresponding to this dev_t. */
    /* Driver needs to keep a global list of its xyz instances
     and walk through them to find this dev_t. */
    isc = lisc->isc;
    hba_info = lisc->hba_info;

    switch (cmd)  
    {
#ifdef __ia64
    case IOC_EFI2HW: {  
        struct ipf_efi2hw_data *efidata =  
            (struct ipf_efi2hw_data *)pdata;
        SCSI_DEVICE_PATH *dev_path = (SCSI_DEVICE_PATH *)efidata->node;
        uint16_t pun = dev_path->Pun;
        uint16_t lun = dev_path->Lun;
        xyz_tgt_t *tgt;

        if (dev_path->Header.SubType != MSG_SCSI_DP)
            return EINVAL;

        /* Swap the PUN and LUN fields to make them big endian. */
        SWAP_BYTES_INPLACE(pun);
        SWAP_BYTES_INPLACE(lun);

        /* Get the escsi_addr_t for the target. */
        tgt = &lisc->target[pun];
        my_addr = ((escsi_iftgt_reg_t *)(tgt->iftgt_reg))->addr;

        /* Add in the LUN ID in single byte peripheral device
         addressing in this example. */
        lun_id.lunid = 0;
        lun_id.byte[1] = lun & 0xff;
        my_addr.l = lun_id.lunid;

        /* Get the corresponding hardware path from the node. */
        rval = escsi_addr_to_node(&my_addr, ESCSI_LPT, &node);
        rval = io_node_to_hw_path(node, NULL, efidata->hw);
        break;
    }
# ifdef __ia64
    case IOC_HW2EFI: {  
        struct ipf_hw2efi_data *efidata =  
            (struct ipf_hw2efi_data *)pdata;
        SCSI_DEVICE_PATH *dev_path;
        xyz_tgt_t *tgt;

        /* Get the escsi_addr_t for the hardware path. */
        node = io_hw_path_to_node(NULL, efidata->hw);
        rval = escsi_node_to_addr(node, &my_addr, &type);

        /* Get the corresponding hardware path from the node. */
        rval = escsi_addr_to_node(&my_addr, ESCSI_LPT, &node);
        rval = io_node_to_hw_path(node, NULL, efidata->hw);
        break;
    }
```
if (type != ESCSI_LPT)
    return EINVAL;

/* Create the corresponding 64-bit LUN ID. */
lun_id.lunid = my_addr.l;
lun = lun_id.byte[1];

/* Get corresponding target information from escsi_iftgt_get(). */

tgt_id = tgt->tgt_id;

/* Construct the SCSI_DEVICE_PATH. */
bzero((caddr_t)&dev_path, sizeof(SCSI_DEVICE_PATH));
dev_path.Header.Type = MESSAGING_DEVICE_PATH;
dev_path.Header.SubType = MSG_SCSI_DP;
dev_path.Header.Length[0] = sizeof(SCSI_DEVICE_PATH);
dev_path.Header.Length[1] = 0;
dev_path.Pun = tgt_id;
dev_path.Lun = lun;

/* Swap the PUN and LUN in place to make little endian. */
SWAP_BYTES_INPLACE(dev_path.Pun);
SWAP_BYTES_INPLACE(dev_path.Lun);

if (sizeof(SCSI_DEVICE_PATH) <= efidata->buf_length) {
bcopy(&dev_path, efidata->buffer,sizeof(SCSI_DEVICE_PATH));
efidata->node_length = sizeof(SCSI_DEVICE_PATH);
} else {
    return EFIDP_NOSPACE_FAILURE;
} break;

/* Swap to make big endian. */
SWAP_BYTES_INPLACE( dev_path.Pun );
SWAP_BYTES_INPLACE( dev_path.Lun );

/* The output format will be Scsi(PunXX,LunXX). */
(void)snprintf((caddr_t)efidata->string_buffer,
    EFI_MAXIMUM_VARIABLE_SIZE,
    "%sX%sX%s",
    "Scsi(Pun",
    dev_path.Pun,
    ",Lun",
    "dev_path.Lun,
   ");
efidata->str_length = strlen(efidata->string_buffer);
break;
}

#endif /* __ia64 */

SCSI Interface Driver Development 441
default:
    status = EINVAL;
} /* switch( cmd ) */

return status;
}

**SCSI I/O Completion Handling**

Each of the request entry points previously described, except for the ioctl interfaces, are asynchronous interfaces that have an associated callback function. The ioctl interfaces are synchronous and completion status is returned as an errno value in the function return of the entry point. The errno status values are also returned on the completion of the if_lpt_open, if_lpt_close, and if_task_mgmt interfaces from an asynchronous callback. The if_start I/O interface uses the values associated with the cdb_status field in the scb structure for completion status. In addition to the SCSI status byte that comes from the device, these cdb_status values include the following:

- **SCTL_INVALID_REQUEST** The I/O request contains invalid parameters.
- **SCTL_SELECT_TIMEOUT** Parallel SCSI-specific. A SELECTION TIMEOUT occurs on the SCSI bus when trying to select a target.
- **SCTL_INCOMPLETE** The I/O request failed because of transient conditions.
- **SCTL_NO_RESOURCE** The I/O request cannot be issued because of lack of resources.
- **SCTL_TP_OFFLINE** The I/O request failed due to an I-T nexus failure.
- **SCTL_IO_TIMEOUT** The I/O request timed out.
- **SCTL_IO_ABORTED** The I/O request aborted as a result of a task management function issued by the if_task_mgmt function.
- **SCTL_RESET_OCCURRED** The I/O request failed because of a bus reset.

**Asynchronous Event Notification**

SCSI interface drivers must support asynchronous event notifications to SCSI Services. When an event occurs, a SCSI interface driver notifies SCSI Services by calling the escsi_if_aen interface. escsi_if_aen is passed a pointer to the escsi_aen_t structure, which contains asynchronous event codes and parameters. SCSI interface drivers must support the following asynchronous event notification codes:

- **ESCSI_TP_OFFLINE** Target path offline
- **ESCSI_TP_ONLINE** Target path online
- **ESCSI_IOBJ_CPU_REG** I/O object registration with CPU
- **ESCSI_IF_IOBJ_REG** Interface-specific I/O object registration
- **ESCSI_PORT_ID_CHANGE** Target port ID change

**Target Path Offline and Online**

A SCSI interface driver must asynchronously notify SCSI Services whenever a target path element goes offline or comes back online. This is done by calling escsi_if_aen with either the ESCSI_TP_OFFLINE or ESCSI_TP_ONLINE event. The escsi_addr_t SCSI instance address of the target path is passed in the escsi_aen_t parameter.

The following scenarios trigger target path offline events:

- HBA controller going offline because the transport link is dead
- Target path failure
- Target initiated termination of an I-T nexus
• PCI online replacement suspend operation on the controller
• HBA controller is disabled
• An existing target path is not being reported by a controller scan

The following scenarios trigger target path online events:
• HBA controller goes from offline to online
• Target path recovers from a failure
• PCI online replacement resume operation on the controller
• HBA controller enabled, leading to target path recovery

A target path offline notification must be made any time an unsolicited logout is received from a target or if the I-T nexus is lost. The `escsi_if_aen` interface must be invoked to notify SCSI Services of the event, independent of whether I/Os are pending on the target path. The SCSI interface driver can attempt to re-establish the I-T nexus following a target path offline asynchronous event notification. If successful, the SCSI interface driver must deliver a target path online asynchronous event notification to SCSI Services.

A target path offline event notification must be made any time the I-T nexus is lost, regardless of the cause. To avoid an avalanche of online and offline events during transient glitches, the driver must not send the `ESCSI_TP_OFFLINE` asynchronous event unless the target remains offline longer than a transient time period setting.

In a controller probe operation, when the driver completes the registration of the last target path, it must invoke `escsi_if_aen` to send an `ESCSI_TP_OFFLINE` event for every registered target path that was not successfully probed.

### I/O Object Registration

During controller registration, a SCSI interface driver specifies the number of I/O objects needed using the `num_iobj` field of the `escsi_ifctlr_attr_t` structure. These I/O objects must be bound to the CPU that is configured to handle interrupts for the controller. An I/O object is bound to a CPU using the `ESCSI_IOBJ_CPU_REG` asynchronous event. This `ESCSI_IOBJ_CPU_REG` event must be sent during initial CPU binding, or whenever interrupt migration is performed on the controller.

If a SCSI interface driver supports an MSI/MSI-X card, it can choose one of the following MSI-X vector distribution policies while registering a controller or target path component during device discovery:

• One vector per controller, shared by all target paths. A single I/O object is created and shared by all target paths under the controller.
• Multiple vectors per controller, distributed among target paths. Multiple I/O objects are used so the interface driver can achieve load balancing by distributing I/O objects among target paths.

When an interrupt I/O object is allocated to a SCSI interface driver during initialization, the driver queries WSIO for the assigned CPU ID. Then it notifies SCSI services to bind the I/O object to that CPU by sending the `ESCSI_IOBJ_CPU_REG` event through `escsi_if_aen`. The details of the I/O object and the CPU ID are entered in the `escsi_iobj_cpu_t` structure and a pointer to this information is passed in the `evt_data` field of `escsi_aen_t`. When interrupt migration occurs, SCSI interface drivers must bind I/O objects with the new CPU IDs by calling `escsi_if_aen`.

A SCSI interface driver can allocate its own interface specific I/O object data structure (`if_iobj`) to encapsulate all of its data structures for the specified queue pair and MSI/MSI-X interrupt vector. The `if_iobj` data structure can be attached to an I/O object using the `iobj->if_iobj` field. To attach its `if_iobj` to an I/O object, a SCSI interface driver must register its `if_iobj` by sending an `ESCSI_IF_OBJ_REG` asynchronous event. Information about the I/O object and `if_iobj` are saved in the `escsi_if_iobj_t` structure. A pointer to this information is passed in the `evt_data` field of `escsi_aen_t`.

SCSI Interface Driver Development 443
Target Port ID Change

SCSI interface drivers can also send an asynchronous event notification when a target port ID change is detected. When a SCSI interface driver detects that a target uniquely identified by a port name and a node name has undergone a change in its port ID, the driver can notify SCSI Services by calling `escsi_if_aen` to send an `ESCSI_PORT_ID_CHANGE` event on the affected target path. This asynchronous event notification must be sent only for target paths that support a unique port WWN and a unique node WWN. The `escsi_iftgt_reg_attr_t` structure associated with the target path must hold valid `port_name` and `node_name` information.

SCSI interface drivers that support agile addressing by handling target port movement within an interconnect can use the `ESCSI_PORT_ID_CHANGE` asynchronous event notification to inform SCSI Services about a port ID change for a specified target. Upon receiving this asynchronous event notification, SCSI Services updates the target path and target port data structures with the new port ID, and updates the `port_ID` field in the `escsi_iftgt_attr_t` target attribute structure.

High Availability Event Handling

The following high availability (HA) events can be optionally supported in a SCSI interface driver:

- **WSIO_EVENT_SUSPEND**: Suspend operation in preparation for a PCI online replacement or online deletion of an HBA.
- **WSIO_EVENT_RESUME**: Resume operation during the completion of a PCI online replacement of an HBA.
- **WSIO_EVENT_REMOVE**: Remove operation during the completion of a PCI online deletion of an HBA.

All HA events are handled by the driver's WSIO event handler entry point, which is registered in the install routine by a call to `wsio_install_drv_event_handler`. The set of supported events and associated attributes are registered in the attach routine. For more information, see the pseudocode in “Driver Installation and Uninstallation” (page 413).
In HP-UX 11i, the *sf commands—insf, lssf, mksf, and rmsf—support IHV drivers and the drivers known to them. This chapter explains how to write a shared library to get *sf command support for IHV drivers.

This chapter addresses the following topics:
- “Overview of *sf Commands”
- “Command Flow” (page 446)
- “Shared Library Creation” (page 447)
- “Sample Code” (page 450)

NOTE: I/O commands patch PHCO_36314 or a superseding patch must be installed.

Overview of *sf Commands

This section describes and provides examples of the commands that install, remove, make, and list special files.

The insf Command

The insf command installs device special files in the devices directory, usually /dev. If required, insf creates any subdirectories defined for the resulting device special file.

If no options are specified, device special files are created for all new devices in the system. New devices are those devices for which no instance numbers have been assigned. A subset of the new devices can be selected with the -C class_name, -d driver_name, and -H hw_path options.

With the -e option, insf reinstallersthe device special files for pseudo drivers and existing devices. This restores device special files when one or more have been removed.

For more information, see insf(1M).

Examples

Install device special files for all new devices belonging to the disk device class as follows:

```
# insf -C disk
```

Recreate device special files for all devices belonging to the disk device class as follows:

```
# insf -e -C disk
```

Install device special files for the new device added at hardware path 64000/0xfa00/0x0 as follows:

```
# insf -H 64000/0xfa00/0x0
```

Recreate device special files for the device at hardware path 64000/0xfa00/0x0 as follows:

```
# insf -e -H 64000/0xfa00/0x0
```

The lssf Command

The lssf command lists information about a device special file. For each device special file name, lssf uses the stat system call to determine the major number of the device special file and whether it is a block or character device file, then scans the kernel I/O system data for a device associated with the device special file. When the device is found, lssf displays a description of the device on standard output along with its hardware path (for example, address). These describe fields that are closely related to the options used with mksf.

Examples

Create a special file with the following command:
Issue the following `lssf` command:

```bash
# lssf /dev/disk/disk4_p9
esdisk section 9 at address 64000/0xfa00/0x0 /dev/disk/disk4_p9
For more information, see `lssf(1M)`. 
```

**The mksf Command**

The `mksf` command creates a device special file in the devices directory, usually `/dev`, for an existing device that has already been assigned an instance number by the system. Specify the device by supplying a combination of the `-C class_name`, `-d driver_name`, `-H hw_path`, and `-I instance_number` options. If the options specified match a unique device in the system, `mksf` creates a device special file for that device. Otherwise, `mksf` prints an error message and exits. If required, `mksf` creates any subdirectories relative to the device installation directory defined for the resulting device special file.

You can use the `mksf` command in the following ways:

- Create a device special file with a set of characteristics by supplying a subset of the driver options.
  
  These driver options are handled by the driver shared library. If a device special file name is specified, `mksf` creates the device special file with that name. Otherwise, it uses the default naming convention for the driver.

- Specify the minor number and special file name explicitly.
  
  This creates a device special file for a driver without using the built-in driver options in `mksf`. The `-r` option directs `mksf` to make a character (raw) device file instead of the default block device file for drivers that support both.

**Examples**

Create a special file with the following command:

```bash
# mksf -C disk -H 64000/0xfa00/0x0
For more information, see `mksf(1M)`. 
```

**The rmsf Command**

The `rmsf` command removes one or more special files from the `/dev` directory and potentially removes information about the associated device or devices from the system. Specify the device by supplying a combination of the `-C class_name`, `-d driver_name`, and `-H hw_path` options.

**Examples**

Remove the device at hardware path 64000/0xfa00/0x0 and its device special files as follows:

```bash
# rmsf -H 64000/0xfa00/0x0
For more information, see `rmsf(1M)`. 
```

**Command Flow**

When the `*sf` command is invoked, it checks the driver name with the built-in driver entry table. If the command finds an entry in the table, it creates, removes, or displays a device special file for that driver. Otherwise, the `*sf` command checks for IHV drivers by looking for a driver shared library in the `/usr/lib` directory (`libsfdriver-name.so` on Integrity systems or `libsfdriver-name.sl` on PA-RISC systems). If the `*sf` command finds a match, it makes the appropriate call to the shared library. If not, it returns an error message.

Figure 19-1 shows the `*sf` command flow.
Figure 19-1 Functional Interface

Where:

**Driver Entry Table**
A static array of driver names are built into each *sf command. The *sf command checks the table for known driver names. If one is found, a device special file is created for it.

**Ioconfig File**
A permanent storage area for I/O tree information saved across reboots. The insf command updates this file for new devices with information such as driver name, instance number, class, and hardware path.

**Device File**
Device special file.

**Shared Library**
If the *sf command does not find a driver in the Driver Entry Table, it checks for an IHV driver shared library. If found, it makes the appropriate call to it.

**Shared Library Creation**
IHV drivers need to provide a shared library to be supported by the *sf command.

When *sf commands do not find a driver name in the built-in driver entry table, they check for the IHV driver shared library. If the shared library exists, it makes an appropriate call by passing relevant information to the shared library function. That information is passed to the shared library function through the sf_info_t structure defined in the /usr/include/sfinfo.h file.

The sf_info_t structure is defined as follows:

```c
typedef struct sf_info {
    char   *class;
    char   *driver;
    char   *hw_path;
    char   *devfile_dir;
    char   *devfile;
    int    instance;
    int    verbose;
    int    b_major;
    int    c_major;
    dev_t  devt;
    void   (*makenode) __((char *, char, dev_t ));
    int    (*options_to_devt) __((int, char *, dev_t *));
    void   *reserved1;
    void   *reserved2;
} sf_info_t;
```

Where:

**class**
Class to which the device belongs. Initialized for the shared library by mksf and rmsf.
**Driver**

Driver name that has claimed this device. Initialized for the shared library by `insf`, `mksf`, `lssf`, and `rmsf`.

**hw_path**

Hardware address of the device. Initialized for the shared library by `insf`, `mksf`, and `lssf`.

**devfile_dir**

Device file directory. Initialized for the shared library by `insf` and `mksf`.

**devfile**

Device file name. Initialized for the shared library by `mksf` and `lssf`.

**instance**

Instance number of the device, a number unique to the class. Initialized for the shared library by `insf`, `mksf`, and `rmsf`.

**verbose**

Verbose output. Initialized for the shared library by `insf` and `mksf`.

**b_major**

Block major number. Initialized for the shared library by `insf` and `mksf`.

**c_major**

Character major number. Initialized for the shared library by `insf` and `mksf`.

**devt**

The `dev_t` value of the device. Initialized for the shared library by `lssf`.

**makenode**

Function to create a device special file. Initialized for the shared library by `insf` and `mksf`.

*Input parameters:*

- Special file name
- Type of special file (`b` or `c`)
- Device number (`dev_t`)

**options_to_devt**

Function to convert an option string to a `dev_t`. Initialized for the shared library by `insf` and `mksf`.

*Input parameters:*

- Device type
  - 0 Character device type
  - 1 Block device type
- Option string
  - Driver-specific options

*Output parameters:*

- `dev_t`

*Return:*

- `SUCCESS` or an error message.

**reserved1**

Reserved field.

**reserved2**

Reserved field.

---

**Shared Library Name and Location**

Shared libraries must be placed in the `/usr/lib/` directory. A shared library name must have the `libsfdriver-name.so` format for Integrity systems and the `libsfdriver-name.sl` format for PA-RISC systems.

**Example**

If the driver name is `&sample_driver_name`, the shared library name is the following:

- `/usr/lib/libsf&sample_driver_name.so` (Integrity)
- `/usr/lib/libsf&sample_driver_name.sl` (PA-RISC)
### Shared Library Supported Interfaces

- **driver-name_insff**
  Creates device special files. The insf command invokes this interface by passing the sf_info_t structure with relevant information to create the device special files.

  Syntax:
  ```c
  int driver-name_insff(
    sf_info_t *ihv_insff
  );
  ```

  Where:
  - `ihv_insff` Pointer to sf_info_t type.

- **driver-name_lssff**
  Displays information about the device. The lssf command invokes this interface by passing the sf_info_t structure with relevant information to display device special file information.

  Syntax:
  ```c
  int driver-name_lssff(
    sf_info_t *ihv_lssff
  );
  ```

  Where:
  - `ihv_lssff` Pointer to sf_info_t type.

- **driver-name_mksff**
  Creates a device special file. The mksf command invokes this interface by passing the sf_info_t structure with relevant information to create a device special file.

  Syntax:
  ```c
  int driver-name_mksff(
    sf_info_t *ihv_mksff,
    int nopt,
    char**dopt
  );
  ```

  Where:
  - `ihv_mksff` Pointer to sf_info_t type.
  - `nopt` Specifies the number of options.
  - `dopt` Pointer to a driver-specific option string.

- **driver-name_dataclean_rmsff**
  Removes information about the device. The rmsf command invokes this interface by passing the sf_info_t structure with relevant information.

  Syntax:
  ```c
  int driver-name_dataclean_rmsff(
    sf_info_t *ihv_rmsff
  );
  ```

  Where:
  - `ihv_rmsff` Pointer to sf_info_t type.

### Compiling a Shared Library

Use the following commands to create a shared library:
Sample Code

The following example shows the creation of a shared library for the &sample_driver_name driver.

```c
#include <stdio.h>
#include <errno.h>
#include <grp.h>
#include <pwd.h>
#include <string.h>
#include <sfinfo.h>
#include <sys/mtio.h>

#define SUCCESS 0
#define ERROR   -1

/* Function prototypes */
int mydriver_insf(sf_info_t *ihv_insf);
int mydriver_lssf(sf_info_t *ihv_lssf);
int mydriver_mksf(sf_info_t *ihv_mksf, int nopt, char **dopt);
int mydriver_sf(char *name, char *mkdev_options);
int mydriver_dataclean_rmsf(sf_info_t *ihv_rmsf);
void mydriver_usage();

/* Globals */
char  *progname="     ";  /* insf, lssf */
int  (*fptr_opt)();       /* Function pointer */
void (*fptr_mnode)();     /* Function pointer */
char  *drv_name;
dev_t  devt = 0;
int    node_instance = 0;
int    dev_type;
char   mkdev_options[16];
char  *usage = 
"usage: %s [-d %s | -C class] [-H hw_path] [-I instance] [-D dir] [-q|-v] \ 
[driver_options] [special_file]\n"

static void
mydriver_usage(void)
{
    (void)fprintf(stderr, usage, progname, drv_name);
}

/****************************
* Name: mydriver_sf()
*****************************
*
* Description:
* Common functionalities of insf and mksf are implemented in the
* mydriver_sf() function.
*
* Input parameters: device name, device-specific options.
*
* Return value: SUCCESS or ERROR
*
* Called by: mydriver_insf() and mydriver_mksf()
* 
```

**NOTE:** After creating the shared library, move it to directory named in “Shared Library Name and Location” (page 448)
static int
mydriver_sf(char *name, char *mkdev_options)
{
    char devname[80];

    /* Call the call back from sf_info_t to dev_t formed
     * with options. */
    if (fptr_opt(dev_type, (char *)mkdev_options, &devt) == ERROR)
        return (ERROR);

    /* form the DSF name, if not provided. */
    if (name == NULL)
        {
        name = devname;
        (void)sprintf(devname, "%s%d", "mydriver", node_instance);
        }

    /* Call the call back from sf_info_t to create the
     * device special file. */
    fptr_mnode(name, dev_type?'b':'c', devt);

    return (SUCCESS);
}

int
mydriver_insf(sf_info_t *ihv_insf)
{
    int errors = 0;

    fptr_opt = ihv_insf->options_to_devt;
    fptr_mnode = ihv_insf->makenode;
    node_instance = ihv_insf->instance;
    strcpy(progname, "insf");
    dev_type = 0; /* for char special file */

    /* Options for persistent type devices have to be
     * passed by converting the option bits to string. */

    /* for BEST (AT&T style device option) device special file */
    sprintf(mkdev_options, "%d", DEN_M_BEST);
    if (mydriver_sf("/dev/tape_BEST", mkdev_options) == ERROR)
errors++;

/* for BESTn (AT&T style with no rewind option) device special file */
sprintf(mkdev_options, "%d", (MT_NO_REWIND_MASK | DEN_M_BEST));
if (mydriver_sf("/dev/tape_BESTn", mkdev_options) == ERROR)
errors++;  

/* for BESTb (BSD style option) device special file */
sprintf(mkdev_options, "%d", (MT_BSD_MASK | DEN_M_BEST));
if (mydriver_sf("/dev/tape_BESTb", mkdev_options) == ERROR)
errors++;  

/* for BESTnb (BSD style with no rewind option) device special file */
sprintf(mkdev_options, "%d", (MT_BSD_MASK | DEN_M_BEST | MT_NO_REWIND_MASK));
if (mydriver_sf("/dev/tape_BESTnb", mkdev_options) == ERROR)
errors++;  

return (errors==0?SUCCESS:ERROR);
}

/*****************************************************************************************************/
/* Name: mydriver_lssf()                                                                 ***/
/*****************************************************************************************************/

int
mydriver_lssf(sf_info_t *ihv_lssf)
{
    devt = ihv_lssf->devt;
    strcpy(progname,"lssf");
    (void)printf("%s at address %s %s \n", ihv_lssf->driver,
    ihv_lssf->hw_path, ihv_lssf->devfile);
    return(SUCCESS);
}

/*****************************************************************************************************/
/* Name: mydriver_mksf()                                                                 ***/
/*****************************************************************************************************/

int
mydriver_mksf(sf_info_t *ihv_lssf)
{
    devt = ihv_lssf->devt;
    strcpy(progname,"mksf");
    printf("%s at address %s %s \n", ihv_lssf->driver,
    ihv_lssf->hw_path, ihv_lssf->devfile);
    return(SUCCESS);
}

452 Device Special File Support in Administration Tools
int mydriver_mksf(sf_info_t *ihv_mksf, int nopt, char **dopt)
{
    extern int optind;
    extern char *optarg;
    int i;
    int gargc;
    char **gargv;
    char *next_char;
    int error=0;
    int dev_opt=0;
    dev_type = 0; /* As this driver supports only char dsfs */
    strcpy(progname,"mksf");
    mkdev_options[0] = '\0';
    fptr_mnode = ihv_mksf->makenode;
    fptr_opt = ihv_mksf->options_to_devt;
    drv_name = ihv_mksf->driver;
    node_instance = ihv_mksf->instance;
    gargc = nopt;
    gargv = dopt;
    if(ihv_mksf->devfile_dir)
        chdir(ihv_mksf->devfile_dir);
    while ((i = getopt(gargc, gargv, "C:D:H:I:d:anqvr")) != EOF)
    {
        switch(i) {
            case 'a':
                dev_type=0;
                /* AT&T no rewind on close */
                dev_opt = DEN_M_BEST;
                break;
            case 'r': /* to create char dsf */
                dev_type=0;
                /* default : AT&T no rewind on close */
                dev_opt = DEN_M_BEST;
                break;
            case 'n':
                /* AT&T no rewind on close */
                dev_opt |= (MT_NO_REWIND_MASK | DEN_M_BEST);
                break;
            case '?':
                mydriver_usage();
                break;
            default:
                break;
        }
    }
    sprintf(mkdev_options, "%d", dev_opt);
    if(ihv_mksf->devfile)
        return (mydriver_sf(ihv_mksf->devfile, mkdev_options));
    return (mydriver_sf(NULL, mkdev_options));
}
This function can clean up or remove implementation-specific data structures associated with the node.
Rmsf invokes this function after deletion of device special files but before deletion of the node the kernel I/O data structures.

Input parameters: sf_info_t structure, number of driver specific options, and driver specific options.
Default members filled as part of sf_info_t: class, driver, hw_path and instance.
Return value: SUCCESS or ERROR

Called by: rmsf command

Calls to: mydriver_dataclean_rmsf()

******************************************************************************/

int mydriver_dataclean_rmsf(sf_info_t *ihv_rmsf)
{
    /* This routine is called before rmsf(1M) deletes the definition of the device from the system to let the driver delete or clean up any of its specific data structures.
    * Once this routine returns to rmsf(1M), rmsf deletes the definition of the device from the system.
    */

    return(SUCCESS);
}

454 Device Special File Support in Administration Tools
OL* and PCI Error Recovery Support in Interface Drivers

OL* and PCI Error Recovery use the same infrastructure to suspend and resume I/O cards. A driver must support online replacement as a prerequisite to supporting PCI Error Recovery. This chapter addresses the following topics:

- “OL* Support Requirements for Interface Drivers”
- “PCI Error Recovery Support Requirements for Interface Drivers” (page 469)

OL* Support Requirements for Interface Drivers

The ability to add, replace, or delete PCI I/O cards while a system is in use, without interrupting user services that are not directly affected by those I/O cards, is a necessary capability for high availability systems. The OL* infrastructure is implemented in hardware, firmware and software.

Online Addition, Replacement, and Deletion (OL*) of PCI cards is a system high availability feature. The OL* support requirements for interface drivers reflect additions to the WSIO interface and related driver modifications needed to handle OL* events while completing online addition, replacement and deletion of PCI cards. Pseudocode provides a better understanding of concepts.

In the examples and pseudocode presented in this chapter, the driver_name variable can be replaced with the actual driver name when you develop a driver that supports OL*.

To develop a driver that supports OL*, you must understand the following:

- HP-UX I/O subsystem
- WSIO driver development environment
- Writing a WSIO driver
- High availability
- The impact of an OL* operation on an application using the PCI HBA

This section addresses the following interface driver OL* support topics:

- “Driver Requirements for OL* Support” (page 455)
- “WSIO Interfaces” (page 456)
- “Registering an Event Handler” (page 458)
- “Registering an Event Mask” (page 458)
- “Writing the Event Handler” (page 460)
- “Configuring Event Timeout Values” (page 461)
- “Miscellaneous Required Driver Changes” (page 461)
- “Managing OL* Operations in the Driver” (page 462)
- “OL* Scripts” (page 469)

Driver Requirements for OL* Support

There are four major requirements for an interface driver to support OL* operations:

1. Register a generic event handler.
   
   To support OL* functionality, each driver must register an event handling function with WSIO. WSIO calls this event handler to service OL* events (such as suspend, resume, and remove) on the I/O card controlled by the driver. The event handling function is registered in the driver_install routine.

2. Register a capability mask.
   
   The driver must register an event mask with WSIO. The event mask specifies the driver’s OL* event handling capabilities. This enables a driver to support various capabilities on different instances of an I/O card. The event capability mask is registered in the driver_attach routine.
3. Write the event handler.
   To support OL* features, the driver must handle suspend, resume, and remove events that are generated by WSIO.

4. Configure timeouts.
   The OL* infrastructure expects drivers to respond to all events in a timely manner. The driver must manage timeouts. However, the driver can optionally register estimated timeout values for events.

Implementation details for these steps are presented in the following sections.

NOTE: OL* enhancements are not located in the driver's main performance path. They do not have a performance impact on the driver's regular data path.

WSIO Interfaces

WSIO provides a set of data structures and services that enable drivers to support OL*. They are defined in `<sys/wsio.h>`.

WSIO interacts with drivers that support OL* through events and event completion callbacks. WSIO delivers events, such as suspend or resume, by calling a driver event handler function. After the driver completes the event, it notifies WSIO by calling a completion function supplied by WSIO in the initial event.

WSIO Generic Event Structure

The driver's event handler is defined as follows:

```c
typedef void (*wsio_drv_event_handler_t) (wsio_generic_event_t *);
```

The `wsio_generic_event_t` passed to the driver event handler function is defined as follows:

```c
typedef struct wsio_generic_event {
    wsio_event_t event;  // Type of event
    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 function
    void *arg; // Event-specific argument
} wsio_generic_event_t;
```

Where:

- `event` The OL* event type. The event types are defined in `<sys/wsio.h>` as follows:

```c
typedef enum { // Event types
    ...
    WSIO_NO_EVENT = 0,
    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;
```

OL* uses only `WSIO_EVENT_SUSPEND`, `WSIO_EVENT_RESUME`, and `WSIO_EVENT_REMOVE` events.

- `event_id` The event identifier. The `event_id` is a tag that identifies an event to a driver's event handler. A driver returns the `event_id` when
it makes a completion callback. WSIO uses the *event_id* to match events with their completion callbacks.

*isc*  
A pointer to the interface driver's *isc* structure.

*wsio_completion_cb*  
The WSIO callback function that the driver must call to report status after it completes an event.

*arg*  
A void * parameter that contains information specific to the event type.

For a WSIO_EVENT_REMOVE event, *arg* contains the slot power status. During a PCI Error Recovery suspend event, WSIO sets a flag to distinguish that the WSIO_EVENT_SUSPEND event is part of a PCI Error Recovery operation. The following flags are passed in the *arg* parameter:

- **WSIO_SLOT_POWER_ON**  
The interface remains powered ON when the WSIO_EVENT_REMOVE event is sent.

- **WSIO_SLOT_POWER_OFF**  
The interface remains powered OFF when the WSIO_EVENT_REMOVE event is sent.

- **WSIO_BUS_ERROR_SUSPEND**  
The WSIO_EVENT_SUSPEND event is part of a PCI Error Recovery suspend operation.

During a WSIO_EVENT_REMOVE event, drivers must access the *arg* field using (wsio_remove_info_t)(intptr_t)arg to obtain the power state.

During a PCI Error Recovery suspend event, drivers must access the *arg* field using (wsio_suspend_info_t)(intptr_t)arg.

### WSIO Callback Function

After the driver completes the event, it invokes the WSIO callback function to report status. WSIO provides a pointer to the callback function in the *wsio_completion_cb* field of the *wsio_generic_event_t* structure. The calling interface for the callback function is as follows:

```c
int (*wsio_completion_cb)(struct isc_table_type *isc,  
wsio_event_id_t event_id, void *status );
```

Where:

- **isc**  
A pointer to the interface driver's *isc* structure.

- **event_id**  
The event identifier passed to the driver handler.

- **status**  
The status value. If the event handler successfully completes the event, it must set *status* to WSIO_OK. If the event handler does not complete the event, it must set *status* to one of the following WSIO status values:

```c
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_COMBINE_EVENTS = -8,
};
```
Registering an Event Handler

The driver registers an event handler function by calling \texttt{wsio\_install\_drv\_event\_handler}. This must be called after installing the driver by calling \texttt{wsio\_install\_driver} in the \textit{driver\_install} routine. The calling convention for \texttt{wsio\_install\_drv\_event\_handler} is as follows:

\begin{verbatim}
int wsio_install_drv_event_handler(
    wsio_drv_info_t *info_ptr,
    wsio_drv_event_handler_t handler
);
\end{verbatim}

Where:
\begin{itemize}
  \item \texttt{info\_ptr} A pointer to the interface driver's \texttt{wsio\_drv\_info\_t} structure. This structure is described in "\textit{drv\_ops\_t} Structure" (page 79)
  \item \texttt{handler} A function pointer of the interface driver's event handler.
\end{itemize}

This service is called in the \textit{driver\_install} routine. Pseudocode for a typical \textit{driver\_install} routine is as follows:

\begin{verbatim}
void driver\_name\_install(void)
{
    if (wsio_install_driver(&driver\_name\_drv\_info)) {
        saved_attach = pci_attach;
        pci_attach = driver\_name\_attach;
    } else { // Install failure
        return;
    }

    // driver\_name\_event\_handler() is the generic event
    // handler implemented in the driver.
    if (wsio_install_drv_event_handler(
        &driver\_name\_drv\_info,
        driver\_name\_event\_handler) != WSIO\_OK) {
        // Driver has not registered its event handler.
        // Driver's normal operation may not be affected.
    }
    return;
}
\end{verbatim}

Registering an Event Mask

The driver notifies WSIO about which events the driver can handle by calling \texttt{wsio\_reg\_drv\_capability\_mask}. This must be called from the \textit{driver\_attach} routine.
after calling `isc_claim`. The calling convention for `wsio_reg_drv_capability_mask` is as follows:

```c
int wsio_reg_drv_capability_mask(
    struct isc_table_type *isc,
    wsio_event_mask_t event_mask
);
```

Where:
- `isc` A pointer to the interface driver's `isc` structure.
- `event_mask` A 64–bit capabilities mask, formed by doing a logical OR operation of one or more `wsio_event_t` values. It represents 64 possible operations associated with the `isc`. The applicable flags for OL* are:
  - `WSIO_EVENT_SUSPEND` Indicates that the driver event handler can handle suspend events (suspend a PCI slot).
  - `WSIO_EVENT_RESUME` Indicates that the driver event handler can handle resume events (replace a PCI card).
  - `WSIO_EVENT_REMOVE` Indicates that the driver event handler can handle remove events (delete a PCI card).

Table 20-1 shows which events the driver must handle for each type of OL* operation.

### Table 20-1 OL* Event Capability Requirements

<table>
<thead>
<tr>
<th>OL* Operation</th>
<th>Event Capability Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Online Addition</td>
<td>There is no event for online addition. Online addition does not use the driver event handler. It uses the normal <code>driver_attach</code> and <code>driver_init</code> routines.</td>
</tr>
<tr>
<td>Online Replacement</td>
<td>To support online replacement, the driver must handle two events, <code>WSIO_EVENT_SUSPEND</code> and <code>WSIO_EVENT_RESUME</code>.</td>
</tr>
<tr>
<td>Online Deletion</td>
<td>To support online deletion, the driver must handle three events, <code>WSIO_EVENT_SUSPEND</code>, <code>WSIO_EVENT_RESUME</code>, and <code>WSIO_EVENT_REMOVE</code>.</td>
</tr>
</tbody>
</table>

Online deletion is implemented as a `WSIO_EVENT_SUSPEND` event followed by a `WSIO_EVENT_REMOVE` event. Support for online replacement operations is a prerequisite to support for online deletion operations, because the `WSIO_EVENT_SUSPEND` capability is a requirement for online replacement and online deletion operations. It is not valid to register the `WSIO_EVENT_REMOVE` flag without registering the `WSIO_EVENT_SUSPEND` and `WSIO_EVENT_RESUME` flags in the event mask.

The following pseudocode is typical for a driver `driver_attach` routine.

```c

driver_name_attach()
{
    ...
    wsio_event_mask_t driver_name_event_mask;
    driver_name_event_mask =
        WSIO_EVENT_SUSPEND | WSIO_EVENT_RESUME | WSIO_EVENT_REMOVE;
    ...
    isc_claim(isc, &driver_name_wsio_drv_info);

    // Register the driver event capability mask.
    // NOTE: This must 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.
    }
}
```
Writing the Event Handler

WSIO generates an event to a driver when it receives an OL* request (from pdweb or olrad) on an I/O card the driver controls. WSIO calls the driver’s event handling function, which the driver registered with WSIO during installation.

A driver event handler can support the OL* WSIO_EVENT_SUSPEND, WSIO_EVENT_RESUME, and WSIO_EVENT_REMOVE events.

The following example illustrates a driver event handler registered with WSIO for WSIO_EVENT_SUSPEND, WSIO_EVENT_RESUME, and WSIO_EVENT_REMOVE events.

**IMPORTANT:**

- A driver must always return successful completion for a WSIO_EVENT_SUSPEND event.
- A WSIO_EVENT_REMOVE event is expected to succeed. If the WSIO_EVENT_REMOVE event fails, the corresponding interface node moves to the UNUSABLE state and no further OL* operations are allowed on the slot containing that node until the next reboot of the system.
- Mass storage interface drivers must send an offline event for all of associated target paths during a WSIO_EVENT_SUSPEND and WSIO_EVENT_REMOVE operation. This target path offline event is propagated to the dump infrastructure subsystem, which reconfigures a dump device to an alternate path (if available).

The following pseudocode is an example of 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 allow the
    // event handler to return to WSIO immediately.
    // Use a timeout value of 0 in those cases
    // where there is no need to delay
    // individual event handler execution.

    switch (handler_arg->event) {

        case WSIO_EVENT_SUSPEND:
            // Save WSIO event information and call timeout().
            timeout (drv_name_suspend, driver_isc, 0);
            break;

        case WSIO_EVENT_RESUME:
            // Check if the driver already received a SUSPEND
            // for this instance. If not, return WSIO_INVALID_EVENT.
            // Save event information and call timeout().
            timeout (drv_name_resume, driver_isc, 0);
            break;

        case WSIO_EVENT_REMOVE:
            // Save WSIO event information.
            // Release all resources.
            // Unmap all registers.
            // Release all memory.
            // Release all locks.

            break;

    }
}
```

460 OL* and PCI Error Recovery Support in Interface Drivers
default:
    // This is an unrecognized event type.
    // Invoke the callback immediately
    handler_arg->wsio_completion_cb(handler_arg->isc,
    handler_arg->event_id, WSIO_UNSUPPORTED_EVENT);
    break;
}
return;

Configuring Event Timeout Values

WSIO does not implement timeouts for WSIO_EVENT_SUSPEND, WSIO_EVENT_RESUME, and WSIO_EVENT_REMOVE events. Drivers must manage timeouts for event completion. As an option, drivers can register an estimated time span for each of these events, which is displayed as the estimated time for completion in the -v option of the olrad command. If a driver does not reply within the registered time span, no WSIO timeout occurs. Instead, WSIO waits until the driver replies to the event. For this reason, drivers must always reply to WSIO, preferably within the registered time estimate, either after completing an event or through a driver timeout if the event is not completed within the registered time span. If a driver does not register the estimated time span for an operation, a default value of 10 seconds is displayed for the estimated time in the -v option of the olrad command.

Drivers set the estimated time for WSIO_EVENT_SUSPEND, WSIO_EVENT_RESUME, and WSIO_EVENT_REMOVE events as follows:

```
wsio_set_parm(
    struct isc_table_type *isc,
    wsio_parm_t parm,
    void *value
);
```

Where:
- `isc` A pointer to the interface driver's isc structure.
- `value` The maximum estimated time in microseconds for the event.
- `parm` One of the following values:
  - WSIO_HW_SUSPEND_TIMEOUT
  - WSIO_HW_RESUME_TIMEOUT
  - WSIO_HW_REMOVE_TIMEOUT

If a driver does not register the estimated time span for an operation, -1 is displayed for the estimated time.

Miscellaneous Required Driver Changes

There are two additional driver requirements for OL* support:

- To support online addition, the driver's driver_attach routine must enable bus mastering.
- While processing control path requests like device reset or device configuration, the driver must check to see if it is currently suspended, resumed, or in the process of suspending, resuming, or deleting. A driver state flag can be included in the driver control structure to keep track of the driver's OL* state.

If the driver is suspended, resumed, or in the process of suspending, resuming, or deleting, the request must be rejected with ENXIO. Pseudocode for a driver reset follows:

```
driver_name_reset(reset_info)
{
    ...
    // Additional state check
    if(driver_olar_state == SUSPENDED) {
```
Managing OL* Operations in the Driver

This section describes how Online Addition (OLA), Online Replacement (OLR), and Online Deletion (OLD) operations are managed when you write a driver that supports OL*.

HP-UX provides the following methods to perform OL* operations on PCI I/O cards:

- `pdweb` (GUI)
- `/usr/bin/olrad` (command-line interface)
- Attention button (used for OLA and OLR only; not available on all server platforms)

NOTE: PCI OLD cannot be performed using the Attention button on any HP server platform.

Use these methods to issue OL* requests on PCI I/O cards that are controlled by a third-party driver. For detailed information on the user interfaces available to perform OL* operations, see the Interface Card OL* Support Guide.

The remainder of this section explains the sequence of operations that must be performed by a driver that supports OL*.

Online Addition

To perform an online addition (OLA) of an I/O card, follow these steps:

1. Power off the slot using either the preadd operation of the `olrad` command (`olrad -a slot_id`) or the Add Card Online procedure in `pdweb`.
2. Insert the card into the slot.
3. Run the postadd operation of the `olrad` command (`olrad -A slot_id`) or the Add Card Online procedure in the `pdweb` command. The slot powers on and scans the new hardware.
4. Run `ioscan`, optionally passing the hardware path of the I/O card.

Online addition of an I/O card does not generate any WSIO events for the driver event handler to process. Instead, the driver's `driver_attach` and `driver_init` routines are called as they are during boot. To support OLA, drivers must do the following:

- Make the driver's `driver_attach` and `driver_init` routines MP safe.
- Handle resource allocation failures during `driver_attach` and `driver_init`.

MP Safe

The `driver_attach` and `driver_init` routines must be Multiprocessing (MP) Safe, because an OLA can occur at any time when the machine is running. If the driver has any global resources common to all instances, you must allocate them during driver installation. Required global locks can be acquired and released during `driver_attach` and `driver_init` routines to serialize access and to protect data.

Resource Allocation Failures

Because an OLA can occur at any time when the machine is running, resource allocation can fail because of a resource shortage. The driver must retrace all the steps and release all the resources allocated before the failure occurred.

To facilitate backing out an OLA due to a failure during `driver_attach` and `driver_init`, Table 20-2 lists the resource allocation services and their corresponding deallocation services.
Resources must be deallocated in last in, first out (LIFO) order. That is, the most recently allocated resources are released first.

**Table 20-2 Resource Allocation and Deallocation Functions**

<table>
<thead>
<tr>
<th>Resource Allocation</th>
<th>Resource Deallocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_nmidx</td>
<td>return_nmidx</td>
</tr>
<tr>
<td>isc_claim</td>
<td>WSIO cleans up when it receives an error from driver_init.</td>
</tr>
<tr>
<td>kmalloc</td>
<td>kfree</td>
</tr>
<tr>
<td>wsio_allocate_shared_mem</td>
<td>wsio_free_shared_mem</td>
</tr>
<tr>
<td>wsio_allocate_dma_handle</td>
<td>wsio_free_dma_handle</td>
</tr>
<tr>
<td>wsio_intr_activate</td>
<td>wsio_intr_deactivate</td>
</tr>
<tr>
<td>wsio_intr_alloc</td>
<td>wsio_intr_free</td>
</tr>
<tr>
<td>wsio_map_dma_buffer</td>
<td>wsio_unmap_dma_buffer</td>
</tr>
<tr>
<td>wsio_map_port</td>
<td>wsio_unmap_port</td>
</tr>
</tbody>
</table>

All drivers must provide a return value from each driver_init function. The value must be one of the following:

- **WSIO_OK (0)**  Success.
- **WSIO_ERROR (-1)**  Failure.

**Online Replacement**

To perform an online replacement (OLR) of an I/O card, follow these steps:

1. Suspend the driver instance associated with the I/O card using pdweb or olrad. A critical resource analysis (CRA) is performed as part of the suspend operation. If a system critical usage is detected, you cannot proceed with the online replacement operation. If a data critical usage is detected, you can manually force the OLR using the olrad command or pdweb to suspend the driver instance. After the driver instance is suspended, the slot powers off.
2. Remove the card.
3. Insert the replacement I/O card.
4. Resume the driver instance associated with the I/O card using pdweb or olrad. After the driver instance is resumed, the slot powers on.

A driver that supports OLR handles the events that happen during steps 1 through 4.

WSIO guarantees OL* requests are single threaded. The underlying OL* infrastructure guarantees that no OL* events are triggered on the same card instance until the previous event processing completes. For OLR, the underlying infrastructure does not trigger a resume event if the replacement card is incompatible. The driver is expected to do a detailed check, such as comparing the PCI subsystem ID, before claiming the replacement card.

**Suspend**

To suspend the driver instance, WSIO sends a WSIO_EVENT_SUSPEND event to the driver. For multiport cards and combo cards, the WSIO_EVENT_SUSPEND event is sent serially to each instance of the driver. A suspend request never occurs while a driver is suspending or suspended. If this happens, the driver must immediately call the callback function with WSIO_INVALID_EVENT status.

While a driver is suspending or after it is suspended, some control requests, such as reset or abort (or any other request that normally causes the driver to change state or interact directly with the adapter) must be rejected. In these cases, the control request path must check for the...
Suspended state (or equivalent) and any other state the driver can be in while it is suspending. Specific handling of a control request during and after a suspend operation is driver-dependent. The driver must take the following actions on receipt of a WSIO_EVENT_SUSPEND event:

1. **Wait for the correct state.** The definition of correct state depends on the driver. No critical operation can be in progress, such as device reset, blocked on I/O request, and so on. Pseudocode to suspend a driver is as follows:

```c
void driver_name_event_handler(wsio_generic_event_t *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 the suspend routine.
                timeout(driver_name_suspend_handler,
                        suspend_info, PREDEFINED_WAITING_TIME);
            } else {
                // Call the suspend handler. No need to wait here.
                timeout(driver_name_suspend_handler,
                        suspend_info, 0);
            }
            break;
        ...
    }
    ...
}
```

If the driver cannot set a predetermined time, the driver’s suspend handler times out after a set number of clock ticks unless the driver comes out of its critical operation state before the timeout occurs. Pseudocode to perform this is as 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 the suspend routine.
        timeout(driver_name_suspend_handler,
                suspend_info, HZ);
        return;
    }
}
```

2. **If the driver has any timers, other than timers specifically used to time the suspension sequence, cancel them using `untimeout`.**

3. **Quiesce the device (driver-dependent).** After quiescing, the device is not expected to perform any I/O operations or generate interrupts to the driver.

4. **Save the required device information for use by a resume operation.** For example, when only like-for-like replacement is allowed during resume, comparing the vendor ID, subsystem ID, and so on, can be used to identify a suitable replacement I/O card.

5. **Save any other required information (driver-dependent).** This information usually includes the device state or configuration that the driver must restore to the replacement I/O card.

6. **Call the callback function with WSIO_OK status.**
Resume

To resume the driver instance, WSIO sends a WSIO_EVENT_RESUME event to the driver. A resume request can occur only when the driver is suspended, and is not resuming. If the resume event occurs at any other time, the driver must immediately call the callback function with WSIO_INVALID_EVENT status.

If resume fails, the driver must return to the Suspended state, and call the callback function with WSIO_ERROR status. While a driver is resuming, some control requests, such as reset or abort (or any other request that normally causes the driver to change state or interact directly with the adapter) must be rejected. Therefore, the control request path must check for the Resuming state (or equivalent) and any state the driver can be in while it is resuming. Specific handling of a control request during and after a resume operation is driver-dependent.

The driver must take the following actions on receipt of a WSIO_EVENT_RESUME event:

1. Check for a like-for-like replacement. The OL* infrastructure 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 call the callback function with WSIO_ERROR status.
2. Initialize the device. Specific initialization operations depend 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.

Pseudocode for a driver resume routine is as follows.

driver_name_event_handler(handler_arg)
{
    ...
    switch(handler_arg->event) {
        ...
        case WSIO_EVENT_RESUME:
            // Check if the driver is in suspended state.
            // Test for 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 the host, if applicable.
    // Initialize the device.
    // Configure the device.
    // Call the callback function with WSIO_OK status.
    return;
}
Online Deletion

To perform an online deletion (OLD) of an I/O card, use the following two-phase process:

Predelete Phase

Run the OLD predelete phase by entering the predelete option of the `olrad` command, `olrad -d slot_id`, or by using the `pdweb` GUI. The steps in the online deletion predelete phase are as follows:

1. CRA with **LOCK** and **ANALYSIS** is performed.
   a. If CRA returns any resource usage, issue unlock to CRA. The delete operation fails with an error message.
   b. If CRA returns success (or the usage found can be overridden), proceed to step 2.

2. Delete the driver instance associated with the I/O card.
   a. All the driver instances associated with the I/O card receive a `WSIO_EVENT_SUSPEND` event.
   b. The slot is powered off.
   c. All the driver instances associated with the I/O card receive a `WSIO_EVENT_REMOVE` event.

This completes the predelete phase. At this point, the slot power is OFF and the slot attention LED is blinking. Now you can physically remove the card from the slot and initiate the postdelete phase, which turns off the blinking attention LED.

Postdelete Phase

Run the OLD postdelete phase by issuing the `olrad` command postdelete option, `olrad -D slot_id`, or by using the `pdweb` GUI.

NOTE: WSIO guarantees that suspend, resume, and remove requests are single threaded. The underlying OL* infrastructure guarantees no suspend, resume, or remove events are triggered on the same card instance until the previous event processing is complete.

PCI Card Online Deletion Flow

The sequence of steps for OLD is described in this section. See “Predelete Phase” (page 466) and “Postdelete Phase” for information on the predelete and postdelete phases, respectively.

OLD follows these steps:

1. Performs a CRA.
   Critical Resource Analysis (CRA) ensures that the components being deleted do not have any critical usage in the system. See Chapter 21 (page 477). No events are generated during the CRA phase.

2. Suspends driver instances on the card.
   To suspend the driver instance, WSIO sends a `WSIO_EVENT_SUSPEND` event to all instances of the driver on the card. This is the same event that is sent during PCI card OLR.

   A suspend request never occurs while a driver is suspending or suspended. If this happens, the driver must immediately call the callback function with `WSIO_INVALID_EVENT` status. While a driver is suspending or after it is suspended, some control requests, such as reset or abort (or any other request that normally causes the driver to change state or interact directly with the adapter) must be rejected. In these cases, the control request path must check for the suspended state (or equivalent), or any other state the driver can be in while it is suspending. Specific actions a driver takes while handling a control request during and after a suspend operation is driver-dependent.

   The driver must take the following actions on receipt of a `WSIO_EVENT_SUSPEND` event:
a. Wait for the correct state. The correct state depends on the driver. No critical operation can be in progress, such as device reset and blocked on I/O request. Pseudocode to suspend a driver follows:

```c
void driver_name_event_handler(wsio_generic_event_t *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 the suspend routine.
                timeout(driver_name_suspend_handler,
                        suspend_info, PREDEFINED_WAITING_TIME);
            } else {
                // Call the suspend handler. No need to wait here.
                timeout(driver_name_suspend_handler,
                        suspend_info, 0);
            }
        break;
        ...
    }
}
```

If the driver cannot set a predetermined time, the driver’s suspend handler times out after a set number of clock ticks unless the driver comes out of its critical operation state before the timeout occurs. Pseudocode to perform this is as 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 the suspend routine.
        timeout(driver_name_suspend_handler,
                suspend_info, HZ);
        return;
    }
}
```

b. If the driver has any timers, other than timers specifically used to time the suspension sequence, cancel them using `untimeout`.

c. Quiesce the device (driver-dependent). After quiescing, the device is not expected to perform any I/O operations or generate interrupts to the driver.

d. Call the callback function with `WSIO_OK` status.

3. Powers off the slot.

   The OL* infrastructure in the kernel powers off PCI slots. Interface drivers do not receive an event when powering off.

4. Removes the card.

   After the slot of powered off, WSIO sends a `WSIO_EVENT_REMOVE` event to the driver. For multiport cards and combo cards, the `WSIO_EVENT_REMOVE` event is sent serially to each instance of the driver. The current power state of the slot is passed to the driver in the `arg` field of the `wsio_generic_event_t` structure, as described in “WSIO Generic Event Structure” (page 456). A `WSIO_EVENT_REMOVE` event only occurs when the driver is
suspended. If the event occurs at any other time, the driver must immediately call the callback function with WSIO_INVALID_EVENT status.

For a WSIO_EVENT_REMOVE event, the driver event handler must do the following:

a. Free all DMA resources used by the driver instance.
b. Free all allocated memory for the driver instance.
c. Free all other driver instance-specific resources, such as spinlocks.
d. Perform any other necessary driver instance-specific cleanup operations.
e. Call the callback function with WSIO_OK status.

Drivers must also handle a remove request when a driver is suspended after a failed automatic error recovery attempt by the PCI Error Recovery feature.

If the WSIO_EVENT_REMOVE event fails, the driver must call the callback function with WSIO_ERROR status. Specific actions a driver takes while handling a control request during and after OLD are driver-dependent.

**NOTE:** Resources must be released (deallocated) in the reverse order from which they were acquired (allocated).

The following example contains pseudocode to remove a driver:

```c
void
driver_name_event_handler(wsio_generic_event_t *handler_arg)
{
    ...
    switch (event_arg->event) {
        ...
        case WSIO_EVENT_REMOVE:
            // Save WSIO event information.
            // Release all resources.
            // Unmap all registers.
            // Release all memory.
            // Release all locks.
            break;
        ...
    }
    ...
}
```

5. Turns off the attention LED at the slot.

After the predelete phase succeeds, the slot power is off and the attention LED is blinking. After the PCI card has been removed from the slot, initiate the postdelete phase by issuing the `olrad -D` command, which turns off the attention LED. No events are sent to the driver during the postdelete phase.

**NOTE:** WSIO does not implement timeouts for WSIO_EVENT_SUSPEND, WSIO_EVENT_RESUME, and WSIO_EVENT_REMOVE events. Drivers must always reply to WSIO, preferably within the registered time estimate, either after completing an event or through a driver timeout if the event is not completed within the registered time span. For more information, see “Configuring Event Timeout Values” (page 461).

### PCI Card Online Deletion Failure Scenarios

PCI card online deletion operations can fail for the following reasons:

- **Power Off Fault During OLD**
  
  A power-off fault at a slot is critical. If the power off at a slot fails, the OLD operation is aborted and the slot node is put into the UNUSABLE state. No further operations are allowed on the slot until the next reboot.
Although rare, drivers can fail the remove event. If a driver fails the remove event, the OLD operation is aborted and the interface node that failed is put into the UNUSABLE state. No further operations are allowed on the slot until the next reboot.

OL* Scripts

The olrad command invokes a driver script for OL* operations before and after a PCI OL* operation. There is one script per driver. The script takes different flags and arguments depending on the OL* operation being performed. The driver script for OL* operations resides in the /usr/sbin/olrad.d directory and is identified by the driver name. For example, if the ioscan command indicates that the driver associated with an interface is driver1, a script named /usr/sbin/olrad.d/driver1 is called with the appropriate flags.

Generically, driver scripts are executed as follows:

```
# sh /usr/sbin/olrad.d/driver_name EXECUTE_ACTION Input_Parameters
```

The valid values for EXECUTE_ACTION are as follows:

- post_add: Execute post add actions.
- pref_replace: Execute preface to replace actions.
- prep_replace: Execute prepare to replace actions.
- post_replace: Execute post replace actions.
- prep_delete: Execute prepare to delete actions.
- post_delete: Execute post delete actions.

The Input_Parameters for the post_add, pref_replace, prep_replace, and post_replace actions are the hardware path of the interface card. For example:

```
# sh /usr/sbin/olrad.d/driver_name post_add HardwarePath
```

Possible return codes are as follows:

- 0: Success
- 1: Warning
- 2: Error

The Input_Parameters for the prep_delete and post_delete values are the instance number and the class name. For example:

```
# sh /usr/sbin/olrad.d/driver_name prep_delete InstanceNum ClassName
```

Possible return codes are as follows:

- 0: Success
- 1: Warning
- 2: Error

PCI Error Recovery Support Requirements for Interface Drivers

PCI Error Recovery is a system high availability feature that detects, isolates, and automatically recovers a faulty PCI card without impacting other system components. The PCI Error Recovery feature is only available on HP-UX 11i v3. The PCI Error Recovery support requirements for HP-UX 11i v3 interface drivers include additions to the WSIO interface and related driver modifications. Pseudocode provides a better understanding of concepts.

In the examples and pseudocode presented in this section, replace the driver_name variable with the actual driver name when you develop a driver that supports PCI Error Recovery.
To develop a driver that supports PCI Error Recovery, you must understand the following:

- HP-UX I/O subsystem
- WSIO driver development environment
- Writing a WSIO driver
- High availability
- The driver's application impact, if any, on a PCI OL* operation on a PCI HBA

This section addresses the following interface driver error recovery support requirements:

- “PCI Error Recovery Capability” (page 470)
- “Registering Event Handler and PCI Error Recovery Capability” (page 471)
- “Detecting a PCI Error Condition” (page 472)
- “Reporting a PCI Bus Error” (page 473)
- “Automatic PCI Error Recovery Stages” (page 474)
- “Manual PCI Error Recovery” (page 475)

**Overview**

If an interface driver is not developed to support the PCI Error Recovery feature, a PCI fatal error or a rope error on the bus causes an High Priority Machine Check (HPMC) and Machine Check Abort (MCA), and the system crashes. The PCI Error Recovery feature automatically detects, isolates, and recovers from a PCI bus or rope error without user intervention. PCI Error Recovery requires support from system software components. If PCI Error Recovery is supported on an interface driver, a PCI fatal error or rope error does not cause a HPMC/MCA on a Parallel I/O Controller (PIO) read. Instead, the PCI bus on which the error occurs is quarantined to isolate the system from further I/O and prevent the error from damaging other system components. PCI Error Recovery attempts to automatically recover from the error and reinitialize the bus so I/O can resume. If an error occurs during the automated error recovery process, the bus and PCI card remain quiesced. If the bus contains a card that supports online addition and replacement, and the card is in a hot-pluggable slot, you can use the olrad command (or the attention button) to manually recover from the error by replacing the card. For information on OL* operations, see the Interface Card OL* Support Guide.

**NOTE:** If an interface driver can perform both Online Suspend and Online Resume operations, full support of the automated PCI Error Recovery feature can be implemented with minimal changes in the driver code. For more information, see “OL* Support Requirements for Interface Drivers” (page 455).

**PCI Error Recovery Capability**

When a hardware path corresponding to an interface card goes into Fatal Error mode, the system reacts differently depending on whether the path is in PCI Error Recovery mode. If the path is not in PCI Error Recovery mode, a PIO read results in an HPMC/MCA when a fatal error occurs. If the hardware path is in PCI Error Recovery mode, the system does not HPMC/MCA on a PIO read. Instead, the hardware path is firewalled preventing interrupts or DMA from devices underneath. In addition, all PIO reads return -1.

To support PCI Error Recovery, drivers must be able to detect when a -1 return on a register read is not a valid return value, which indicates that the I/O path is in Fatal Error mode. Drivers must register this capability with WSIO CDIO for each instance of an I/O card that is claimed. In the case of multiport/multifunction cards, each instance must register its PCI Error Recovery capability. All instances of I/O cards on a PCI bus must be in PCI Error Recovery mode for the automated PCI Error Recovery feature to be supported on that bus. After a driver instance registers its PCI Error Recovery capability, whether it operates in PCI Error Recovery mode or not is dependent on the capability of the corresponding platform and system firmware and other
driver instances on the same PCI bus. If the platform/system firmware or other driver instances on the same PCI bus do not support PCI Error Recovery, a PCI error results in an HPMC/MCA.

Registering Event Handler and PCI Error Recovery Capability

Interface drivers must register an event handler with WSIO CDIO to be invoked by WSIO when specific events occur. This registration must be completed in a driver's install routine. In addition, interface drivers must register an event capabilities mask for each instance they claim. The event capabilities mask defines a driver's capabilities and indicates what events the driver handler responds to.

To register an event handler a driver calls the WSIO service `wsio_install_drv_event_handler` in the driver's install routine after it calls `wsio_install_driver` to register with WSIO CDIO. The calling interface for this service is as follows:

```c
int wsio_install_drv_event_handler(
    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. The second parameter is a pointer to the driver's event handler.

The driver's event handler must have the following calling interface:

```c
void (*drv_handler) (wsio_generic_event_t *);
```

When called, the event handler is passed a single parameter that is a pointer to a `wsio_generic_event_t` structure type. This structure is defined in the header file `<sys/wsio.h>` as follows:

```c
typedef struct wsio_generic_event {
    wsio_event_t event;
    wsio_event_id_t event_id;
    struct isc_table_type *isc;
    generic_complete_callback_t wsio_completion_cb;
    void *arg;
} wsio_generic_event_t;
```

Where:

- `event` Specifies the type of event the driver event handler is called for. For PCI Error Recovery events, the `event` field has the following values:
  - `WSIO_EVENT_SUSPEND`
  - `WSIO_EVENT_RESUME`

- `event_id` Identifies a specific event instance.

- `isc` A pointer to the interface driver's `isc` structure.

- `arg` Used to distinguish between a Suspend operation and a PCI Error Suspend operation. Set to `WSIO_BUS_ERROR_SUSPEND`. Drivers must access this field as type `wsio_suspend_info_t` to read the `WSIO_BUS_ERROR_SUSPEND` flag.

For example:

```c
(wsio_suspend_info_t) (intptr_t) WSIO_BUS_ERROR_SUSPEND
```

A driver must call the `wsio_completion_cb` callback function to report status after it completes an event. The calling interface for the `wsio_completion_cb` callback function is as follows:

```c
int (* wsio_completion_cb)( struct isc_table_type *isc,
    wsio_event_id_t event_id, void * status );
```

Where:

- `isc` A pointer to the interface driver's `isc` structure.

- `event_id` The value of the `event_id` field passed into the driver event handler.
The status. If the event handler successfully competes an event, it must set the
status to WSIO_OK. The driver must always succeed in suspending the driver
instance.

When a driver registers itself as PCI Error Recovery capable, for each instance that is claimed,
the driver must register an event capabilities mask indicating that it can handle bus errors and
indicating which events the event handler responds to.

A driver registers its event capabilities mask by calling the
wsio_reg_drv_capabilities_mask WSIO service. The calling interface for this service is
as follows:

```c
int wsio_reg_drv_capabilities_mask(
    struct isc_table_type *isc,
    wsio_event_mask_t mask
);
```

Where:

- `isc` A pointer to the interface driver's isc structure.
- `mask` The capabilities mask, which is formed by doing a logical OR operation on one or more
  wsio_event_t values.

When a driver registers itself as supporting PCI Error Recovery and able to handle bus errors,
then for each instance that is claimed by that driver, it must register an event capabilities mask
that has the following events:

- WSIO_EVENT_SUSPEND
- WSIO_EVENT_RESUME
- WSIO_EVENT_BUS_ERROR

For example:

```c
wsio_event_mask_t V_mask = WSIO_EVENT_SUSPEND |
    WSIO_EVENT_RESUME | WSIO_EVENT_BUS_ERROR;
wsio_status = wsio_reg_drv_capabilities_mask(isc, V_mask);
```

Drivers must call `wsio_reg_drv_capabilities_mask` and register the event capabilities
mask in the driver's attach routine after calling `isc_claim`.

Drivers must not hold any spinlocks while calling `isc_claim`.

Drivers must register the SUSPEND, RESUME and BUS_ERROR capabilities together in one
invocation, as shown in the previous example.

---

**NOTE:** PCI Error Recovery ensures that a defective PCI card does not cause a system to
HPMC/MCA during the initial bus scan and claim. This is achieved by keeping each PCI slot in
PCI Error Recovery mode during scan, until the system is notified that a driver claiming a card
or port in any PCI slot is not PCI Error Recovery capable. All HP-UX 11i v3 interface driver
attach routines run in PCI Error Recovery mode. Therefore, interface drivers that do not support
PCI Error Recovery must also ensure that their attach function can handle a -1 return value
for all register reads done during the attach routine. A driver must not interpret a -1 return as a
valid value for a register read done in the attach routine and then continue with the claiming
process. Instead, an interface driver that does not support PCI Error Recovery must fail the attach
if a -1 returns. For more information on attach routine requirements for interface drivers that
do not support PCI Error Recovery, see “Detecting a PCI Error Condition” (page 472) and Chapter 4
(page 77).

---

**Detecting a PCI Error Condition**

To support PCI Error Recovery, drivers must periodically check the health of the PCI bus. Most
drivers have watchdog timers that gather card and link status information. The watchdog timer
can detect a PCI Error condition with other card and link status information. Typically card and
link status is re-evaluated once every second in a watchdog timer function, either by doing a
PIO read to a register or by doing a configuration read as follows:

- If the watchdog timer is doing a PIO read and a -1 value is found in a register that must
  never have -1 as a value, an error has occurred.
- If the watchdog timer is doing a configuration read and the return value is -1, an error has
  occurred.

If the configuration read returns any value other than -1, the command register must be read
and the I/O space enable bit and memory space enable bit must be checked. If neither the I/O
space nor the memory space bits are set, this indicates that the firmware reset the bus because
of an error, and the driver must assume that an error occurred. However, since a PCI Error can
occur between the time that the configuration read is done and the time that the I/O and memory
space enable bits are checked, the command register must be checked first to see if it has a value
of -1. The order of events is as follows:

1. Check if the configuration read is -1. If yes, an error has occurred.
2. If the configuration read is not -1, check if the command register is -1. If yes, an error has
   occurred.
3. If the command register is not -1, check if the I/O enable bit, the memory enable bit, the SERR
   bit, and the PERR bit in the command register are reset. If all four of these bits are reset, an
   error has occurred.

Alternatively, when doing a configuration read, drivers can use the
\texttt{wsio\_check\_bus\_error} WSIO interface to confirm the PCI Error condition, where the
isc parameter is the isc structure of the driver instance. The \texttt{wsio\_check\_bus\_error}
function returns 1 if the bus is in error. Otherwise, it returns 0.

For every configuration register read done in the code, the driver can either implement the
previous three steps or call the \texttt{wsio\_check\_bus\_error} WSIO interface to confirm that
a PCI Error has occurred.

After it is determined that a PCI Error occurred, a driver must report the error to WSIO CDIO
using the \texttt{wsio\_device\_event} interface as described in “Reporting a PCI Bus Error” (page 473).

### Reporting a PCI Bus Error

To report an event, a driver calls the \texttt{wsio\_device\_event} service in WSIO. The calling interface
for this service is as follows:

\begin{verbatim}
int wsio_device_event(
    struct isc_table_type *isc,
    wsio_event_type_t event_type,
    wsio_event_info_t event_info
);
\end{verbatim}

Where:

- *isc* A pointer to the interface driver's isc structure.
- *event\_type* The type of event the driver wants to report. One possible value is
  \texttt{WSIO\_BUS\_ERROR\_DETECTED}.
- *event\_info* Do not use. Set this to 0.

When a driver determines that a PCI Error condition has occurred, it must report to WSIO CDIO
by calling \texttt{wsio\_device\_event} and passing \texttt{WSIO\_BUS\_ERROR\_DETECTED} as the second
parameter. For example:

\begin{verbatim}
wsio_status = wsio_device_event (ift->isc, WSIO_BUS_ERROR_DETECTED, 0);
\end{verbatim}

A driver must not hold any spinlocks while calling the \texttt{wsio\_device\_event} interface. After it
is determined that a PCI Error has occurred, the driver must always report the error to WSIO,
with the following exceptions:
• If a driver encounters a PCI Error while executing its attach routine, the driver must fail the attach routine. It is optional for a driver to report the error in this situation, but the PCI Error Recovery infrastructure ensures that nodes corresponding to the card are in the ERROR state.

• If a driver encounters a PCI Error during a resume operation, it must fail the resume operation and return the WSIO_ERROR status using the callback function. It is optional for a driver to report the error in this situation, but the driver must fail the resume and return the WSIO_ERROR status to WSIO.

• If a driver encounters a PCI Error during a suspend operation, the driver must succeed the SUSPEND because SUSPEND must not fail. It is optional for a driver to report an error in this situation.

NOTE: If a driver encounters a PCI error while executing its initialization routine, the driver must fail the initialization routine and the driver must report the error to WSIO using wsio_device_event.

NOTE: The wsio_device_event interface is a nonblocking interface, so drivers are also able to report an error in an interrupt context.

Automatic PCI Error Recovery Stages

After a driver has reported a bus error, the hardware path is firewalled, preventing interrupts or DMA from devices underneath. In addition, all register reads of I/O card registers return -1. The I/O subsystem goes through DIAGNOSE, SYNC, and RELEASE stages for automatic PCI Error Recovery. Table 20-3 summarizes the actions of each automatic PCI Error Recovery stage.

<table>
<thead>
<tr>
<th>Table 20-3 PCI Error Recovery Stages</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DIAGNOSE</strong></td>
<td>The driver is not included at this stage. This event goes only to the root node of the error (LBA in most cases) and is handled at the error recovery infrastructure level.</td>
</tr>
<tr>
<td><strong>SYNC</strong></td>
<td>The WSIO CDIO calls the driver's event handler with a WSIO_EVENT_SUSPEND event type. The arg field of wsio_generic_event_t is set to WSIO_BUS_ERROR_SUSPEND. Drivers must access this field as type wsio_suspend_info_t to read the WSIO_BUS_ERROR_SUSPEND flag. The driver must always succeed in suspending the driver instance. At this point, the card is not accessible. The driver must ignore any errors when reading or writing to card registers. If the driver is written to report errors found during the SYNC phase, it does. The automated error recovery infrastructure handles the situation. The driver must return the WSIO_OK status using the callback function.</td>
</tr>
<tr>
<td><strong>NOTE:</strong> Drivers have the option to support only manual recovery from PCI Errors by returning the WSIO_OK_MANUAL_RECOVERY status in the SYNC phase (see “Manual PCI Error Recovery” (page 475)).</td>
<td></td>
</tr>
<tr>
<td><strong>RELEASE</strong></td>
<td>WSIO CDIO calls the driver's handler with a WSIO_EVENT_RESUME event type. The driver resumes the instance.</td>
</tr>
<tr>
<td><strong>NOTE:</strong> This is the same resume mechanism that is used in an OL* resume operation. Prior to resume, it is guaranteed that a bus or rope reset is performed. But, there is no guarantee that the bus is still alive at the time of the resume as other instances on the same bus might have caused a fatal error. If the driver encounters a fatal error during resume, it must fail the resume and return the WSIO_ERROR status using the callback function. If the driver is written to report errors found during the RESUME phase, it does. The automated error recovery infrastructure handles the situation. Upon completion of the resume after the driver's callback function is executed, the error recovery infrastructure checks the condition of the rope or bus. If the bus or rope is still in the fatal error state, the bus is left in the fatal error state and all driver instances remain in the same state as at the end of the SYNC phase. These driver instances can be recovered using a manual recovery operation, which is described in “Manual PCI Error Recovery” (page 475).</td>
<td></td>
</tr>
</tbody>
</table>
Manual PCI Error Recovery

If automatic PCI Error Recovery fails (there is a failure in the RELEASE phase), you have two options:

- Manually replace the failed cards and initiate a manual recovery by running the `olrad` prereplace option (`olrad -r`) and the `olrad` postreplace option (`olrad -R`).
- Delete the card from the system by performing a PCI OLD (`olrad -d slot id`) followed by `olrad -D slot id`.

**NOTE:** Drivers can support only manual recovery from PCI errors. This option is implemented in drivers that use the RDMA mechanism and cannot support recovery from PCI Errors without user intervention. For example, Infiniband.

The driver requirements to support only manual PCI Error recovery are as follows:

- The driver instance must register PCI Error Recovery capability and event handlers as described in “Registering Event Handler and PCI Error Recovery Capability” (page 471).
- The driver instance must detect and report PCI errors as described in “Detecting a PCI Error Condition” (page 472) and “Reporting a PCI Bus Error” (page 473).
- When a driver instance receives a SUSPEND due to a PCI error, the driver must succeed the SUSPEND, but return the `WSIO_OK_MANUAL_RECOVERY` status (instead of the `WSIO_OK` status) through the callback function. See the SYNC phase in Table 20-3 (page 474).

Implementing these requirements in a driver results in the PCI Error Recovery infrastructure skipping the RELEASE stage of automatic error recovery. The node state of the driver instance in the I/O tree remains in the ERROR state. The user can then perform a manual recovery operation using `olrad -r` and `olrad -R` as described in “Manual PCI Error Recovery” (page 475).
21 Critical Resource Analysis

Critical Resources Analysis (CRA) provides usage analysis for I/O resources during PCI card OL* operations.

CRA Framework

Starting with HP-UX 11i v3, CRA provides an environment that enables individual subsystems to develop and deliver usage analysis plug-in modules. These plug-in modules perform usage analysis specific to their subsystems and report any potential impact on a set of I/O components. The complete CRA framework consists of multiple subsystem CRA modules and one infrastructure CRA module. The following sections describe each module.

Infrastructure CRA Module

The infrastructure CRA module provides basic functionality and coordination between the subsystem CRA modules. This module is delivered as a shared library, and performs the following tasks:

- During analysis, the infrastructure CRA module interacts with and queries each subsystem CRA module through a well-defined interface.
- Gathers usage return code from all the subsystem CRA modules and returns the highest critical usage reported.
- Drives the lock and unlock events during usage analysis and implements a graceful exit mechanism for certain error scenarios during CRA.
- Collates the usage analysis reports from all subsystem CRA modules and returns the combined logs in a single CRA report for the system.

Subsystem CRA Modules

Subsystem CRA plug-in modules analyze usage for specific subsystems and report impacts on I/O components that are associated with those subsystems. A subsystem is a group of components for which a subsystem CRA plug-in module covers the maximum number of usage scenarios using the same analysis techniques. These modules are developed and delivered as shared library plug-ins.

Module Interaction

A CRA framework has an infrastructure CRA, a networking subsystem CRA module, and a mass storage subsystem CRA module.

For example, a 4-port LAN card is taken offline (suspended or deleted). Before the card goes offline, the four LAN interface ports on the card must be analyzed for usage. For CRA purposes, an I/O resource is represented by its hardware path. The infrastructure CRA module invokes both the networking and mass storage subsystem CRA modules with the hardware paths for the four LAN ports on the I/O card. The subsystem CRA modules check whether any of the LAN ports are in use within the networking or mass storage subsystems. Each subsystem CRA module returns its analysis report to the infrastructure CRA module. The infrastructure CRA module returns a combined report to the caller.

CRA Interface

The infrastructure CRA library exports an interface that returns usage analysis details for a set of I/O components. The analysis details are returned in a log buffer. This interface is called from the olrad command. For example, the infrastructure CRA interface is called before an I/O card is deleted from the system. In addition, the infrastructure CRA module calls interfaces exported from each of the subsystem CRA libraries to query I/O component usage in each subsystem.
Infrastructure CRA Call Semantics

The infrastructure CRA module completes a resource analysis before executing an online replacement or deletion operation by calling the exported CRA interface. For example, the infrastructure CRA interface can be called from the `olrad` command in either of the following cases:

- Before a PCI card is online replaced to get a usage report for the card and associated components
- Before a PCI card is online deleted to get a usage report for the card and associated components

The infrastructure CRA interface supports the following call semantics:

- **Locked analysis**
  
  Applies to subsystem CRA modules that support locked analysis. Locked analysis ensures that no new usage is initiated, and that the usage reported by the CRA module remains unchanged until a `RELEASE` event is sent. During locked analysis, the CRA interface must be invoked twice. The first call gets usage analysis details for a set of I/O components. The second call releases (unlocks) any subsystem CRA modules that support locked analysis. The second call must be made independent of a successful or failed outcome for the first call. Both of these calls must be from the same process context, for example, from the same process instance of a command. CRA lock mechanisms are not persistent across different process contexts.

  Locked analysis is typically used before a destructive operation, for example, online deletion of an I/O card. The card is deleted only if no critical usage is discovered by the CRA. A `RELEASE` event is sent to the infrastructure CRA module after the card is deleted.

- **Resource analysis**
  
  Used by subsystems that do not support locked analysis. Resource analysis get a snapshot of resource usage for a set of components, independent of destructive operations. For example, resource analysis reports usage information for an I/O card through the `pdweb` tool. Because there is no locked analysis, a second call for `RELEASE` (unlock) is not required. The CRA interface is called only once to get the usage report.

For flow diagrams of `olrad` command interactions with the infrastructure CRA interface, see “CRA Flow Examples” (page 487).

Subsystem CRA Interface

The infrastructure CRA module calls each subsystem CRA interface to get the resource usage report for that subsystem. This interface must be exported by all subsystem CRA modules. When a locked analysis is done, the subsystem CRA interface is invoked twice from the infrastructure CRA module. The first call gets the usage analysis details for I/O components; the second call unlocks or releases the subsystem.

Each of the subsystem CRA shared libraries is dynamically loaded from the infrastructure CRA library. If the subsystem CRA libraries have dependencies on other libraries, these dependencies must be resolved while they are dynamically loaded.

Each of the subsystem CRA libraries must export the following interface:

```c
#include <cra.h>

int cra_ret_t cra(
    cra_event_t event,
    char *context,
    void *component_handles [],
    int log_fd
);
```

Where:
event

Passed from the infrastructure CRA module to each of the subsystem CRA modules. This argument is of type `cra_event_t`, which is defined as follows:

```c
typedef enum {
    ANALYZE,
    ANALYZE_LOCK,
    RELEASE
} cra_event_t;
```

Where:
- **ANALYZE**: All subsystem CRA modules must complete the analysis without holding any locks.
- **ANALYZE_LOCK**: If the subsystem CRA module implements locked analysis, it must complete the analysis holding a lock. The subsystem remains locked until a `RELEASE` event is sent to the subsystem CRA module.
- **RELEASE**: A subsystem CRA module implementing locked analysis must release the lock that was acquired during locked analysis and any other resources that were acquired or allocated but not released during the analysis phase.

context

Specifies a string that indicates the context in which a CRA is called. The infrastructure CRA module passes this argument to each of the subsystem CRA modules. Use one of the following context strings:

- CardDelete
- CardSuspend
- CardReset
- CardUsageReport
- LunDelete

A subsystem CRA module participates in analysis only if it recognizes the context string. Otherwise, the subsystem CRA interface must return `SUCCESS` for the call. The data types represented by component handles passed in the `component_handles` array must be interpreted based on the context string.

component_handles

Specifies an array of component handles for which a usage analysis must be done. This array of pointers terminates with a `NULL` pointer. The infrastructure CRA module passes this argument to each of the subsystem CRA modules.

- `component_handles[0]` points to the first component. `component_handles[1]` points to the second component. The array terminates for `N`, where `component_handles[N]` is `NULL`.

The data type of components passed in this array depends on the context string. For all of the contexts currently identified, the elements in the `component_handles` array are of type `char *`, representing hardware path strings.

Subsystem CRA modules must do a cumulative analysis on the set of components passed in the `component_handles` array. Cumulative CRA is a resource analysis that checks the overall impact of any and all components in the subsystem.

log_fd

Specifies a file descriptor passed from the infrastructure CRA module to the subsystem CRA modules. Subsystem-specific CRA reports must be written to `log_fd`. The log format and logging conventions are described
in “CRA Log File” (page 482). For ANALYZE and ANALYZE_LOCK events, subsystem-specific usage analysis logs must be written to log_fd. For the RELEASE event, a one-line success or failure statement must be written to log_fd.

For a list of return values, see “CRA Return Flags” (page 480).

### Common CRA Context Parameters

Table 21-1 shows typical values for the parameters that must be passed to subsystem CRA modules.

<table>
<thead>
<tr>
<th>context</th>
<th>event</th>
<th>component_handles</th>
<th>log_fd</th>
</tr>
</thead>
<tbody>
<tr>
<td>CardDelete,</td>
<td>ANALYZE_LOCK</td>
<td>List of hardware paths to be analyzed. Used for I/O cards being suspended, deleted, or reset.</td>
<td>Temporary files passed to subsystem CRA modules for logs.</td>
</tr>
<tr>
<td>CardSuspend,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CardReset</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LunDelete</td>
<td>ANALYZE_LOCK</td>
<td>List of hardware paths to be analyzed. Used for LUNs being deleted.</td>
<td>Temporary files passed to subsystem CRA modules for logs.</td>
</tr>
<tr>
<td>CardUsageReport</td>
<td>ANALYZE</td>
<td>List of hardware paths to be analyzed.</td>
<td>Temporary files passed to subsystem CRA modules for logs.</td>
</tr>
<tr>
<td>NULL</td>
<td>RELEASE</td>
<td>The same list of hardware paths passed during the previous invocation of CRA with ANALYZE_LOCK.</td>
<td>Temporary files passed to subsystem CRA modules for logs.</td>
</tr>
</tbody>
</table>

### CRA Return Flags

Each of the subsystem CRA cra entry points returns a flag of type cra_ret_t, which is defined as follows:

```c
typedef enum {
    CRA_SUCCESS,
    CRA_WARNING,
    CRA_DATA_CRITICAL,
    CRA_SYSTEM_CRITICAL,
    CRA_FAILURE
} cra_ret_t;
```

Where:

- **CRA_SUCCESS** Indicates that no critical usage is detected on the analyzed components.

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRA_WARNING</td>
<td>A warning issued because a resource is to be deleted, but does not cause any loss of services. A CRA_WARNING can be returned for the potential performance impact of a hardware redundancy to be deleted. For example, a warning is issued when one of the network links from a networking aggregate is going offline.</td>
</tr>
<tr>
<td>CRA_DATA_CRITICAL</td>
<td>A loss of resources classified as not critical to the running system. However, applications might be affected. If a CRA_DATA_CRITICAL flag is returned, the olrad command provides an option to override the flag and go ahead with the operation.</td>
</tr>
</tbody>
</table>

**NOTE:** If the interface path is suspended, the subsystem CRA module must return **CRA_WARNING**.
removal of I/O components. For example, if an IP address is configured on a networking port, it is considered a DATA CRITICAL usage.

**CRA_SYSTEM_CRITICAL**

Critical resources are in use. If these resources are taken offline, the system might crash. Swap and boot configurations are examples. With system critical resources in use, the corresponding I/O component must not be taken offline.

**CRA_FAILURE**

Internal errors occurred during the analysis.

CRA Header File and Library Details

The function prototypes and data structure definitions described in this section are delivered in the `/usr/include/sys/cra.h` header file.

Subsystem CRA modules must be delivered as both 32-bit and 64-bit shared library files. On PA-RISC systems, 32-bit subsystem CRA shared libraries must be delivered to the `/usr/lib/cra` directory. 64-bit subsystem CRA shared libraries must be delivered to the `/usr/lib/pa20_64/cra` directory.

On Integrity systems, 32-bit subsystem CRA shared libraries must be delivered to the `/usr/lib/hpux32/cra` directory. 64-bit subsystem CRA shared libraries must be delivered to the `/usr/lib/hpux64/cra` directory.

There are no naming restrictions for subsystem CRA libraries. HP recommends that you provide library names that are similar to related subsystem names to avoid creating two libraries with the same name.

Input Parsing Guidelines

“CRA Interface” (page 477) describes the parameters that are passed to the subsystem CRA interface `cra`. If the `event` flag is `ANALYZE` or `ANALYZE_LOCK`, parse the parameters as follows:

1. Compare the context string that was returned with the set of context strings the subsystem CRA module recognizes.
   - If the subsystem CRA module does not recognize the context string, do not log anything to the log file. Return `CRA_SUCCESS`.
   - If the subsystem CRA module recognizes the context string, go to step 2.

   **NOTE:** Some subsystem CRA modules hold a subsystem lock during locked analysis, and release the lock only when an explicit `RELEASE` event is sent. If the process running a CRA gets killed before sending the `RELEASE` event, the subsystem remains in a locked state. For more information on locked analysis, see “Subsystem CRA Locked Analysis” (page 485).

2. Check the `component_handles` array.
   - If `component_handles` is NULL, log that no components were passed in for analysis. Return `CRA_SUCCESS`.
   - If `component_handles[0]` is NULL, log that no components were passed in for analysis. Return `CRA_SUCCESS`.
3. For all the component handles passed in the `component_handles` array, follow these steps:
   a. Verify that all the handles represent valid components in the system. For example, if
      hardware path strings are passed, verify that they represent valid paths in the system.
      If not, log a CRA failure. Return the CRA_FAILURE flag.
   b. Verify that the handles represent components that are relevant to this subsystem CRA.
      For example, when a SCSI card is being deleted, if the SCSI interface hardware path is
      passed to the networking subsystem CRA module, it might not be relevant for network
      analysis. Valid component handles that are not relevant to the subsystem CRA module
      must return CRA_SUCCESS (no usage found).

4. For all component handles relevant to the subsystem CRA (usually hardware path strings),
   follow these steps:
   a. If the event flag is ANALYZE_LOCK and the subsystem CRA module has implemented
      a locked analysis, get the subsystem-specific lock for CRA.
      If the event flag is ANALYZE or this subsystem CRA module does not implement locked
      analysis, go to the step b.
   b. Perform a cumulative analysis considering the impact of all the components being taken
      offline, and identify the criticality of usage for these components.
   c. Log analysis details to the file pointed to by `log_fd`.
   d. Return the appropriate criticality flag.

If the `event` flag is RELEASE, the subsystem CRA module must unlock and release resources.
Follow these steps:
1. If the subsystem CRA module does not implement locked analysis or if it does not have any
   states to be released, return CRA_SUCCESS.
   If the unlock and release of resources is successful, return CRA_SUCCESS.
   If there is an unlock failure, return CRA_FAILURE.

   **NOTE:** For any internal errors during analysis, such as memory allocation failures or ioctl
   failures, log the failure details and return CRA_FAILURE.

2. Write a one-line entry to the log file for each success or failure. List the subsystem name in
   the entry. The return code also indicates success or failure of the RELEASE event.

   Logs from subsystem CRA modules must follow the format described in “Subsystem CRA Logs”
   (page 482).

Subsystem CRA Modules and Execution Order

The infrastructure CRA module does not call the subsystem CRA modules in any particular
order and no mechanism imposes an order. Subsystem CRA modules must not impose or assume
any execution order.

CRA Log File

Each subsystem CRA module must provide a detailed log of resource usage for its corresponding
subsystem. The infrastructure CRA module combines these logs and provides a systemwide
usage report for I/O components.

Subsystem CRA Logs

Subsystem CRA modules log usage reports containing analysis details to a usage analysis log
file passed from the infrastructure CRA module. Usage report size varies depending on the
number of I/O components analyzed and the usage statistics. Logs of varying sizes can be written
to the usage analysis log file. Because there is no limit on the file size, the file grows as details are written to it.

**NOTE:** If a subsystem CRA analysis detects no resource usage, by default the infrastructure CRA module does not include the corresponding logs in the combined log buffer. This keeps the CRA logs brief and relevant. This behavior is tunable. You can set the `CRA_TRACE_LOG` shell environment variable to include all subsystem CRA logs in the log buffer.

### CRA Log Conventions

Follow these conventions when writing analysis details to a CRA log file:

- The contents must be in plain ASCII format.
- Subsystem CRA logs must follow the format.
- Each subsystem CRA log entry starts with a newline character.
- Log entry lines must not exceed 80 characters.
- The subsystem CRA log format has a Detailed Report section. This section must provide system administrators with detailed information on resource usage in a subsystem.

**TIP:** You can use subsystem-specific terms that are already mentioned in the manpages and other released documentation in the log file. However, HP recommends that you not log data structure states and other internal information to the CRA log file.

In case of CRA errors (CRA_FAILURE), HP recommends that you log one line, stating where the analysis failed, then return FAILURE. If a subsystem CRA finds any resource usage, indicate in the CRA log file which subsystem-specific command or tool can be used to get more details on the usage. This also helps system administrators to shut down the usage before initiating the same operation again. For example, if a LAN CRA finds networking usage, mention the `nwmgr` command for further queries and usage shutdown.

- Do not tamper with subsystem CRA log file descriptor offsets. Also, the log file descriptor must not be closed to subsystem CRA modules. If subsystem CRA modules create a stream (FILE*) over a file descriptor that is passed, the stream must be flushed from the file before returning to the infrastructure CRA.
- Do not add any delimiters to mark a log entry.
- When the RELEASE event flag is passed, log entries must be a brief one-line statement indicating the success or failure of the RELEASE event.
- The CRA log buffer ends with the delimiter string `_CrA_BuF_EnD_`. The buffer must be traversed until this delimiter string is reached to get the log file size. The `strlen` routine might not always return the correct buffer size because the buffer can contain \0 delimiters in between. A `get_cra_log_size` routine provided in the infrastructure CRA shared library traverses the CRA log buffer and returns the correct size.
- Subsystem CRA modules write to the CRA log file, but writing string delimiters (\0) to the CRA log file, is not recommended. For example, the following example writes a newline, but it also writes a \0 delimiter to the log file:

```c
write (log_fd, "\n", strlen("\n"));
```
- Some subsystem CRA modules can provide an exhaustive usage analysis report, including detailed lists of processes and other user space entities that can be impacted. An optional, subsystem-specific environment variable can control the amount of detail logged based on the value of this variable.
Subsystem CRA Log Format Examples

A subsystem analysis report (log) must contain the following sections:

**ANALYSIS SCOPE**
A short description of the usage analysis scope covered by a subsystem CRA.

**RESULT**
A one-line statement that reports the analysis results.

**DETAILED REPORT**
Must include specific descriptions of resources, their usage, and related details. Must also list the components that were analyzed, including hardware paths (for example, 1/0/0/1 and 1/0/0/2). Other subsystem-specific information is included in this section of the subsystem analysis report.

The following examples contain a few typical networking subsystem scenarios that illustrate the subsystem CRA log format.

**No Usage on Hardware Paths Example**
A subsystem CRA identifies no usage on the hardware paths passed in either because the hardware paths have no usage associated with the subsystem or because the hardware paths passed in are not relevant to the scope of the analysis.

**ANALYSIS SCOPE: NETWORKING**
This report provides details of any networking related usages for a set of hardware paths in the system.

**RESULT:** No resource usage detected.

**DETAILED REPORT:** No components are passed in for analysis.

**USEFUL NETWORKING COMMANDS:**
lanadmin lanscan nwmgf netstat ifconfig linkloop

**Data Critical Usage with Hardware Paths Example**
A subsystem CRA identifies `DATA_CRITICAL` usage with the hardware paths.

**ANALYSIS SCOPE: NETWORKING**
This report provides details of any networking related usages for a set of hardware paths in the system.

**RESULT:** DATA-CRITICAL resource usage detected.

**DETAILED REPORT:** Analyzed the following hardware paths to detect any usages in the system:
1/0/1/1/0/4/0 (lan0)

**DATA CRITICAL RESULTS:**

Interface lan0: IPv4 address 192.0.2.42

**USEFUL NETWORKING COMMANDS:**
lanadmin lanscan netmgr netstat ifconfig linkloop

**Internal Error During Analysis Example**
A subsystem CRA encountered an internal error during analysis. An ioctl to query the usage might have failed.

**ANALYSIS SCOPE: NETWORKING**
This report provides details of any networking related usages for a set of hardware paths in the system.

**RESULT:** FAILURE during resource usage analysis.
DETAILED REPORT: Analyzed following hardware paths to detect any usages in the system:
1/0/1/0/4/0 (lan0)

DATA CRITICAL RESULTS:

Error: Failed to get usage information from DLPI.

System Critical Usage with Hardware Paths Example
A subsystem CRA identifies SYSTEM_CRITICAL usage with the hardware paths.

ANALYSIS SCOPE: MASS STORAGE
This report provides details of any critical mass storage hardware path usages in the system.

RESULT: SYSTEM CRITICAL resources will be affected

DETAILED REPORT: Analyzed the following mass storage hardware paths to detect any critical usages in the system:
1/0/1/0/1/0
1/0/1/0/1/1

SYSTEM CRITICAL RESULTS

Affected File Systems:
/ under the affected card(s)
1/0/1/0/1/0/1

CRA Log File for OL* Operations
During PCI card OL* operations, CRA is invoked from the olrad command. The contents of the buffer returned from the infrastructure CRA module are logged in the /var/adm/cra.log file by the olrad command. The olrad command output indicates to the user that detailed CRA logs are available in the /var/adm/cra.log file.

Logs from the RELEASE event, which is sent to unlock subsystem CRA modules, are not written to the /var/adm/cra.log by the olrad command. If the RELEASE event succeeds, the log contents in the buffer are ignored. If the RELEASE event fails, the contents of the log buffer are written to the olrad command output.

CRA Synchronization Scenarios
This section describes different synchronization scenarios during CRA.

Subsystem CRA Locked Analysis
Subsystem CRA modules can hold a subsystem level lock during CRA. This is a locked analysis. The lock prevents any new resource usage from being attached to I/O components. With a locked analysis, the conditions reported in the CRA usage report do not change until an explicit RELEASE event releases the lock.

NOTE: If a subsystem CRA module implements locked analysis, a try_lock interface must be provided for the lock.

Performing CRA on a 4-Port LAN Card
Following are the CRA steps for a 4-port LAN card that is being either suspended or deleted using the olrad command.

1. The user runs the olrad command to suspend or delete a 4-port LAN card in a PCI I/O slot.
2. The `olrad` command invokes the infrastructure CRA module to export the generic CRA interface to the user space. It also passes the hardware paths corresponding to the four LAN interface ports to the subsystem CRA modules for a locked analysis.

3. The infrastructure CRA module acquires a user space lock that ensures only one CRA instance can operate at a time. This is a locked analysis.

4. The infrastructure CRA module invokes the locked analysis entry points that are exported from each of the subsystem CRA shared libraries. Then it passes the four hardware paths to each of the subsystem modules. For more information about locked analysis, see “Input Parsing Guidelines” (page 481).

5. Each subsystem CRA module confirms whether any of the four LAN interface hardware paths are in use in its corresponding subsystem.

**NOTE:** The subsystem CRA modules perform a cumulative CRA. A cumulative CRA analyzes the overall impact if all LAN ports are taken offline. For example, with four LAN ports, the networking subsystem CRA module can detect that one or more of the LAN ports have active IP addresses configured, but the mass storage subsystem CRA module finds that none of these interfaces are connected to storage. Therefore, networking usage is detected but mass storage usage is not.

6. Each subsystem CRA module entry point returns a flag to the infrastructure CRA module indicating how critical the resource usage is. The infrastructure CRA module identifies the highest critical usage level reported by any of the subsystem CRA modules, and returns a flag corresponding to this critical usage level to the `olrad` command. Each subsystem CRA module also provides a detailed report of its resource analysis. The infrastructure CRA module combines analysis details from all subsystem CRA modules and returns a single collated buffer to the `olrad` command. For more information on the CRA log file, see “CRA Log File” (page 482).

7. The CRA logs for OL* operations are written to the `/var/adm/cra.log` file. The `olrad` command adds audit information and analysis details from the buffer to `/var/adm/cra.log`. For example, the `olrad` command checks the flag returned from the infrastructure CRA module for any critical resources in use for any of the four LAN ports. If the flag is `CRA_DATA_CRITICAL`, the `olrad` command might not allow the LAN card to be suspended or deleted.

A subsystem CRA module that has done a locked analysis remains in the locked state while the card is deleted.

8. The `olrad` command invokes the infrastructure CRA module entry point again to release any subsystem CRA modules that have done a locked analysis. Then the infrastructure CRA module invokes all subsystem CRA module entry points to unlock any subsystem CRA module in a locked state. Subsystem CRA modules that do not use locked analysis always return `SUCCESS` for this event.

Logs from release events are brief. Each subsystem CRA module logs one line with the subsystem name and an unlock succeeded or unlock failed statement. If a subsystem CRA module fails the `RELEASE` event, a failure code is returned.

At the end of a `RELEASE` event, the infrastructure CRA module releases the user space lock held at the start of analysis and returns to the `olrad` command. The infrastructure CRA module collates the release logs from the subsystem CRA modules in a buffer and returns this buffer and the return code to the `olrad` command.

If a `RELEASE` event returns failure, the `olrad` command writes the buffer with the release event logs to the `olrad` command output. If a release event returns success, the `olrad` command ignores these logs. In either case, the logs from the release event are not logged to the CRA log file by the `olrad` command.
CRA Flow Examples

To minimize complexity, the CRA framework in the following examples is limited to an infrastructure CRA module, a networking subsystem CRA plug-in module, and a mass storage subsystem CRA plug-in module. This CRA framework is atypical because actual frameworks typically include many more subsystem CRA modules.

PCI Card Delete - CRA Successful

The following example illustrates a successful CRA during a PCI card online deletion operation.

1. The `olrad` command invokes the infrastructure CRA module to analyze resource usage for the card.
2. The infrastructure CRA module acquires the user space CRA lock to ensure only one instance of CRA is active on the system. The infrastructure CRA module invokes `cra` from both the networking and mass storage subsystem CRA modules with the following arguments:
   
   | event       | ANALYZE_LOCK |
   | context     | CardDelete   |
   | component_handles[] | List of hardware path strings associated with the I/O card being deleted. |
   | log_fd      | Temporary file passed to the subsystem CRA to log analysis details. |

3. Both the networking and mass storage subsystem CRA modules complete the analysis. Analysis details are logged in the temporary files that are passed to the infrastructure CRA. Both subsystem CRA modules return `CRA_SUCCESS` to the infrastructure CRA.
4. Because the subsystem CRA modules detected no resource usage (and if the `CRA_TRACE_LOG` flag is not set), the infrastructure CRA module ignores the log details from the temporary files. The infrastructure CRA module adds a one-line summary of the analysis result to the log buffer.
5. The infrastructure CRA module returns the `CRA_SUCCESS` flag to the `olrad` command. The log buffer that contains the log details from the subsystem CRA modules is also returned to the `olrad` command.
6. The `olrad` command writes the log buffer contents to `/var/adm/cra.log`. Because the CRA succeeded, the `olrad` command deletes the card.
7. The `olrad` command invokes the infrastructure CRA module to perform the `RELEASE` operation.
8. The infrastructure CRA module invokes `cra` from both the networking and mass storage subsystem CRA modules with the following arguments:
   
   | event       | RELEASE     |
   | context     | NULL        |
   | component_handles[] | List of hardware path strings associated with the I/O card being deleted. |
   | log_fd      | Temporary file passed to the subsystem CRA modules for logs. |

9. Both the networking and mass storage subsystem CRA modules return `CRA_SUCCESS` for the `RELEASE` event. The `RELEASE` event success is logged in the temporary files that are then passed to the infrastructure CRA. The infrastructure CRA module collects logs from the respective temporary files and collates them into a buffer.
10. The infrastructure CRA module releases the user space lock acquired in step 2. The infrastructure CRA module returns the `CRA_SUCCESS` flag to the `olrad` command.
PCI Card Delete - CRA Detects Data Critical Resources

This example illustrates CRA detecting critical resources in use during a PCI card online deletion operation.

1. The \texttt{olrad} command invokes the infrastructure CRA module to analyze resource usage for the card.

2. The infrastructure CRA module acquires the user space CRA lock to ensure only one instance of CRA is active in the system. The infrastructure CRA module invokes \texttt{cra} from both the networking and mass storage subsystem CRA modules with the following arguments:
   \begin{verbatim}
   event         ANALYZE_LOCK
   context       CardDelete
   component_handles/1 List of hardware path strings associated with the I/O card being deleted.
   \end{verbatim}
3. Both the networking and mass storage subsystem CRA modules complete the analysis. Analysis details are logged in the temporary file that then passes to the infrastructure CRA module. The networking subsystem CRA module detects data critical usage and returns \texttt{CRA\_DATA\_CRITICAL} to the infrastructure CRA module. The mass storage subsystem CRA module returns \texttt{CRA\_SUCCESS} to the infrastructure CRA module.

4. If the \texttt{CRA\_TRACE\_LOG} flag is not set, the infrastructure CRA module ignores the log details from the mass storage subsystem CRA because the analysis was successful. The analysis details from the networking subsystem CRA that detected data critical resource usage are collated to the CRA log buffer. The infrastructure CRA module also adds a one-line summary indicating the overall analysis result to the log buffer.

5. The infrastructure CRA module returns the \texttt{CRA\_SUCCESS} flag to the \texttt{olrad} command. The log buffer that contains the log details from the subsystem CRA modules is also returned to the \texttt{olrad} command.

6. The \texttt{olrad} command writes the log buffer contents to \texttt{/var/adm/cra.log}. Because the CRA detected data critical usage, the \texttt{olrad} command cannot allow the card to be deleted.

7. The \texttt{olrad} command invokes the infrastructure CRA module to perform the \texttt{RELEASE} operation.

8. The infrastructure CRA module invokes \texttt{cra} from both the networking and mass storage subsystem CRA modules with the following arguments:
   
   \begin{verbatim}
   event         RELEASE
   context       NULL
   component_handles [ ] List of hardware path strings associated with the I/O card being deleted.
   log_fd        Temporary file passed to subsystem CRA modules for log.
   \end{verbatim}

9. Both the networking and mass storage subsystem CRA modules return \texttt{CRA\_SUCCESS} for the release event, and a one line summary of the release event success is logged in the temporary file passed to the infrastructure CRA module. The infrastructure CRA module collects logs from the respective temporary files and collates them into a buffer.

10. The infrastructure CRA module releases the user space lock acquired at the beginning of analysis, in step2. The infrastructure CRA module returns the \texttt{CRA\_SUCCESS} flag to the \texttt{olrad} command.
PCI Card Delete - Internal Failure During Analysis

This example illustrates CRA detection of an internal failure during a PCI card online deletion operation.
1. The \texttt{olrad} command invokes the infrastructure CRA module to analyze resource usage for the card. The \texttt{olrad} command invokes the infrastructure CRA interface \texttt{cra}, passing the following arguments:

- \textit{event}: \texttt{ANALYZE\_LOCK}
- \textit{context}: \texttt{CardDelete}
- \textit{component\_handles[ ]}: List of hardware path strings associated with the I/O card being deleted.
- \textit{buf}: Return buffer with analysis details. The \texttt{olrad} command passes a \texttt{char**} pointer containing the address of the buffer with the analysis logs.

2. The infrastructure CRA module acquires the user space CRA lock to ensure only one instance of CRA is active in the system. The infrastructure CRA module invokes \texttt{cra} from both the networking and mass storage subsystem CRA modules with the following arguments:

- \textit{event}: \texttt{ANALYZE\_LOCK}
- \textit{context}: \texttt{CardDelete}
- \textit{component\_handles[ ]}: List of hardware path strings associated with the I/O card being deleted.
- \textit{log\_fd}: Temporary file passed to the subsystem CRA modules to log analysis details.

3. The networking subsystem CRA module finds an internal failure during analysis. Failure details are logged in the temporary file and passed to the infrastructure CRA module. The networking subsystem CRA module returns \texttt{CRA\_FAILURE} to the infrastructure CRA module.

4. The mass storage subsystem CRA module is not called in this case because the analysis has already failed.

5. The infrastructure CRA module collects the failure log from the temporary file and collates it in a buffer. The infrastructure CRA module adds a one line summary to the log buffer, indicating the results of the analysis.

6. The infrastructure CRA module returns the \texttt{CRA\_FAILURE} flag to the \texttt{olrad} command. The log buffer that contains the log details from the subsystem CRA modules is also returned to the \texttt{olrad} command.

7. The \texttt{olrad} command prints the failure logs from the buffer to the command output. The failure logs do not go to the \texttt{/var/adm/cra.log} file. Because the CRA failed, the \texttt{olrad} command does not allow the PCI card to be deleted.

8. The \texttt{olrad} command invokes the infrastructure CRA module to perform the \texttt{RELEASE} operation.

9. The infrastructure CRA module invokes \texttt{cra} from both the networking and mass storage subsystem CRA modules with the following arguments:

- \textit{event}: \texttt{RELEASE}
- \textit{context}: \texttt{NULL}
- \textit{component\_handles[ ]}: List of hardware path strings associated with the I/O card being deleted.
- \textit{log\_fd}: Temporary file passed to the subsystem CRA modules for logs.

10. Both the networking and mass storage subsystem CRA modules return \texttt{CRA\_SUCCESS} for the release event, and a one-line summary of the release event is logged in the temporary file that is passed to the infrastructure CRA module. The infrastructure CRA module collects logs from the temporary files provided by each subsystem CRA module and collates them into a buffer.
11. The infrastructure CRA module releases the user space lock acquired in step 2. The infrastructure CRA module returns the CRA_SUCCESS flag to the olrad command.

**Figure 21-3 CRA Flow During PCI Card Delete - Internal Failure During Analysis**

**PCI Card Delete - Subsystem CRA Module Fails RELEASE Event**

This example illustrates user space lock RELEASE event failure during a PCI card online deletion operation.

1. The olrad command invokes the infrastructure CRA module to analyze resource usage for the card.
2. The infrastructure CRA module acquires the user space CRA lock to ensure only one instance of CRA is active in the system. The infrastructure CRA module invokes cram from both the networking and mass storage subsystem CRA modules with the following arguments:
   - event: ANALYZE_LOCK
   - context: CardDelete
component_handles[ ]  List of hardware path strings associated with the I/O card being deleted.

log_fd  Temporary file passed to the subsystem CRA modules to log analysis details.

3. Both the networking and mass storage subsystem CRA modules complete the analysis. Analysis details are logged in the temporary file that is passed to the infrastructure CRA module. Both the networking and mass storage subsystem CRA modules return CRA_SUCCESS to the infrastructure CRA module.

4. Because the subsystem CRA modules detected no resource usage (and if CRA_TRACE_LOG flag is not set), the infrastructure CRA module ignores the log details from the subsystem CRA temporary files. The infrastructure CRA module adds a one-line summary indicating the analysis result to the log buffer.

5. The infrastructure CRA module returns the CRA_SUCCESS flag to the olrad command. The log buffer that contains the log details from the subsystem CRA modules is also returned to the olrad command.

6. The olrad command writes the log buffer contents to the /var/adm/cra.log file. Because the CRA succeeded, the olrad command deletes the PCI card.

7. The olrad command invokes the infrastructure CRA module to perform the RELEASE operation.

8. The infrastructure CRA module invokes cra from both the networking and mass storage subsystem CRA modules with the following arguments:

   event RELEASE
   context NULL
   component_handles[ ]  List of hardware path strings associated with the I/O card being deleted.
   log_fd  Temporary files passed to subsystem CRA modules for logs.

9. The networking subsystem CRA module returns CRA FAILURE for the RELEASE event. The mass storage subsystem CRA module returns CRA_SUCCESS. The infrastructure CRA module collects the subsystem CRA release logs from the temporary files and collates them into a buffer.

10. The infrastructure CRA module releases the user space lock acquired in step 2. The infrastructure CRA module returns the CRA FAILURE flag to the olrad command.
Figure 21-4 CRA Flow During PCI Card Delete - Subsystem CRA Module Fails RELEASE Event

olrad
Command
Mass Storage CRA
Module
Infrastructure CRA
Module
Networking CRA
Module
Delete Card

ANALYZE_LOCK
CRA SUCCESS
ANALYZE_LOCK
CRA SUCCESS
ANALYZE_LOCK
CRA SUCCESS
RELEASE
CRA SUCCESS
RELEASE
CRA SUCCESS
RELEASE
CRA SUCCESS
CRA FAILURE
22 Interrupt Migration

At boot time, the HP-UX kernel allocates external card interrupts to CPUs in a round-robin method. The kernel is not aware of card traffic patterns at that time. The boot allocation scheme can allow two heavily loaded cards to map interrupts to the same CPU. This can 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/contrib/bin/intctl` command enables the user to display the interrupt configuration of the system and to migrate the interrupts between CPUs.

This chapter addresses the following topics:

- “Interrupt Migration Impact on Drivers”
- “Interrupt Migration Event Masks and Registration with WSIO” (page 497)
- “Interrupt Migration Flow” (page 507)
- “Assumptions and Dependencies” (page 508)

Interrupt Migration Impact on Drivers

For the most part, interrupt migration is transparent to applications. The associated cards need not be stopped during migration between processors. The applications using the cards are not affected by these operations. The only exception is if a high availability operation like `olrad`, `ioscan`, or a DLKM load or unload is initiated while an interrupt migration is in progress. In this case, the high availability operation might return an error.

Drivers must be designed to spend minimal time in their Interrupt Service Routines (ISRs). They might need to stop the card interrupts during a migration operation, but stopping the card DMA engines is not necessary.

In the HP-UX environment, drivers can use **Line Based Interrupts** (LBIs), **Transaction Based Interrupts** (TBIs), or **Message-Signaled Interrupts** (MSIs).

**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 LBIs has not registered for these interrupt event flags with WSIO, the associated card interrupts migrate to a different processor without informing the driver. Drivers that use I/O forwarding must be informed which CPU is handling an interrupt, otherwise they cannot update internal information and continue to forward the interrupt to the offline CPU.

**TBI Drivers**

Drivers using TBIs spread the interrupt load across CPUs in the system and can interrupt more than one CPU. The drivers must program their cards with the interrupt specifics (CPU address and data vector), so any TBI interrupt migration is processed by the driver. The TBI drivers must first register with WSIO to use the new events.

WSIO provides new event masks for the drivers. The drivers are notified when a new CPU is enabled for interrupts in the system.

**MSI and MSI-X Drivers**

Drivers using MSIs direct all their interrupts to a single CPU. Drivers using MSI-X interrupts can spread their interrupt load across multiple CPUs. However, both types of drivers program their cards with the interrupt address of a CPU, so the driver must be involved in interrupt migration.
MSI and MSI-X drivers must register with WSIO to receive migration event notification. They are notified when a CPU is being disabled or reserved, and can choose to be notified when a new CPU is enabled for interrupts.

CLM Services

Drivers that use Cell Local Memory (CLM) services have the following new WSIO interrupt services available in HP-UX 11i v3:

- `wsio_intr_req_loc`
- `wsio_intr_get_loc`
- `wsio_intr_assign_cpus`

These new services are required for drivers that preallocate I/O card memory objects in the same location as the CPU that handles interrupts for that I/O card. This is done before setting up and enabling interrupts. For example, a driver can claim an I/O card in the driver attach routine and allocate a number of control structures, but does not set up interrupts until its `init` routine is called.

The `wsio_intr_req_loc` service can be called to request a set of one or more interrupt location vectors. The caller specifies how many interrupt vectors are needed and provides hints about the priority of the driver, or which location to distribute the driver's interrupts. Then the WSIO CDIO service calls the underlying platform services to select locations. The vectors can be retrieved later by the driver by calling `wsio_intr_get_loc`. Each vector consists of an `intr_id` and a location for the interrupt vector. The location can be used for allocating memory for control structures in the same location as the CPUs that handle the interrupts. The underlying platform services select the location and vector assignments based on the hints passed in by `wsio_intr_req_loc` and on internal algorithms that attempt to load balance interrupts and local memory allocation. The policy for location and interrupt distribution is in the system firmware.

The `wsio_intr_get_loc` service can be called to obtain the interrupt location assignments. The information returned is an array of one or more vectors, depending on the requested number.

The `wsio_intr_assign_cpus` service can be called to assign CPUs to the location vectors returned by `wsio_intr_get_loc`. The caller passes in the list of interrupt location vectors, then the underlying platform services assign CPUs to them based on the location information.

The `wsio_intr` services provided in HP-UX 11i v2 are still available to set up LBI or TBI interrupts for drivers that use only a single interrupt resource. Most drivers using the `wsio_intr` services in HP-UX 11i v2 allocate only a single interrupt resource or object for each driver instance (PCI card). These drivers are not required to call the new WSIO CLM services to set up their interrupts, if they do not want to allocate structures local to the CPU handling their thread. Drivers using `isrlink` do not use the functionality provided by the WSIO CLM services. Drivers using `isrlink` can port their drivers to use WSIO interrupt services, which have been modified to set up PCI card interrupts with the location defaulted to the PCI card.

**NOTE:** The `isrlink` and `isrunlink` services are deprecated.

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, but only if all the drivers involved are MP-safe. If any driver sharing an interrupt line is not MP-safe, interrupts on that line are not migrated.
Interrupt Migration Event Masks and Registration with WSIO

WSIO defines three event flags as part of interrupt migration. One is for LBI based drivers. The other two are for TBI based and MSI and MSI-X based drivers.

LBI Event Flags and Migration Algorithm

Upon registration, an optional event flag, `WSIO_EVENT_LBI_INTR_MIGR`, notifies the driver when the specified LBI is migrated to a new CPU.

`WSIO_EVENT_LBI_INTR_MIGR`

Drivers of cards using LBIs can register for this event flag using the `wsio_reg_drv_capability_mask` call. A driver registers for events in its `driver_attach` routine after claiming the card it controls by calling `wsio_claim_node`. The driver must also register for an event handler in its `driver_install` routine. Registration for this event flag is needed only if the driver has cached the interrupt CPU. With an interrupt migration initiated, the interrupting CPU changes to a different CPU. If the driver registered for this event flag, it is notified of this CPU change through the event handler.

Following is an example of a driver registering for a `WSIO_EVENT_LBI_INTR_MIGR` event:

```c
driver_attach(...) {
    wsio_event_mask_t newmask;
    ...
    wsio_claim_node(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 v3 Driver Development Reference*.

If the driver has registered for the event flag, its event handler is invoked twice by WSIO with the `WSIO_EVENT_LBI_INTR_MIGR` flag. 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` structure passed to the driver event handler. The `wsio_intr_migr_info_t` structure inside the `wsio_generic_event_t` structure differentiates the two notifications. For example:

```c
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
genetic_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 {
    wsio_intr_object_t    intr_obj;  // Associated interrupt object
    intptr_t              dest_spu;  // '-1' or new CPU id
    wsio_intr_migr_info_t migr_info; // WSIO_LBI_INTR_MIGR_NOTIFY
    // or WSIO_LBI_INTR_MIGR_COMPLETE
    wsio_ret_code_t       ret_val;   // Return value set by the driver
    void                 *resvd;     // Reserved field
} wsio_intr_migr_t;
```

After the driver handles the event, it invokes the WSIO-provided completion callback routine to indicate that the event is complete. The code for the completion callback routine is as follows:
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);

Where:
isc Pointer to the ISC structure.
event_id Event ID.
wsi0_intr_migr_p Pointer to the wsio_intr_migr_t that passed to the driver event handler through the arg field of the wsio_generic_event_t structure.

Figure 22-1 shows the flow of LBI events.

Figure 22-1 LBI Interrupt Migration

Return Completion Status

Invoke Event Handler

Migrates Interrupt to New CPU

Migration Notify

Return Completion Status

Invoke Event Handler

WSIO Driver

Return Completion Status

The only permissible return value from the driver for this event flag is WSIO_OK.

NOTE: The driver event handler must not register for a timeout call to invoke the WSIO-provided callback routine.

The specific steps are as follows:

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 hardware-dependent routines to migrate the LBI to the new CPU.

4. The migration causes a false interrupt on the CPU to which the interrupt of this card has been migrated. Drivers must be capable of handling false 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 dest_spu 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 and Migration Algorithm

Two flags are used as part of interrupt migration for drivers using TBIs. The first flag, WSIO_EVENT_OFFLINE_CPU, is a mandatory event. Upon registration, it notifies the driver when the concerned interrupt is migrated to a new CPU.
The second flag, WSIO_EVENT_ONLINE_CPU, is an optional event. Upon registration, it notifies the driver of any CPU being enabled for interrupts.

**WSIO_EVENT_OFFLINE_CPU**

This event flag is used by drivers using TBIs for their cards. All drivers using TBIs 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 in its driver_attach routine after claiming the card it controls by calling wsio_claim_node. The driver also must register for an event handler in its driver_install routine.

Following is an example of a driver registering for a WSIO_EVENT_OFFLINE_CPU event:

```c
driver_attach(...) {
    wsio_event_mask_t newmask;
    ...
    wsio_claim_node(isc);
    newmask = oldmask | WSIO_EVENT_OFFLINE_CPU;
    ret = wsio_reg_drv_capability_mask( isc, newmask);
    ...
}
```

For a description of the wsio_reg_drv_capability_mask call, see the HP-UX 11i v3 Driver Development Reference.

The interrupt objects are allocated in the driver_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 disabled or reserved for interrupts.
- The card interrupt is binding 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 {
    wsio_intr_object_t    intr_obj;  // Associated interrupt object
    intptr_t              dest_spu;  // '-1' or new CPU ID
    wsio_intr_migr_info_t migr_info; // NULL for this event
    wsio_ret_code_t       ret_val;   // Return value set by the driver
    void                 *resvd;     // Reserved field
} wsio_intr_migr_t;
```

After the driver handles the event, it invokes the WSIO-provided completion callback routine to indicate that the event is complete. 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);
```

Where:

- `isc` Pointer to the ISC structure.
- `event_id` Event identification.
Upon return from the driver event handler, the status is set in the `ret` field inside the `wsio_intr_migr_t` structure. The valid return values from the driver event handler are the following:

- **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**: Success.

Figure 22-2 shows the flow of TBI events.

---

**NOTE:** The driver event handler must not register for a timeout call to invoke the WSIO-provided callback routine.

TBI interrupt migration follows these steps:

1. WSIO invokes the driver's event handler using the `WSIO_EVENT_OFFLINE_CPU` flag.
2. The driver performs the following tasks to migrate the interrupt from the CPU:
a. Disables the card interrupts. Only the interrupt corresponding to the interrupt object passed in the `wsio_generic_event_t` is disabled. It does not need to disable all other interrupts associated with this card.

b. Invokes the `wsio_intr_deactivate` routine for the interrupt object passed in the `wsio_generic_event_t` structure. This disables the interrupt at the higher levels. For example, the driver’s ISR is removed from the CPU interrupt switch table. This can cause pending interrupts to be lost. To nullify these effects, a false interrupt is generated when the driver calls `wsio_intr_activate` (described later in this sequence).

c. Invokes the `wsio_intr_set_cpu_spec` routine. This migrates the interrupt object sent in the `wsio_generic_event_t` structure to the new CPU. See the `wsio_intr_set_cpu_spec` call for more information.

d. Invokes the `wsio_intr_get_assigned_cpu` routine to determine the transaction address of the new CPU to which the interrupt has been migrated, and invokes the `wsio_intr_get_txn_info` routine to determine the transaction data.

e. Invokes the `wsio_intr_activate` routine to enable the interrupt object passed in the `wsio_generic_event_t` structure. This interface results in a false interrupt being generated on the migrated CPU. Drivers must be capable of handling these false interrupts.

f. Programs the card to use the new transaction address and transaction data obtained in step d.

g. Returns the completion status (WSIO_OK) with the `wsio_completion_cb` routine.

**WSIO_EVENT_ONLINE_CPU**

This event flag is used by drivers using TBIs. If the driver 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 `driver_attach` routine after claiming the card it controls by calling `wsio_claim_node`. The driver must also register for an event handler in its `driver_install` routine.

Following is an example of a driver registering for a WSIO_EVENT_ONLINE_CPU event:

```c
driver_attach(...) {
    wsio_event_mask_t newmask;

    wsio_claim_node(isc);
    newmask = oldmask | WSIO_EVENT_ONLINE_CPU;
    ret = wsio_reg_drv_capability_mask(isc, newmask);
    ...
}
```

For a description of the `wsio_reg_drv_capability_mask` call, see the *HP-UX 11i v3 Driver Development Reference*. 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 CPUs enabled
} wsio_generic_event_t;
```
MSI and MSI-X Event Flags and Migration Algorithm

MSI and MSI-X based drivers use the same two events (WSIO_EVENT_OFFLINE_CPU and WSIO_EVENT_ONLINE_CPU) as drivers using TBIs, so most of the information and algorithm is identical. MSI and MSI-X based drivers use a third event, WSIO_EVENT_MSI_INTR_MIGR.

To support interrupt migration, MSI and MSI-X drivers register for the following event flags:

**WSIO_EVENT_OFFLINE_CPU**

This is a mandatory event. Upon registration, it notifies the driver when a CPU is being disabled or reserved for interrupts, so the individual MSI or MSI-X vectors programmed with the specified CPU migrate to a different CPU.

**WSIO_EVENT_ONLINE_CPU**

This is an optional event. It informs the driver of any new CPUs that are available for external interrupts.

**WSIO_EVENT_MSI_INTR_MIGR**

This is a mandatory event. It notifies the driver to program a specific vector with a specific CPU. The driver event handler is passed in a pointer to a wsio_intr_migr_msi_t structure in the event_ptr->arg field. The driver gets the MSI-X vector number and new CPU from this structure, and reprograms the vector.

To receive these events, the driver must register a generic event handler using wsio_install_drv_event_handler, passing pointers to the driver's wsio_drv_info structure and its event handling function. This must be done in the driver_install routine after the call to wsio_install_driver. Following is an example of a PCI driver registering its event handler:

```c
void my_driver_install(void)
{
    ...
    if(wsio_install_driver(&my_drv_info) == WSIO_OK)
    {
        if(wsio_install_drv_event_handler(&my_drv_info,
            my_drv_handler) != WSIO_OK)
        {
            wsio_uninstall_driver(&my_drv_info);
            return;
        }
        my_drv_saved_attach = pci_attach;
        pci_attach = my_driver_attach;
    }
    return;
}
```

The driver must then register for the event flags with the wsio_reg_drv_capability_mask function. This is done in the driver_attach routine after claiming the card with wsio_claim_node. Following is an example of a driver registering for the WSIO_EVENT_OFFLINE_CPU event:

```c
driver_attach(...) {
    wsio_event_mask_t newmask;
    ...
    wsio_claim_node(isc);
    newmask = oldmask | WSIO_EVENT_OFFLINE_CPU;
    ret = wsio_reg_drv_capability_mask( isc, newmask);
    ...
}
```

For a description of the wsio_install_drv_event_handler and wsio_reg_drv_capability_mask functions, see the HP-UX 11i v3 Driver Development Reference.
All drivers using MSI or MSI-X based interrupts must register for the `WSIO_EVENT_OFFLINE_CPU` event flag. This is enforced by the WSIO interrupt object allocation function `wsio_msi_alloc`. If a driver attempts to allocate an MSI or MSI-X interrupt object and it has not registered for the `WSIO_EVENT_OFFLINE_CPU` event flag, WSIO fails the allocation with `WSIO_ERROR`.

When a CPU is disabled or reserved for interrupts, the WSIO subsystem invokes the driver handler with `wsio_generic_event_t` as the argument. The `event` field is set to `WSIO_EVENT_OFFLINE_CPU` and the `arg` field contains a pointer to a `wsio_intr_migr_t` structure. The `wsio_generic_event_t` and `wsio_intr_migr_t` structures are defined in `/usr/include/sys/wsio.h` 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;

typedef struct wsio_intr_migr {
    wsio_intr_object_t    intr_obj;  // Not used for MSI/MSI-X
    intptr_t              dest_spu;  // CPU disabled or reserved
    wsio_intr_migr_info_t migr_info; // NULL for this event
    wsio_ret_code_t       ret;       // Return value set by the driver
    void                   *resvd;     // Pointer to MSI/MSI-X interrupt object
} wsio_intr_migr_t;
```

After the driver handles the event, it invokes the WSIO-provided completion callback routine to indicate that the event is complete. The completion status must be set in the `ret_val` field of the `wsio_intr_migr_t` structure. If successful, the driver must set `ret_val` to `WSIO_OK`; any other value is treated as an error. The prototype and calling convention for the completion callback routine are as follows:

```c
typedef int (*generic_complete_callback_t) (struct isc_table_type *, wsio_event_id_t, void *);
int (*wsio_completion_callback)(isc, event_id, wsio_intr_migr_p);
```

Where:
- `isc` Pointer to the ISC structure.
- `event_id` Event identification passed in the `event_id` field of the `wsio_generic_event_t` structure.
- `wsio_intr_migr_p` Pointer to the `wsio_intr_migr_t` that passed to the driver event handler through the `arg` field of the `wsio_generic_event_t` structure.

Figure 22-3 shows the flow of MSI and MSI-X events.
**Figure 22-3 MSI and MSI-X Interrupt Migration**

NOTE: The driver event handler must not register for a timeout call to invoke the WSIO-provided callback routine.

NOTE: The driver event handler must not hold any spinlocks while calling `wsio_msi_assign`, `wsio_msi_disable`, or `wsio_msi_enable`.

MSI and MSI-X interrupt migration follows these steps:

1. WSIO invokes the driver’s event handler with the `wsio_generic_event_t` structure as the argument. The `event` flag is set to `WSIO_EVENT_OFFLINE_CPU`, and the `arg` field contains a pointer to a `wsio_intr_migr_t` structure. That structure contains the affected CPU in its `dest_spu` field and the driver’s interrupt object in its `resvd` field.

2. Using the interrupt object, the driver obtains a list of its interrupt vectors. The driver might have saved this list from a previous `wsio_msi_assign` call, or it can get the list using `wsio_msi_query`.

3. For each of the MSI/MSI-X vectors that have a CPU identifier matching the CPU in the `WSIO_EVENT_OFFLINE_CPU` event, the driver performs the following tasks:
   a. Disables the vectors using `wsio_msi_disable`.
   b. Decides whether to leave each vector disabled or to reassign it to a different CPU. If the driver leaves a vector disabled, it can skip the remaining tasks for this vector.
   c. Finds a new CPU by calling `wsio_msi_get_cpus`.
   d. Programs the vectors, setting the `mask` field to `WSIO_MSI_SET_CPU` and the `cpu_id` field to the new CPU.
e. Programs the vectors to new CPUs by calling `wsio_msi_assign`.

f. Calls `wsio_msi_enable` to enable the newly assigned vectors.

4. After processing all of the MSI or MSI-X vectors, the driver sets the completion status in the `ret_val` field of the `wsio_intr_migr_t` structure, and signals the completion of the event handling by calling `wsio_completion_cb`. If any of the migration operations failed, the driver must set the completion status to `WSIO_ERROR`. Otherwise it must be set to `WSIO_OK`.

For example:

```c
my_drv_event_hndler(wsio_generic_event_t *event_ptr) {
    wsio_msix_object_t *msi_hdl;
    ret, j, vector_cnt;
    cpu_ids[MAX_CPU_CNT];
    vec_vals;
    *migrp;

    msi_hdl = (wsio_msix_object_t *)(migrp->resvd);

    switch (event_ptr->event) { // Process specified event type
        CASE WSIO_EVENT_OFFLINE_CPU:
            // The CPU going is going offline.
            // The driver must disable any vectors using the
            // CPU and then determine if it wants to assign
            // them to another CPU or leave them disabled.

            msi_hdl = (wsio_msix_object_t *)(event_ptr->arg);

            // Disable all MSI vectors programmed with the old CPU.
            if (wsio_msi_disable(msi_hdl, old_cpu_id, 0, 0) != WSIO_OK) {
                migrp->ret_val = WSIO_ERROR;
                goto ERR_RET;
            }

            // Determine if we want to re-enable the vectors with
            // another CPU, and if so select a new one.

            if (not leave disabled) {
                // Get the list of available CPUs in the cards
                // and select a new one.
                ret = wsio_msi_get_cpus(isc, &cpu_list, &cnt,
                                       WSIO_MSI_CPU_LOCAL);

                new_cpu = determine the next cpu to use from cpu_list;
                vec_vals.mask = WSIO_MSI_SET_CPU;
                vec_vals.cpu_id = new_cpu;
                type = WSIO_MSI_CPU_ID | WSIO_MSI_ASSIGN_ALL;
                ret = wsio_msi_assign(msi_hdl, type, &vec_vals,
                                      new_cpu, 0);

                if (ret == WSIO_OK)
                    ret = wsio_msi_enable(msi_hdl, new_cpu, 0, 0);
                else
                    migrp->ret_val = WSIO_ERROR;
                    goto ERR_RET;
            }

            break;
    }
}
```
CASE WSIO_EVENT_MSI_INTR_MIGR:
    drv_migr = (wsio_intr_migr_msi_t *)event_ptr->arg
    vec_num = drv_migr->msi_vec;
    new_cpu = drv_migr->migr.dest_spu;

    // Disable the specified MSI vector
    if (wsio_msi_disable(hndl, WSIO_MSI_VECTOR_RANGE,
                          vec_num, 1) != WSIO_OK) {
        migrp->ret_val = WSIO_ERROR;
        goto ERR_RET;
    }

    vec_vals.mask = WSIO_MSI_SET_CPU;
    vec_vals.cpu_id = new_cpu;
    type = WSIO_MSI_VECTOR_RANGE | WSIO_MSI_ASSIGN_ALL;
    ret = wsio_msi_assign(hndl, type, &vec_vals, vec_num, 1);
    if (ret == WSIO_OK)
        ret = wsio_msi_enable(hndl, WSIO_MSI_VECTOR_RANGE,
                               vec_num, 1);
    if (ret != WSIO_OK) {
        migrp->ret_val = WSIO_ERROR;
        goto ERR_RET;
    } else
        migrp->ret_val = WSIO_OK;

    break;

CASE WSIO_EVENT_ONLINE_CPU:
    ...
    break;

CASE WSIO_EVENT_SUSPEND:
    CASE WSIO_EVENT_RESUME:

WSIO_EVENT_MSI_INTR_MIGR

All drivers using MSI interrupts must register for the WSIO_EVENT_MSI_INTR_MIGR event. This is enforced by the WSIO interrupt object allocation function wsio_msi_alloc. The intctl command uses this event to reprogram an MSI vector with a specific CPU.

The WSIO subsystem invokes the driver event handler with a pointer to a wsio_generic_event_t structure. The event field is set to WSIO_EVENT_MSI_INTR_MIGR and the arg field points to a wsio_intr_migr_msi_t structure. This structure is defined in /usr/include/sys/wsio.h as follows:

typedef struct wsio_intr_migr_msi {
    wsio_intr_migr_t migr;
    wsio_msi_hndl_t msi_obj;
    int msi_vec;
    uint32_t flags;
} wsio_intr_migr_msi_t;

The msi_vec field is the MSI vector to be reprogrammed. The new_cpu is in the dest_spu field of the wsio_intr_migr_t structure embedded in the wsio_intr_migr_msi_t structure.

The driver event handler disables the interrupt vector by calling wsio_msi_disable. It then reprograms it by calling wsio_msi_assign and re-enables the vector by calling wsio_msi_enable.
After the driver handles this event, it calls the completion callback routine provided by WSIO to indicate the status of the operation in the same manner that it handles the WSIO_EVENT_OFFLINE_CPU event.

**WSIO_EVENT_ONLINE_CPU**

This event flag is optional for drivers using MSI and MSI-X. When WSIO is notified that new CPUs are enabled for interrupt processing, and if the driver has registered for this event, WSIO invokes the driver handler with a `wsio_generic_event_t`. The `event` field is set to WSIO_EVENT_ONLINE_CPU and the `arg` field contains the number of enabled CPUs.

The `wsio_generic_event_t` structure for this event is defined 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 enabled CPUs
} wsio_generic_event_t;
```

After the driver handles the event, it invokes the WSIO-provided completion callback routine to indicate that the event is complete. The completion status is the final argument to the callback routine. If successful, the driver must set the completion status to `WSIO_OK`; any other value is treated as an error. The prototype and calling convention for the completion callback routine are as follows:

```c
typedef int (*generic_complete_callback_t) \n    ((struct isc_table_type *, wsio_event_id_t, void *));
(*wsio_completion_callback)(isc, event_id, ret_val);
```

Where:
- `isc` Pointer to the ISC structure.
- `event_id` Event identification that passed in the `event_id` field of the `wsio_generic_event_t` structure.
- `ret_val` Completion status, usually `WSIO_OK`.

On receipt of a WSIO_EVENT_ONLINE_CPU event, a driver can program new vectors for the new CPUs.

**Interrupt Migration Flow**

In the HP-UX environment, interrupts can be LBIs, TBIs, or MSI and MSI-X. From the WSIO perspective, the events that can lead to interrupt migration operation areas follows:

- Change 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 reassigned to one or more CPUs.

- Change 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.

- Migrate an interrupt from one CPU to another CPU through the `/usr/contrib/bin/intctl` command.

For CPU interrupt state change details, see subsystems such as Processor Sets, **Instant Capacity on Demand** (ICOD), or **Real Time Extension** (RTE).

To migrate an interrupt, these steps are followed:
1. Process LBIs. Table 22-1 shows the LBI actions by layer.

<table>
<thead>
<tr>
<th>WSIO Layer</th>
<th>Driver Layer</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-dependent routines to migrate the interrupt to the new CPU. Later, WSIO invokes the driver event handler to pass the completion event information as WSIO_LBI_INTR_MIGR_COMPLETE.</td>
<td>The driver handles the COMPLETE event and returns WSIO_OK to WSIO.</td>
</tr>
</tbody>
</table>

2. Process TBIs. Table 22-2 shows the TBI actions by layer.

<table>
<thead>
<tr>
<th>WSIO Layer</th>
<th>Driver Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSIO invokes the driver event handler with an event flag as WSIO_EVENT_CPU_OFFLINE.</td>
<td>The driver performs the necessary steps to reprogram the card with the new CPU value. The driver returns WSIO_OK to WSIO.</td>
</tr>
</tbody>
</table>

3. Process MSIs and MSI-Xs. Table 22-3 shows the MSI and MSI-X actions by layer.

<table>
<thead>
<tr>
<th>WSIO Layer</th>
<th>Driver Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSIO invokes the driver event handler with an event flag as WSIO_EVENT_CPU_OFFLINE.</td>
<td>The driver performs the necessary steps to reprogram the card with the new CPU value. The driver returns WSIO_OK to WSIO.</td>
</tr>
</tbody>
</table>

Assumptions and Dependencies

To support interrupt migration, drivers must follow these guidelines:

- The driver must handle false interrupts. The interrupts can occur even if the card is suspended or powered off.
- The driver_attach function must not call the wsio_intr_set_cpu_spec or wsio_intr_set_irq_line functions. These functions can be called from the driver_if_init function.
- The wsio_intr_alloc and wsio_intr_free routines 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 when 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 wsio_intr_activate and wsio_intr_deactivate routines behave in different ways.
  - If drivers using LBI invoke wsio_intr_activate or wsio_intr_deactivate when interrupt migration is in progress, the routines return failure.
  - If interrupt migration is invoked and driver invocation with the above calls is in progress, interrupt migration waits for the calls to complete.
  - If the driver uses a TBI, the driver finishes the migration. Part of the migration process is disabling, then enabling card interrupts. This can involve invocation of the wsio_intr_activate, wsio_intr_deactivate, and wsio_intr_set_cpu_spec
calls. Because the driver initiates all the processing, the driver ensures there are no other threads calling these routines and the interrupt migration thread at the same time. WSIO does not 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 that 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 can be in progress in the system at any given point in time. If an interrupt migration is in progress when a high availability operation like `olrad`, `ioscan` or a DLKM load or unload is initiated, then the operation returns an error.
23 Creating a Software Depot

This chapter describes Software Depot creation techniques. These techniques are necessary to package a driver for distribution. This chapter addresses the following topics:

- Creating a Package
  - “Software Depot Overview”
  - “Step 1: Designing an SD Structure” (page 552)
  - “Step 2: Selecting the Product Directory Structure” (page 552)
  - “Step 3: Writing a PSF” (page 552)
  - “Step 4: Writing Control Scripts” (page 557) (if required)
  - “Step 5: Packaging the Components” (page 564)
  - “Step 6: Registering the Depot” (page 566)
- “Installing a Depot” (page 566)
- “Managing the Depot Software” (page 568)

Software Depot Overview

After a software application is developed, files are taken from the programming environment and integrated for distribution. To ensure ease of installation, maintenance, and uninstallation, software developed on HP-UX is often distributed as Software Depots (SDs). An SD is a directory location on the local or remote host that acts 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 Software Distributor prepares these software files by organizing them into specific products, subproducts, and fileset structures. It also uses special information files that are used to help other commands identify, distribute, and manage the application. See “Step 3: Writing a PSF” (page 552). After it is organized, the software is mastered or copied onto CDs 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:

- 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, refer to a directory depot from its topmost directory. In a CD depot, this directory is the CD mount point.

- Tape Depot
  Software in a tape depot is formatted as a tar archive. Tape depots, such as cartridge tapes, DAT, and 9-track tape are 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 used 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 (a directory depot). Therefore, a host can contain several depots. For example, a designated SD server on the network can contain a word processing software depot, a CAD software depot, and a spreadsheet software depot all on the same server.

The SD provides a powerful set of tools for centralized HP-UX software management. SD commands are included with the HP-UX operating 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 abstract, making it sometimes difficult to visualize objects, processes, and what is 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 managed.

Bundles

A collection of filesets, possibly from several different products, encapsulated for a specific purpose. Bundles can be stored in depots and copied, installed, removed, listed, configured, and verified as single entities. All HP-UX OS software is packaged in bundles. Because bundles are groups of filesets, they are not necessarily supersets of products.

Products

A collection of subproducts (optional) and filesets. The SD commands focus on products but still enable specifying of subproducts and filesets. Different versions of software can be defined for different platforms and operating systems, and also different versions of the software itself. Several different versions can be included on one distribution media or depot.

Subproducts

Subproducts are used to logically group related filesets within a product if the product contains several filesets.

Filesets

Filesets include all the files and control scripts that make up a product. They are the smallest manageable (selectable) SD software objects. Filesets can be part of a single product only, but they can be included in several different HP-UX bundles.

Bundles are designed to provide customers with a single installation unit to install when they purchase software products, such as the ANSI/C compiler. Bundles can be used to provide logical groupings for functionality, such as a web server.

Bundling the products makes it easy to treat several filesets as a single entity using the SD commands. By specifying a bundle, all filesets under that bundle are automatically included in the operation. Performing a single operation on a bundle is the same as performing it individually on each fileset listed in the bundle.

IMPORTANT: Bundles do not eliminate the ability to choose products and filesets to install.

NOTE: A depot is often confused with a software object. Depots cannot be installed or moved like software objects. A depot cannot exist without at least one software object residing in it.

SD Structure Capabilities

Figure 23-1 shows the structure of a Software Depot.
SD is shown as a 4-tier or 3-tier software structure. The bottom tier are filesets to be packaged in the depot. Each fileset must appear in exactly one SD-Product. SD-Filesets can only be installed or removed as a whole entity. Files used for similar purposes must be grouped together in one SD-Fileset. Keep files not used for similar purposes apart. The top tier are products, a collection of filesets or (optional) subproducts and control scripts. A product is the fundamental component in SD, and has the following characteristics:

- 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 used for the same functionality must be in the same SD-Product.

The middle tier are subproducts. If an SD-Product contains several filesets, use subproducts to logically group related filesets.

Bundles are another tier in the software structure shown in Figure 23-1 (page 513). They are collections of filesets, possibly from several different SD-Products, encapsulated for a specific purpose. Bundles provide an alternate way for users to view or select software. The SD structure can be viewed as follows:

Bundle ----> Products ----> Subproducts ----> Filesets

An SD structure can be any of these combinations:

Bundle 1---->Product 1---->SP1---->Fileset 1 and Fileset 2
Bundle 2---->Product 2---->SP4---->Fileset 1 and Fileset 2
Bundle 1---->Product 1---->SP3---->Fileset 2 and Fileset 1
and similarly for SP2 and SP5.

**NOTE:** Product1.Fileset1 != Product2.Fileset1

**Software Objects Nomenclature**

The following terms are used to describe SD components:

- **SD-Bundles**: SD-Bundle names must be a maximum 16 characters with no underscores or white spaces. The following characters are not allowed in SD-Bundle names:

  #
SD-Products

SD-Product names must be a maximum 16 characters, and specify the functionality that they contain. HP recommends that you not use underscores in SD-Product names because of display problems. You can write SD-Product names in mixed case, with significant letters capitalized, and with no white space. Acronyms can be entirely capitalized. Like SD-Fileset names, make SD-Product names descriptive and unique. No two SD-Products can have the same name.

SD-Subproducts

SD-Subproducts names must be a maximum 16 characters, and accurately describe the SD-Subproduct contents. Make the name descriptive enough that a user selecting the software to install or remove can make an informed choice regarding the SD-Subproduct based solely on its name. Do not force the user 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-Filesets names must be entirely capitalized, with a unique name. The maximum length of SD-Fileset names is 16 characters.

SD Package Components

The SD consists of the following components. All except the Agent are installed in the /usr/sbin/ directory.

Packager (swpackage)

The swpackage standalone utility takes a Product Specification File (PSF), a set of control scripts, and the files to be delivered, and packages them into a depot. A depot can be either a directory depot or a tape depot. You can install both depots using the swinstall command.

NOTE: Although the depot is simply a directory structure, you cannot install it using cp or tar.

Controller (Integrated Controller)

The commands used to manage packaged software: swacl, swconfig, swcopy, swinstall, swjob, swlist, swreg, swmodify, 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 GUI is available for swcopy, swinstall, swlist -i, and swremove.

Daemon

The /usr/sbin/swagentd daemon coordinates Controllers and Agents, acting as an intermediary. When a controller command is executed, swagentd is contacted using a Remote Procedure Call (RPC). The swagentd then forks, and the child process issues a call to run the swagent agent.

Agent

The /usr/sbin/swagent Agent performs most of the work done by SD. The agent installs and removes the software. Agents perform source and target activities. A source agent is a swagent process that reads a software
source, such as a depot, while a target agent operates on a target. A target can be a root file system or a depot.

The complete list of commands and a description of their functionality is available in the HP-UX 11i v3 Driver Development Reference.

**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 Product Specification File (PSF) file.

The PSF contains attribute information for all the software objects. Its structure is as follows:

```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
#Subproduct Attributes
Subproduct2
   Subproduct2 Attributes
end #Subproduct2
#Fileset information
Fileset1
   Fileset1 Attributes
end #Fileset1
Fileset1
   Fileset1 Attributes
end # Fileset1
end #Product
```

**NOTE:** The category information is optional. An SD can be packaged without category information. It is used only as a selection mechanism.

Figure 23-2 shows the package information.
SD Objects Attributes Classification and Flow

Following are the SD objects:

Vendor

- **Attributes:**
  - Object definition; for example, tag and title.
  - Information attributes; for example, description.

SD-Category

- **Attributes:**
  - Object definition; for example, tag and title.
  - Information attributes; for example, description.
SD-Bundle Attributes:
- Object definition; for example, tag and title.
- Information attributes; for example, description.
- Control attributes; for example, OS, OS version, release, architecture, and contents.

SD-Product Attributes:
- Object definition; for example, tag and title.
- Information attributes; for example, description.
- Control attributes; for example, OS, OS version, release, and architecture.
- Control scripts; for example, preinstall and postinstall.
- Subproducts and filesets.

SD-Subproduct Attributes:
- Object definition; for example, tag and title.
- Information attributes; for example, description.
- Control attributes; for example, contents.

SD-Fileset Attributes:
- Object definition; for example, tag and title.
- Information attributes; for example, description.
- Control attributes; for example, OS, OS version, release, and architecture.
- Control scripts; for example, preinstall, and postinstall.
- Files. Sources, binaries, and object files to package.

Policies of the PSF Attributes

A PSF is a master file that contains configuration information (attributes). The SD commands operate based on the values set for these attributes. The attributes are classified as either optional or required.

If you must structure and package software for distribution across SD, you must use policies for SD structuring and SD attributes. These policies help customers use the software distribution tools if the software they are dealing with is consistently named and structured.

Vendor Attributes

The following attributes must be included in the PSF file and policies:

- **tag** (Required) The short name for the vendor (distributor) of the software.
- **title** (Required) The long name for the vendor of the software.
- **description** (Required) A file or text string describing the vendor. This value displays only when the user asks to see a description of the software, then selects the vendor description to view.

**NOTE:** The vendor attributes refer to the distributor of the product. For example, if company A develops the product and HP distributes it, the vendor attributes are HP’s.

Category Attributes

Following are the category attributes and policies:

- **tag** (Required) The identifier for the category object. There is a one-to-one relationship between category.tag and category.title attributes as
explained under the `category.title` attribute. Select one of the following tags.

Use the following category tags for HP-UX 11.0, 11i v1, and 11i v2 only.

**HPUXAdditions** | HP-UX SD-Bundle
---|---
**HPUXBaseOS** | Base OS SD-Bundles; reserved for integration.
**LanguagesUI** | Language-specific software SD-Bundles; reserved for integration.
**OpEnvironments** | Operating Environment SD-Bundles; reserved for integration.
**OrderedApps** | Specifies a product that a customer can obtain from HP.
**OrderedHP-UX** | Corresponds to an HP-UX core product that a customer can order.
**TrialUseApps** | Trial version of an available product.
**Patch** | Set automatically if the `is_patch` attribute is set to true; reserved for patches.

Use the following category tags for HP-UX 11i v3 only:

- **CompilerDevelopment**
- **CoreOS**
- **Desktop**
- **DisksFileSystems**
- **Drivers**
- **HighAvailability**
- **InetServices**
- **Internet**
- **Interoperability**
- **Java**
- **Localization**
- **Manuals**
- **Migration**
- **Networking**
- **Obsolescence**
- **Performance**
- **Security**
- **SecurityChoices**
- **SupportTools**
- **SystemManagement**
- **Utilities**

Use the following category tag for HP-UX/OE Integration only:

**OpEnvironments** | HP-UX Operating Environment SD-Bundles; reserved for integration.

Use the following category tags for patches on all OSes:

- **defect_repair**
- **hardware_enablement**
- **enhancement**
- **general_release**
- special_release
- trial_patch
- beta_release
- manual_dependencies
- critical
- panic
- halts_system
- corruption
- memory_leak

**title**

(Required) The one-line, detailed name for the category. There is a one-to-one relationship between `category.tag` and `category.title` attributes. Use the `category.title` values exactly as defined.

Use the following category titles for HP-UX 11.0, 11i v1, and 11i v2 only:

**Ordered Software**
- When `category.tag` is `OrderedApps`

**Trial Use Software**
- When `category.tag` is `TrialUseApps`

**Ordered HP-UX Bundles**
- When `category.tag` is `OrderedHP-UX`

**HP-UX Base Operating System**
- When `category.tag` is `HPUXBaseOS`

**HP-UX Operating Environment Solutions**
- When `category.tag` is `OpEnvironments`

**HP-UX Language Bundles**
- When `category.tag` is `LanguagesUI`

**Additional HP-UX Functionality**
- When `category.tag` is `HPUXAdditions`

Use the following category titles for HP-UX 11i v3 only:

**Compilers and Development Tools**
- When `category.tag` is `CompilerDevelopment`

**Core HP-UX Functionality**
- When `category.tag` is `CoreOS`

**Desktop Environments**
- When `category.tag` is `Desktop`

**File Systems and Volume Management**
- When `category.tag` is `DisksFileSystems`

**I/O Drivers**
- When `category.tag` is `Drivers`

**High Availability**
- When `category.tag` is `HighAvailability`

**Internet Services**
- When `category.tag` is `InetServices`

**Internet Software Tools**
- When `category.tag` is `Internet`

**Computing Interoperability Tools**
- When `category.tag` is `Interoperability`

**Java Tools and Utilities**
- When `category.tag` is `Java`

**Localization**
- When `category.tag` is `Localization`

**HP-UX Manual Pages**
- When `category.tag` is `Manuals`

**Migration to HP-UX Tools**
- When `category.tag` is `Migration`
Networking Infrastructure When category.tag is Networking.
Product Obsolescence When category.tag is Obsolescence.
Performance Tools When category.tag is Performance.
Security Tools When category.tag is Security.
Security Level Choices When category.tag is SecurityChoices.
Diagnostic and Support Tools When category.tag is SupportTools.
System Management Tools When category.tag is SystemManagement.
Miscellaneous Utilities When category.tag is Utilities.
Use the following category titles for HP-UX/OE Integration only:
HP-UX Operating Environment Solutions When category.tag is OpEnvironments.
Use the following category.titles for patches. They are controlled by the patch creation tools for all OSes:
" " When category.tag is beta_release, special_release, or trial_patch.
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 an 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 a manual review of dependencies When category.tag is manual_dependencies.
Fix a system panic When category.tag is panic.
description (Optional) A file or text string providing a more detailed description of the category. Use only the text strings listed in the policy because the category.description is global. That is, a particular category.description applies to all objects in the depot or on the media with the corresponding category.tag.
Use the following category descriptions for HP-UX 11.0, 11i v1, and 11i v2 only:
Ordered Software Applications When category.tag is OrderedApps.
Trial Use Software When category.tag is TrialUseApps.
Ordered HP-UX Bundles When category.tag is OrderedHP-UX.
HP-UX Base Operating System When category.tag is HPUXBaseOS.
HP-UX Operating Environment Solutions When category.tag is OpEnvironments.
HP-UX Language Bundles When category.tag is LanguagesUI.
Additional HP-UX Functionality When category.tag is HPUXAdditions.
Use the following category descriptions for HP-UX 11i v3 only:
Compilers and Development Tools When category.tag is CompilerDevelopment.
Core HP-UX Functionality When category.tag is CoreOS.
Desktop Environments When category.tag is Desktop.
I/O Drivers When category.tag is Drivers.
High Availability When category.tag is HighAvailability.
Internet Services When category.tag is InetServices.
Internet Software Tools When category.tag is Internet.
Computing Interoperability Tools When category.tag is Interoperability.
Java Tools and Utilities When category.tag is Java.
Localization When category.tag is Localization.
HP-UX Manual Pages When category.tag is Manuals.
Migration to HP-UX Tools When category.tag is Migration.
Networking Infrastructure When category.tag is Networking.
Product Obsolescence When category.tag is Obsolescence.
Performance Tools When category.tag is Performance.
Security Tools When category.tag is Security.
Security Level Choices When category.tag is SecurityChoices.
Diagnostic and Support Tools When category.tag is SupportTools.
System Management Tools When category.tag is SystemManagement.
Miscellaneous Utilities When category.tag is Utilities.
Use the following category descriptions for HP-UX/OE integration only:
HP-UX Operating Environment Solutions

Use the following category.descriptions for patches. They are controlled by the patch creation tools for all OSes:

- Fix corruption
- Fix a critical defect
- Provide defect repair
- Provide an enhancement
- Fix a hang or abort
- Provide new hardware support
- General release patch
- Fix a memory leak
- Requires a manual review of dependencies
- Fix a system panic

SD-Bundle Attributes

Following are bundle attributes and policies:

**tag**

(Required) 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 must be unique and descriptive.

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**

(Required) The one-line, detailed name for the SD-Bundle (appears on the Software Selection window and in the swlist one-liner). 80 characters maximum. Use this attribute to expand upon the sixteen character name of the SD-Bundle.

This attribute is used often, so select it carefully to clearly and concisely tell the user what the SD-Bundle is.

**description**

(Required) A file or text string (limited to 8 K) describing the SD-Bundle, displayed only when the user asks to see a description of the software and selects this description for view. The free-form ASCII style of this attribute allows for detailed description of the software.

For maximum readability, limit each line to 72 characters or less. Do not use tabs in this file or text string.
**revision**  
(Required) The revision (release) number of the SD-Bundle.  
The revision number of an SD-Bundle form is `major:minor:release`  
(path).

**architecture**  
(Required) The target systems on which this SD-Bundle is to be installed and run. The attribute is structured as follows:

`os name_os release_os bits`

Where:

- **os name**: HP-UX
- **os release**: The minimum OS revision necessary to run the software.  
  For example:
  - If the SD-Bundle contains bits compiled on and can run on HP-UX 11i v1.6 and later, the `os release` must be B.11.22.
  - If the SD-Bundle contains bits compiled on and can run on HP-UX 11i v2 and later, the `os release` must be B.11.23.
  - If the SD-Bundle contains bits compiled on and can run on HP-UX 11i v3 and later, the `os release` must be B.11.31.
  - If the SD-Bundle can run only on HP-UX 11i v3 and later, the `os release` must be B.11.31.

- **os bits**: Specifies the OS bit configuration on which the SD-Bundle will run. The `os bits` can reflect either 11.22/11.23 native software or pre-HP-UX 11i v1.6 PA-RISC native software.

For SD-Bundles that have Integrity native bits or HP-UX 11i v1.6, HP-UX 11i v2, or HP-UX 11i v3 PA-RISC native bits, or both, use the following `bundle.architecture` values:

- **HP-UX_B.11.31_IA**: SD-Bundles that can be installed only on HP-UX 11i v3 Integrity machine architectures.
- **HP-UX_B.11.31_IA/PA**: SD-Bundles that can be installed on both HP-UX 11i v3 Integrity and PA-RISC machine architectures.
- **HP-UX_B.11.31_PA**: SD-Bundles that can be installed only on HP-UX 11i v3 PA-RISC machine architectures.
- **HP-UX_B.11.23_IA/PA**: SD-Bundles that can be installed on both HP-UX 11i v2, on both Integrity and PA-RISC machine architectures.
- **HP-UX_B.11.23_PA**: SD-Bundles that can be installed only on HP-UX 11i v2 PA-RISC machine architectures.
- **HP-UX_B.11.23_IA**: SD-Bundles that can be installed only on HP-UX 11i v2 Integrity machine architectures.
- **HP-UX_B.11.22_IA**: SD-Bundles that can be installed only on HP-UX 11i v1.6 and HP-UX 11i v2 Integrity machine architectures.
- **HP-UX_B.11.22_IA/PA**: SD-Bundles that can be installed on both HP-UX 11i v1.6 and HP-UX 11i v2 Integrity and PA-RISC machine architectures.
- **HP-UX_B.11.22_PA**: SD-Bundles that can be installed on both HP-UX 11i v1.6 and HP-UX 11i v2 PA-RISC machine architectures.
For SD-Bundles that have only PA-RISC bits from HP-UX 11i v1 that are being delivered to HP-UX 11i v3 for installation on Integrity or PA-RISC, or both machine architectures and run on Integrity machines using the ARIES emulator, use the following `bundle.architecture` values:

- **HP-UX_B.11.22_32/64**
  32- and 64-bit PA-RISC native software built for HP-UX 11i v1 and packaged for installation on HP-UX 11i v1.6, HP-UX 11i v2, or HP-UX 11i v3 Integrity or PA-RISC machine architectures.

- **HP-UX_B.11.22_64**
  64-bit PA-native software built for HP-UX 11i v1 and packaged for installation on HP-UX 11i v1.6, HP-UX 11i v2, or HP-UX 11i v3 Integrity or PA-RISC machine architectures.

- **HP-UX_B.11.23_32/64**
  32- and 64-bit PA-RISC native software built for HP-UX 11i v1 and packaged for installation on HP-UX 11i v2 or HP-UX 11i v3 Integrity or PA-RISC machine architectures.

- **HP-UX_B.11.23_64**
  64-bit PA-RISC native software built for HP-UX 11i v1 and packaged for installation on HP-UX 11i v2 or HP-UX 11i v3 Integrity or PA-RISC machine architectures.

- **HP-UX_B.11.31_32/64**
  32- and 64 bit PA-RISC native software built for HP-UX 11i v1 and packaged for installation on HP-UX 11i v3 Integrity or PA-RISC machine architectures.

- **HP-UX_B.11.31_64**
  64-bit PA-RISC native software built for HP-UX 11i v1 and packaged for installation on HP-UX 11i v3 Integrity or PA-RISC machine architectures.

Regardless of the architecture defined for a software object, you must define the `machine_type`, `os_name`, and `os_release` compatibility attributes.

---

**NOTE:** An SD-Bundle with an Integrity or PA-RISC architecture can have SD-Products or SD-Filesets with Integrity/PA-RISC and 32- and 64-bit architectures. However, SD-Bundles with 32- and 64-bit architectures cannot have Integrity/PA-RISC, Integrity, or PA-RISC SD-Products or SD-Filesets.

**NOTE:** Core HP-UX SD-Bundles that have a `bundle.category_tag` attribute of `HPUXBaseOS`, `OpEnvironments`, or `LanguagesUI` must have a `bundle.architecture` attribute of `HP-UX_B.11.31_IA/PA`, `HP-UX_B.11.31_IA`, or `HP-UX_B.11.31_PA`.

**os_name**
(Required) The OS on which the SD-Bundle will run.
Use HP-UX as the value.

**os_release**
(Required) OS releases on which the SD-Bundle will run.
- `?..11.31` HP-UX 11i v3 only
- `B.11.31` HP-UX 11i v3 only
- `?..11.3*` HP-UX 11i v3 and any 11.3x release
- `?11.[23]?` HP-UX 11i v1.6, HP-UX 11i v2, HP-UX 11i v3, and any HP-UX 11i v2 and HP-UX 11i v3 release
<table>
<thead>
<tr>
<th>Version</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>?.11. [23] [123]</td>
<td>HP-UX 11i v1.6 and any, HP-UX 11i v2, and any HP-UX 11i v3 release</td>
</tr>
<tr>
<td>?.11.11</td>
<td>?.11.2</td>
</tr>
<tr>
<td>?.11.*</td>
<td>Any HP-UX 11i release (HP-UX 11i v1.6 and earlier PA-RISC native bits installed on Integrity and run through the ARIES translator)</td>
</tr>
</tbody>
</table>

**NOTE:** You are responsible for accurate packaging. Contact your development and marketing teams to determine which OS releases are supported with the product.

**NOTE:** Core HP-UX SD-Bundles that have the bundle.category_tag attributes HPUXBaseOS, OpEnvironments, or LanguagesUI must have a bundle.os_release attribute of B.11.31.

**os_version**
(Optional) OS versions on which the SD-Bundle will run. This attribute is not used. If it is set, it must be set to "*".

**machine_type**
(Required) Machine types on which the SD-Bundle will run. Use the "*.*" construct for backward compatibility.

- *: SD-Bundle installable on both Integrity and PA-RISC systems.
- ia64*: SD-Bundles installable on Integrity only.
- ia64*server*: SD-Bundle installable on Integrity only.
- ia64*workstation*: SD-Bundles installable on Integrity workstations only.
- 9000/*/: SD-Bundles installable on any PA-RISC system.
- 9000/8*: SD-Bundles installable on PA-RISC systems only.
- 9000/7*: SD-Bundles installable on PA-RISC workstations only.
- ia64*server*|9000/8*: SD-Bundles installable on either Integrity or PA-RISC systems only.
- i64*workstation*|9000/7*: SD-Bundles installable on either Integrity or PA-RISC workstation only.

**category_tag**
(Required if category object was defined in the PSF). Visible only from swlist, when users specify a software specification to select a class of SD-Bundles from the command line. Use one of the following values.

**NOTE:** There is a one-to-one relationship between bundle.category_tag and bundle.category_title attributes.

- OrderedApps: Specifies a product that a customer can obtain from HP.
- TrialUseApps: Trial version of an available product.
HPUXAdditions Additional HP-UX functionality.

vendor_tag (Required) The short name for the software vendor (distributor). This must match the vendor.tag attribute.

contents (Required) Describes the software contained in the SD-Bundle. Must be a list of SD-Filesets with their software specifications, but can also include a list of SD-Products with their software specifications. Each software specification must explicitly list the revision, architecture, and vendor. The revision listed in any software specification is always the product revision. The software specification must not include the SD-Fileset level architecture (fa=) unless the SD-Bundle contains a subset of multistream SD-Fileset pairs, if applicable. For example, a typical bundle.contents is as follows:

Prod1.FILESET1,r=B.11.23,a=HP-UX_B.11.23_IA/PA,v=HP

SD-Product Attributes

Following are the product attributes and policies:

tag (Required) The identifier or name for the SD-Product, which appears on the Software Selection window as the basic name for the SD-Product. 16 characters maximum.

SD-Product names must be 16 or fewer characters. Succinctly specify the functionality the software contains.

**TIP:** HP recommends 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 (Required) The one-line, detailed name for the SD-Product, which appears on the Software Selection window and in the swlist output. 80 characters maximum. Use this attribute to expand upon the 16-character tag of the SD-Product.

This attribute is used often, so select it carefully to clearly and concisely describe the SD-Product.

description (Required) A file or text string (limited to 8K) describing the SD-Product, displayed only when the user asks to see a description of the software and selects this description for view. The free-form ASCII value of this attribute allows for detailed description of the software.

For maximum readability, limit each line to 72 characters or less. Do not use tabs in this string.

revision (Required) The revision (release) number of the SD-Product.

Because each product has its own revision cycle, there is not a definitive policy for specifying product revision. Manage your revision numbers in a way that works best for your product.

The suggested revision number of a SD-Product is major:minor:release (path).
NOTE: If the product is part of a bundle, the revision attribute has the same value as that of the Bundle.

architecture
(Required) The target systems on which this SD-Product will run. Summarizes the supported hardware and operating systems.
The attribute is structured as follows:

\[ \text{os name}_\text{os release}_\text{os bits} \]

Where:

- \( \text{os name} \): HP-UX
- \( \text{os release} \): The minimum OS revision necessary to run the software.
- \( \text{os bits} \): Specifies the OS bit configuration on which the SD-Product will run. The \( \text{os bits} \) can reflect either HP-UX 11i v1.6 or HP-UX 11i v2 Integrity/PA-RISC native software or pre-HP-UX 11i v1 PA-RISC native software.

\[ \begin{align*}
    \text{HP-UX}_B.11.23_\text{PA} & \quad \text{SD-Products that can be installed on HP-UX 11i v2 PA-RISC only.} \\
    \text{HP-UX}_B.11.22_\text{IA} & \quad \text{SD-Products that can be installed on HP-UX 11i v1.6 and HP-UX 11i v2 Integrity only.} \\
    \text{HP-UX}_B.11.22_\text{IA/PA} & \quad \text{SD-Products that can be installed on both HP-UX 11i v1.6 and HP-UX 11i v2 Integrity and PA-RISC systems.} \\
    \text{HP-UX}_B.11.23_\text{IA/PA} & \quad \text{SD-Products that can be installed on HP-UX 11i v2 only, on both Integrity and PA-RISC systems.}
\end{align*} \]

NOTE: If the SD-Product can run only on HP-UX 11i v2 and later, the \( \text{os release} \) portion must be \( B.11.23 \).

os name
(Required) OS releases on which the SD-Product will run. Use HP-UX as the value.

NOTE: If the product is part of a bundle, the \( \text{os name} \) attribute has the same value as that of the Bundle.

os release
(Required) OS releases on which the SD-Product will run.

\[ \begin{align*}
    \text{?}.11.23 & \quad \text{HP-UX 11i v2 only} \\
    B.11.23 & \quad \text{HP-UX 11i v2 only} \\
    \text{?}.11.2* & \quad \text{HP-UX 11i v1.5, HP-UX 11i v1.6, and any HP-UX 11i v2 release} \\
    \text{?}.11.2[023] & \quad \text{HP-UX 11i v1.5, HP-UX 11i v1.6, and HP-UX 11i v2 only}
\end{align*} \]
**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**
(Optional) OS versions on which the SD-Product will run. This attribute is not used.

* Allow all versions.

**machine_type**
(Required) Systems on which the SD-Bundle will run.

**NOTE:** The `*. *` construct is for backward compatibility.

<table>
<thead>
<tr>
<th>Constructs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>SD-Products installable on both Integrity and PA-RISC systems.</td>
</tr>
<tr>
<td>ia64*</td>
<td>SD-Products installable on Integrity only.</td>
</tr>
<tr>
<td>ia64<em>server</em></td>
<td>SD-Products installable on Integrity only.</td>
</tr>
<tr>
<td>ia64<em>workstation</em></td>
<td>SD-Products installable on Integrity workstations only.</td>
</tr>
<tr>
<td>9000/*</td>
<td>SD-Products installable on any PA-RISC system.</td>
</tr>
<tr>
<td>9000/8*</td>
<td>SD-Products installable on PA-RISC systems only.</td>
</tr>
<tr>
<td>9000/7*</td>
<td>SD-Products installable on PA-RISC workstations only.</td>
</tr>
<tr>
<td>ia64<em>server</em></td>
<td>9000/8*</td>
</tr>
<tr>
<td>i64<em>workstation</em></td>
<td>9000/7*</td>
</tr>
</tbody>
</table>

**directory**
(Required) The default absolute path name of the directory which is the root of the file system tree under which the SD-Product will be installed. Can be remapped by the user if the `is_locatable` attribute is true.

Use a standard location (`/opt/myprod`).

**NOTE:** Do not include a trailing slash like `/opt/mywrongslash/`.

**is_locatable**
(Required) If set to true, users can install the SD-Product into an alternate product directory.

FALSE  SD-Product must be installed into the path specified by `product.directory`.

TRUE  SD-Product can be installed to an alternate location.

**NOTE:** This attribute must be explicitly set to either TRUE or FALSE. The default is TRUE.

**category_tag**
(Required) The short name for the category of the software. This attribute associates the SD-Product with the standard category object.

This tag must match the proper `category.tag` attribute.
vendor_tag  (Required) The short name for the software vendor. HP (or the company providing ISV software). This tag must match the vendor.tag attribute.

is_patch  (Required) A boolean flag that identifies a software object as a patch. The default value is FALSE. When set to TRUE, a built-in category_tag attribute of patch value is automatically included with the SD-Product definition.

Only use the is_patch attribute when packaging patches.

preinstall  (Optional) Path name for the install preload SD-Product control script. The script is run by swinstall during the execution phase before loading the software files.

unpreinstall  (Optional) Path name for the install undoing preload SD-Product control script. The script is run during the swinstall load phase if it is initiated. An unpreinstall script must undo any operation taken by the preinstall script.

The unpreinstall scripts must be carefully created and extensively tested.

postinstall  (Optional) Path name for the install postload SD-Product control script. The script is run by swinstall during the Load phase after loading the software files.

unpostinstall  (Optional) Path name for the install undoing postload SD-Product control script. The script is run during the swinstall load phase if recovery is initiated. An unpostinstall script must undo any operation taken by the postinstall script.

configure  (Optional) Path name 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 with host-specific information. Configure scripts are run by swinstall for all SD-Products after they complete the load phase. The swconfig command can also be used to rerun configure scripts that failed during a normal install.

verify  (Optional) Path name for the verify integrity SD-Product control script. This script checks for the correctness of the product or fileset installation and configuration. It is run by the swverify command.

unconfigure  (Optional) Path name 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  (Optional) Path name for the remove preremove SD-Product control script. The script is run by swremove during the remove phase just before removing files.

postremove  (Optional) Path name for the remove postremove SD-Product control script. The script is run by swremove during the remove phase immediately after the files have been removed.

request  (Optional) Path name for the request interactive SD-Product 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.

Because request scripts are user interactive, they cannot be used in cold install or unattended installation situations. HP recommends that you not use them.
SD-Subproduct Attributes

Following are the subproduct attributes and policies:

tag

(Optional) The identifier for the SD-Subproduct that appears in the Software Selection window as the basic name for the SD-Subproduct. 16 characters maximum.

A one-to-one relationship exists between the subproduct tag and subproduct title attributes. If a non-standard SD-Subproduct name was created for the SD-Product, the subproduct tag attribute must match that name.

SD-Subproducts and SD-Filesets within the same SD-Product cannot have the same tag.

Standard subproduct tag names are as follows:

Manuals Manpages and product documentation.
ManualsByLang Localized manual pages and product documentation.
Help Online help.
HelpByLang Localized online help.
MessagesByLang Localized messages.
Demonstration Demonstration of the product.
Development Software development SD-Filesets.
ReleaseNotes Separate SD-Filesets for release notes.

title

(Optional) The one-line, more detailed name for the SD-Subproduct that appears in the Software Selection window. 80 characters maximum. Expand 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.

Standard subproduct titles are as follows:

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.

description

(Optional) A file or text string (maximum 8 K) describing the SD-Subproduct, displayed only when the user requests a description of the software and selects...
SD-Fileset Attributes

Following are the fileset attributes and policies:

tag
(Required) The identifier for the SD-Fileset that appears in the Software Selection window as the name for the SD-Fileset. 14 characters maximum. SD-Subproducts and SD-Filesets within the same SD-Product cannot 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 must be in all capital letters.

title
(Required) The one-line, detailed name for the SD-Fileset that appears in the Software Selection window. 80 characters maximum. Expand the 14-character SD-Fileset tag. This attribute is viewed often. Choose it carefully to clearly and concisely describe the SD-Fileset. Often, the first position also includes an abbreviation of the SD-Product.

description
(Required) A file or text string (maximum 8 K) describing the SD-Fileset, displayed only when the user asks to see a description of the software and selects this description. The free-form ASCII style of this attribute enables detailed description of the software. For maximum readability, limit each line to 72 characters or less.

revision
(Required) The SD-Fileset revision or version number. Each SD-Fileset must have the same revision as the SD-Product containing it.

architecture
(Required) The target systems on which this SD-Fileset will run. Summarizes the supported hardware and operating systems. This attribute is structured as follows:

```
    os name_os release_os bits
```

Where:

- **os name**
  - HP-UX.

- **os release**
  - The minimum OS revision necessary to run the software.
  - If the SD-Product can run only on HP-UX 11i v2 and later, the **os release** must be B.11.23.

- **os bits**
  - Specifies the OS bit configuration on which the SD-Product will run. The **os bits** can
reflect either HP-UX 11i v1.6/HP-UX 11i v2 Integrity/PA-RISC native software or HP-UX 11i v1.6 PA-native software.

<table>
<thead>
<tr>
<th>SD-Filesets</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HP-UX_B.11.23_PA</td>
<td>SD-Filesets that can be installed on HP-UX 11i v2 PA-RISC only.</td>
</tr>
<tr>
<td>HP-UX_B.11.22_IA</td>
<td>SD-Filesets that can be installed on HP-UX 11i v1.6 and HP-UX 11i v2 Integrity only.</td>
</tr>
<tr>
<td>HP-UX_B.11.22_IA/PA</td>
<td>SD-Filesets that can be installed on both HP-UX 11i v1.6 and HP-UX 11i v2 Integrity and PA-RISC systems.</td>
</tr>
<tr>
<td>HP-UX_B.11.23_IA/PA</td>
<td>SD-Filesets that can be installed on HP-UX 11i v2 only, on both Integrity and PA-RISC systems.</td>
</tr>
</tbody>
</table>

**NOTE:** Because the fileset is part of a product or a bundle, the architecture attribute must have the same value as that of the product or bundle.

### os_name

(Required) OS releases on which the SD-Fileset will run.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HP-UX</td>
<td></td>
</tr>
</tbody>
</table>

**NOTE:** Because the fileset is part of a product or a bundle, the os_name attribute must have the same value as the product or bundle.

### os_release

(Required) OS releases on which the SD-Fileset is to be installed. Values must be one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.11.23</td>
<td>HP-UX 11i v2 only</td>
</tr>
<tr>
<td>B.11.23</td>
<td>HP-UX 11i v2 only</td>
</tr>
<tr>
<td>.11.2*</td>
<td>HP-UX 11i v1.5, HP-UX 11i v1.6, and any HP-UX 11i v2 release</td>
</tr>
<tr>
<td>.11.2 [023]</td>
<td>HP-UX 11i v1.5, HP-UX 11i v1.6, and HP-UX 11i v2 only</td>
</tr>
</tbody>
</table>

**NOTE:** Because the fileset is part of a product or a bundle, the os_release attribute must have the same value as that of the product or bundle.

### os_version

(Optional) OS versions on which the SD-Fileset will run. This attribute is not used.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Allow all versions</td>
</tr>
</tbody>
</table>

**NOTE:** Because the fileset is part of a product or a bundle, the os_version attribute must have the same value as that of the product or bundle.

### machine_type

(Required) System types on which the SD-Fileset will run.

**NOTE:** The *.* construct is for backward compatibility.

Value must be one of the following:
* SD-Filesets installable on both Integrity and PA-RISC systems.

ia64* SD-Filesets installable on Integrity only.

ia64*server* SD-Filesets installable on Integrity only.

ia64*workstation* SD-Filesets installable on Integrity workstations only.

9000/* SD-Filesets installable on any PA-RISC system.

9000/8* SD-Filesets installable on PA-RISC systems only.

9000/7* SD-Filesets installable on PA-RISC workstations only.

ia64*server*|9000/8* SD-Filesets installable on either Integrity or PA-RISC systems only.

i64*workstation*|9000/7* SD-Filesets installable on either Integrity or PA-RISC workstation only.

**category_tag** (Optional) The short name for the category of the software. This attribute associates the SD-Fileset with the standard category object. This attribute must match the proper category.tag attribute.

**is_patch** (Required for packaging patches) A boolean flag that identifies a software object as a patch. The default value is false. When set to true, a built-in category_tag attribute of valuepatch is automatically included with the SD-Product definition.

Use the is_patch attribute only when packaging patches.

**is_kernel** (Optional) Defined if the SD-Fileset contains OS kernel files. If it is set to TRUE, additional policy must be handled in the control scripts. Possible values are as follows:

FALSE The SD-Fileset contains kernel files.

TRUE The SD-Fileset does not contain kernel files.

**is_reboot** (Optional) Defined if the SD-Fileset requires a system reboot after installation. Possible values are as follows:

FALSE The SD-Fileset does not require a system reboot after installation.

TRUE The SD-Fileset requires a system reboot after installation.

**is_drd_safe** (Required for all HP-UX 11i v3 filesets. Optional for HP-UX 11i v2 and previous releases.) Indicates that the fileset can be managed (installed, removed, verified, and unconfigured) in a DRD-chrooted shell without affecting the running system. The default is false. A fileset can only be considered DRD-safe if both the fileset and product control scripts are DRD-safe. If a product-level control script is not DRD-safe, all of the filesets for that product must be listed as unsafe (the attribute for each fileset in the product is set to false). If any fileset is selected for management (install, remove, verify, or unconfigure) and that fileset does not have is_drd_safe set to true, that fileset is excluded from the list.
of software to be managed in a DRD-chrooted shell. This includes any filesets selected due to dependencies.

Possible values are as follows:

**FALSE**  The fileset cannot be managed safely in a DRD-chrooted shell.

**TRUE**  The fileset can be managed safely in a DRD-chrooted shell.

All filesets for HP-UX 11i v3 and beyond must explicitly set this. All filesets in HP-UX 11i v3 and higher that are included in OE bundles are required to be DRD-safe and set this attribute to **TRUE**. Any filesets that set this attribute to **FALSE** are flagged for review by the DRD team.

**dynamic_module**  (Required) Defined if the SD-Fileset includes a Dynamically Loadable Kernel Module (DLKM).

One or more DLKMs can be listed after the **dynamic_module** keyword. The dynamic module keyword can be repeated in the section of the PSF for a single SD-Fileset. For example, each of the following values is acceptable and they can both be included in the PSF for a single SD-Fileset.

```
dynamic_module modname_1

dynamic_module modname_2 modname_3
```

For each DLKM `modname` in the **dynamic_module** list, the file `/usr/conf/mod/modname` must be packaged. If the `modprep` script is supplied (see `modprep(9E)`), it must be delivered to `/usr/conf/mod`.

**NOTE:**  If the DLKM is made static by a call to `control_util mod_systemfile` in the postinstall script, the `is_kernel` and `is_reboot` attributes must be set to `true`.

**preinstall**  (Optional) Path name for the install preload SD-Fileset control script. The script is run by `swinstall` during the execution phase before loading the software files.

**unpreinstall**  (Optional) Path name for undoing the install preload SD-Fileset control script. The script is run during the `swinstall` load phase if initiated. An unpreinstall script must undo any operation taken by the preinstall script.

Must be carefully created and extensively tested.

**postinstall**  (Optional) Path name for the install postload SD-Fileset control script. The script is run by `swinstall` during the load phase after loading the software files.

**unpostinstall**  (Optional) Path name for undoing the install postload SD-Fileset control script. The script is run during the `swinstall` load phase if recovery is initiated. An unpreinstall script must undo any operation taken by the postinstall script.

Must be carefully created and extensively tested.

**configure**  (Optional) Path name 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 with host-specific information. Configure scripts are run by `swinstall` for all SD-Filesets after they complete the load phase. The `swconfig` command can also be used to rerun configure scripts that failed during a normal install.
verify (Optional) Path name for the verify integrity SD-Fileset control script. The verify script is run by the `swverify` command to check for the correctness of the product or fileset installation and configuration.

unconfigure (Optional) Path name for undoing the configure 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 (Optional) Path name for the remove preremove SD-Fileset control script. The script is run by `swremove` during the remove phase just before removing files.

postremove (Optional) Path name for the remove postremove SD-Fileset control script. The script is run by `swremove` during the remove phase just after the files are removed.

request (Optional) Path name 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.

Because request scripts are user-interactive, they cannot be used in cold installs or unattended installation situations. HP recommends that you not request scripts.

File Definitions

The files contained in each SD-Fileset must be specified within the PSF. A file can be in only one SD-Fileset. It cannot reside in multiple SD-Filesets.

**NOTE:** A symlink command can cause confusion during installation. For example, a link `/usr/bin/x` is a symbolic link to `/usr/bin/y`. If the product wants to install `/usr/bin/x` as a file or directory, SD installs the file or directory at `/usr/bin/y` and leaves `/usr/bin/x` as a symbolic link to `/usr/bin/y`. SD follows the links down and does not replace them.

file permissions (Optional) 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 an SD-Fileset replace previous definitions. In the default condition, destination files receive permissions from their respective source files.

```
file_permissions [\-m mode (octal) | \-u umask]
[\-o [owner[,]]\[uid\]] [\-g [group[,]]\[gid\]]
```

directory (Optional) Specifies a source directory in which subsequently listed filenames are located (for this SD-Fileset only). The source directory can be an absolute or relative pathname. Relative path names are interpreted relative to the current working directory in which `swpackage` is invoked.

```
directory source_dir [= destination_dir]
```

file (Required) Specifies the files to be packaged into a SD-Fileset. When the directory keyword is used, path names are relative to the directory specified. Otherwise, path names 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 the following format for explicit naming:
```plaintext
file [-m mode (octal)] [-o [owner[,]][uid]]
     [-g [group[,]][gid]] source [destination]
```
Use the following format for implicit (recursive) naming:
```plaintext
file *
```
Use the following format for explicit naming of directories:
```plaintext
file [-m mode (octal)] [-o [owner[,]][uid]] [-g
     [group[,]][gid]]
     source_dir [destination_dir]
```
Use the following format for combining file definitions. For example,
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, enter the following:
```plaintext
directory/src/develop/ddk/driver/qlispdrv=/opt/ddk/sampldrvs/qlisp/file
```
To implement the previous example giving the `qlisp` directory and
group all of the files in * permission 755, owner root, and group users,
enter the following:
```plaintext
file_permissions -m 755 -o root -g users
directory/src/develop/ddk/driver/qlispdrv=/opt/ddk/sampldrvs/qlisp/file
```
To explicitly provide the permission set 555, owner root, and group
root to the `/opt/ddk/ SD-Product directory, enter the following:
```plaintext
file_permissions -m 555 -o root -g root
file /src/develop/ddk /opt/ddk
```

Control Scripts Overview

You can optionally use control scripts to manage the SD-Product more efficiently. This section
describes in detail all the control scripts you can use during SD packaging. Each script must
fulfill the requirements of the SD-UX software distribution tools and commands, and of the
specific fileset.

The control scripts are separate shell scripts included as attributes in the PSF for specific purposes
and used with SD commands and tools.

The following control scripts can be used in SD-UX. A product might not use all control scripts.
Also, each fileset in a product can have its own control scripts.

```plaintext
checkinstall Run within a swinstall session.
preinstall Run within a swinstall session.
postinstall Run within a swinstall session.
configure Run within a swinstall session and from swconfig.
verify Used with swverify.
fix Run by swverify to correct and report problems on installed software.
checkremove Run within a swremove session and from svconfig.
unconfigure Run within a swremove session and from svconfig; the reverse of
configure.
preremove Run within a swremove session.
postremove Run within a swremove session.
request Run from swinstall and swremove sessions; user- interactive script.
```
Control Script Levels

The control scripts can be used at two SD-UX levels:

Product level Used with SD-UX, associated with both filesets and 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 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 specific fileset.

Any actions included in a specific control script for every fileset in a product must be put in the appropriate product-level control script instead.

Fileset level The scripts for a specific 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 requires that the target system be rebooted as part of software installation and configuration. The fileset.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 reboots 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.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 must be correctly installed for proper install-time operation. A fileset specifies a prerequisite fileset only if it needs the prerequisite fileset configured before the fileset configure script is run.

Corequisite Fileset

A corequisite fileset must be correctly installed for proper run-time operation.

Control Script Format

A control script file must be a shell script (as opposed to a binary), capable of being interpreted by the Posix.2 shell /sbin/sh. Korn shell (formerly /bin/ksh) syntax works with the Posix.2 shell. However, the script must be run in the Posix shell (the first line of any script must be #!/sbin/sh and not #!/bin/ksh or #!/usr/bin/ksh). Scripts written in csh are not supported.

The file must have a simple header. The first line must indicate the interpreter shell by using the #! convention. Also, include comment lines in the header, which state the product and fileset to which the script belongs, the name of the script, the revision string as required by the what command, and a simple copyright statement. For example:

```bash
#!/sbin/sh
#########
# Product: SD Package Components 537
```
Execution Environment

This section describes 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, and postinstall scripts for a specific fileset are downloaded to the /var/tmp/CATALOG_DIR/catalog/PRODUCT/FILESET/control_script temporary directory from which they are invoked.

The form of the CATALOG_DIR is AAAapid, where pid is the swinstall process ID number. These files are delivered to that location from the depot immediately after product selection completes, 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 appear in the Installed Product Database (IPD). They are delivered to that location from the depot as part of the installation of the other files in the fileset as follows:
/var/adm/sw/products/PRODUCT/FILESET/control_script

The location of the IPD is relative to the root directory under which the software is installed. If the installation is to an alternate root (for example, /mnt/disk2), the IPD for that software is located in the following file:
/mnt/disk2/var/adm/sw/products/PRODUCT/FILESET

Environment Variables

The control scripts are invoked as the superuser, with an effective uid of 0. All of the control scripts are invoked without arguments. Control scripts learn by running commands from within the scripts, by calling other utilities, and by environment variable values. The following environment variables pass to each control script. You can test them for a particular value or use them to construct another environment variable, but you must never set or alter these values within a control script. The following are the environment variables:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SW_PATH</td>
<td>A path to commands that 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.</td>
</tr>
<tr>
<td>PATH</td>
<td>The path to commands that 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.</td>
</tr>
<tr>
<td>SW_ROOT_DIRECTORY</td>
<td>The path to the root directory in which the software is installed. This is typically /, but is different in the case of an alternate root</td>
</tr>
</tbody>
</table>
install. This variable is always / for configure, swverify, and unconfigure. Prefix this value to the path of any installed file, but not to commands to be run.

When prefixing this variable to a path, do not use a / as the leading character in the path because it then expands to / /. The value is set on the command line as a suffix to the target system name, which is distinguished by the at symbol (@),

swinstall \* @ hpmy.sys:/mnt/disk30...

The installation to an alternate root requires that you specify the -r option when you run swinstall in either interactive and noninteractive mode.

NOTE: Changing product location has no effect on this variable.

SW_LOCATION

The directory where the installed product is located. This is typically /, but is 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 because it then expands to / /.

This value is derived from a path that can be appended to the product name of each locatable product on the command line or in the software selection file. For example:

swinstall... Accounting:/opt/alternate1/acct...

In the GUI, a Change Product Location menu button (accessible from the Actions pull-down menu) enables you 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 as follows:

PRODUCT[.FILESET], l=$SW_LOCATION, r=revision, a=architecture, v=vendor

SW_CONTROL_DIRECTORY

The directory path where the running script is located. For locations of control scripts, see “Location of the File System” (page 538).

Use this value if the control script invokes a subscript that is shipped with the fileset.

SW_SESSION_IS_KERNEL

This variable is set to TRUE only if there are one or more kernel filesets to be installed. It is unset at all other times. Its value must be tested by control scripts to learn whether a kernel rebuild and subsequent system reboot must happen.

SW_SESSION_IS_REBOOT

This variable is set to TRUE only if there are one or more reboot filesets to be installed. It is unset at all other times. This variable determines that reboot filesets have been selected, not that they are installed or that the system was rebooted because of them.

Relocating the Product

Many non-OSSD-Products can be installed to a directory other than the default directory specified by the developer. This is called relocating the SD-Product. In the GUI, click Change Product.
**Location** to implement this redirection. In command line mode, specify the product directory by appending that directory to the name of the product to be installed as follows:

```
# swinstall -s /opt/ddk/sampldrvrs/qlisps/qlispsdoc QLISPDOC:/opt/docs
```

This command relocates the product QLISPDOC to the new location /opt/docs. SW_LOCATION is now /opt/docs. You can also relocate products while installing to an alternate root as follows:

```
# swinstall -s /opt/ddk/sampldrvrs/qlisps QLISP:/opt/ddk/sampldrvrs/qlisps@newhost: /opt/ddk
```

In the previous example, both SW_LOCATION and SW_ROOT_DIRECTORY change to /opt/ddk/ and newhost.

**System Commands**

Use only commands and syntax that comply to the IEEE Standard 1003.2 (Posix.2) standard.

**NOTE:**

Compliance does not guarantee that all operating systems on which the scripts might run conform to Posix.2.

Because control scripts are run under the POSIX shell, any built-in command that works for the shell must also work in a control script. Table 23-1 lists the valid system commands.

<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>

Table 23-2 lists nonbuilt-in commands that can be used safely by the checkinstall, preinstall, and postinstall scripts. These commands and the control script syntax must be available on all supported architectures and operating systems. In addition, the commands must comply with the Posix.2 standard.

<table>
<thead>
<tr>
<th>awk</th>
<th>cmp</th>
<th>ln</th>
<th>ps</th>
<th>tail</th>
</tr>
</thead>
<tbody>
<tr>
<td>ch rc</td>
<td>cp</td>
<td>ls</td>
<td>rep</td>
<td>uname</td>
</tr>
<tr>
<td>chgrp</td>
<td>ioscan</td>
<td>mkboot</td>
<td>rm</td>
<td>wc</td>
</tr>
<tr>
<td>chmod</td>
<td>lifcp</td>
<td>mkdir</td>
<td>rmdir</td>
<td></td>
</tr>
<tr>
<td>chown</td>
<td>lifls</td>
<td>mv</td>
<td>sed</td>
<td></td>
</tr>
</tbody>
</table>

When an OS is updated, incompatible versions of essential commands can occur on the system. You must first obtain the correct version of SD-UX. Next, put usable versions of these commands in a location on the system where control scripts can access them.

These commands are preserved for the duration of installation for use by the checkinstall, preinstall, and postinstall scripts. Use $SW_PATH in all control scripts to set PATH to
ensure that the preserved version of the command is accessed rather than the newly installed, possibly incompatible version.

In addition to the commands previously listed, the commands shown in Table 23-3 are preserved during an OS update.

**IMPORTANT:** These commands are not appropriate for use in control scripts.

<table>
<thead>
<tr>
<th>Table 23-3 Additional Preserved Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>ar</td>
</tr>
<tr>
<td>cc</td>
</tr>
</tbody>
</table>

**SW-DIST Commands**

In addition to the SD executables, tools are used by SD-initiated processes and are delivered in the SW-DIST product. These include the following:

- `/usr/lbin/sw/control_utils`: Utility functions for all control scripts.
- `/usr/sbin/update-ux`: Utility that updates to a newer OS.

**NOTE:** Installing system software, especially during development cycles, must be preceded by importing the most recent version of the SW-DIST product from the OS-Core depot using `swinstall`. The `update-ux` utility installs the latest SW-DIST automatically.

**General Control Script Actions**

This section describes the details of control script actions such as inputs from the user, the control script output, and any exit values returned from the control scripts.

**Input**

A control script must never require interaction from a user. All control scripts (except request scripts) must be designed to run to completion, whether successful or not, without intervention.

**Output**

Standard output and standard error from control scripts are directed to the `swagent` log file as follows:

```
$SW_ROOT_DIRECTORY/var/adm/sw/swagent.log
```

The log file is located relative to the root directory in which the software is installed. The output consists of messages initiated by the `swagent` process and messages resulting from the control script actions. If any intermediate events must be viewed by the user, 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. The `swagent` command generates no further messages if the script succeeds (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 is handled by the calling process and appended to the log file. Output must be simple echo statements to `stdout`, captured by the parent process and appended to the log file.

For example, output to the log file if a warning is returned from the `checkinstall` script is as follows:

```
* 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.

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 includes the following 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. The return values are as follows:

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 according to script type. See the individual script sections.
2  —  WARNING   A condition exists or an event was detected that does not impede the installation or removal process, but must be presented to the user. This message type is also to be used if the possibility of a failure exists, but not enough information exists to guarantee a failure.
3  —  EXCLUDE   The action differs according to script type. This exit value is only valid in checkinstall and checkremove scripts.

In checkinstall and checkremove scripts, the EXCLUDE exit value has serious consequences when used with filesets that are dependencies for other filesets.

TIP:  HP recommends that this return value be implemented only for isolated filesets.

11  —  GLOBAL_ERROR A serious condition exists that stops the install process from continuing past the analysis phase. The process does not exit immediately, but no selected software can be installed. The remaining check scripts run and can write messages to the log file. The system is checked for available disk space. The session exits after cleaning up.

Required Actions

SD does not require that specific control script be present for correct operation. An absent control script file is functionally equivalent to an existing script that returns SUCCESS. If a fileset's requirements do not include any operations in a specific control script, then that script must be omitted from the fileset's delivery. You must verify whether a control script is needed or not.

A user must be able to rerun a control script. You must include safeguards against a failure for an invocation of a script that has already been run. A script that makes a change to the system the first time it is run must have protection against error for subsequent executions.

Permitted Actions The set of actions that a particular type of script can take is described for each script type in “Control Script Guidelines” (page 543).

Prohibited Actions Do not initiate the following actions 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 running.
• 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 IPD.
• Do not alter files owned by a different fileset.

Do not modify the startup and shutdown scripts directly or indirectly. Each fileset that has a startup or shutdown script must have a configuration file in /etc/rc.config.d, which contains system configuration parameters unique to that fileset. The startup or shutdown script must be completely data free and require no modifications. Use caution when making modifications to the configuration data file.

Do not change any links between the startup and shutdown scripts. The links must exist on the media and remain as shipped after configuration. However, when an application installed in /opt requires a startup script, moving files and creating links are valid components of the install process.

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 that perform commonly used tasks. The control_utils file must be sourced by each control script. If the control script does not require anything from the control_utils file, omit it. The establishment of this function library makes available a standardized set of functions to all filesets.

Configuration Directory and Functions

The newconfig directories and functions are used in configure and preinstall scripts. The newconfig functions handle files delivered to the newconfig directory. These functions are required for the configure, preinstall, and postinstall scripts. The newconfig directory for the OS products is /usr/newconfig.

For example, if /etc/disktab is a fileset that changes with each major release, the working file is the file at the ultimate path. This file might have been modified by the system administrator after being placed there by a previous installation. For example, /etc/disktab.

The previous file is the file delivered by the last software installation of the current fileset. This file must be unchanged. For example, /usr/old/usr/newconfig/etc/disktab.

The new file is the file that is newly installed by swinstall. For example, /usr/newconfig/etc/disktab.

Control Script Guidelines

This section addresses the guidelines for all control scripts in SD. All information needed to write and execute a control script in the SD environment are described, including the purpose and order of script execution, and actions of all control scripts in SD.

Checkinstall Guidelines

A checkinstall script ensures that the target system has no product or fileset specific conditions that can 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, and inappropriate architecture.

Script Execution Order

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 available disk space analysis. The checkinstall script runs only when the user invokes
swinstall, not during swcopy or swremove. The checkinstall script for a specific 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 all run before any installation begins. 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.

For an explanation of the phases, see “Step 5: Packaging the Components” (page 564) and “Installing a Depot” (page 566).

**Available System Commands**

Few commands are available to the checkinstall script under all conditions. They include the Posix.2 commands listed in “System Commands” (page 540), and commands that are part of the SW-DIST product. For example, use the get_sysfile command 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, you must specify the command with its full path.

**Actions**

Output:

See “Output” (page 541).

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 as follows:

- **0** SUCCESS. No impediment to installation found. Installation proceeds immediately if in noninteractive mode. If in interactive mode, the process waits for the user to select the start_install action item.
- **1** FAILURE. In a product level checkinstall script, the entire product is unselected, along with all other products that have any of this product's filesets as a requisite. In a fileset level checkinstall script, the fileset is unselected along with all other filesets that have this fileset as a requisite. The error message must explain what happened. Any remaining filesets and products unaffected by the failure remain selected for installation. Any remaining checkinstall scripts run in install order.
- **2** WARNING. Possible conflict is explained in the warning message. The selection of the current fileset and its prerequisite and corequisite filesets remains intact, barring other failures. The warning message must explain the condition.
- **3** EXCLUDE. The current fileset is unselected. A message to the log file must explain the unselection. Any remaining filesets unaffected by the unselection remain selected for installation. Any remaining checkinstall scripts run in install order.
- **11** GLOBAL_ERROR. The install session exits before any software is installed. The analysis phase proceeds through all the checks in order to accumulate all analysis information. This enables a checkinstall script to prevent installation if test results warrant such action.

**Required Actions**

The actions of a checkinstall script must be unobtrusive because there is no commitment to installing at the time the scripts run. The
checkinstall script, like all control scripts, must be re-executable. A checkinstall script can be executed numerous times within a single install session.

Permitted Actions
Typical actions valid from within a checkinstall script at any time are as follows:

- 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.
- Test for software already installed.
- Test the revision or other attributes of installed software.

Prohibited Actions
See “Required Actions” (page 542). In addition, the checkinstall script must not do any of the following:

- 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.

Preinstall Guidelines
A preinstall script prepares the system for fileset installation. By the time the preinstall script for a particular fileset is executed, the fileset’s checkinstall script has determined that there are no fileset-specific impediments to installing the fileset. The analysis phase has ensured that there are no known system-specific impediments to installing the current fileset.

A typical preinstall script kills processes that can interfere with installation, for example daemons that keep an executable file opened. The preinstall script also removes obsolete software.

Script Execution Order
The preinstall scripts are called while the installation is in the execution phase. Each fileset’s preinstall script executes just before that fileset’s files are installed onto the target system. A product level preinstall script is called prior to calling any product fileset’s preinstall scripts.

Available System Commands
Few commands are available to the preinstall script under all conditions. They include the Posix.2 commands that are part of the SW-DIST product listed in “System Commands” (page 540).

Actions
Output:
See “Output” (page 541).
Exit Values:
The calling process acts upon the value returned by each preinstall script. The return values are as follows:

- 0 SUCCESS. All processes ran successfully.
- 1 FAILURE. A failure occurred, indicating that the installation and configuration is certain to fail. The failure message must explain the failure. If a kernel fileset’s preinstall script returns FAILURE, the install process exits if it is in noninteractive mode, and suspends and waits for user input if in interactive mode. When the fileset is neither a kernel fileset nor a prerequisite for a kernel fileset, file installation proceeds even if a preinstall failure occurs.
2 WARNING. A warning was sent to the log file. A condition was detected that might result in installation or configuration failure. The warning message must explain the condition. Selected software installation proceeds.

Actions

Typical actions valid from within a preinstall script are as follows:

Call the newconfig_prep function to set aside files to be delivered under /usr/newconfig.

The fileset's preinstall script must handle all files delivered under /usr/newconfig by that fileset. It must 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. For example:

`newconfig_prep /etc/disktab`

The function copies the previous file from its place under /usr/newconfig to a corresponding path under /usr/old/usr/newconfig (see “Configuration Directory and Functions” (page 543)). The file under /usr/newconfig remains. Results are unspecified if the working file is linked to the previous file, or if either is linked to any other file.

The newconfig_cp function is usually called from a fileset's configure script. It must be called for every file handled by newconfig_prep.

Before the fileset's postinstall or configure script executes, the previously installed file has been copied (by the newconfig_prep function) from under /usr/newconfig to /usr/old/usr/newconfig, the new file has been installed under /usr/newconfig, and the working (that is, currently installed) file is unaffected. The newconfig_cp function decides whether the newly installed file overwrites the working file, and does so if necessary. Invoke the newconfig_cp function 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. For example:

`newconfig_cp /etc/disktab`

The function first checks for the existence of the new file, and exits with an error value if the new file is not found. This error can occur if the file is delivered directly under /usr/newconfig rather than as a pseudoroot, or if the argument is passed as a relative path rather than an absolute path.

Prohibited Actions

The list of prohibited actions described in “Required Actions” (page 542) also applies here.

Postinstall Guidelines

A postinstall script prepares for a kernel build when required by the install conditions. The script can also drive events that must occur before a system reboots.

Script Execution Order

Each fileset's postinstall script runs just after the fileset's files load 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 are installed.

Available System Commands

Few commands are available to the postinstall script under all conditions. They include the Posix.2 commands listed in “System
 Commands” (page 540) and commands that are part of the SW-DIST product.

Actions

Output:

See “Output” (page 541).

Exit Values:

The calling process acts upon the value returned by each postinstall script. The return values are as follows:

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 postinstallation action failed. This can jeopardize the correct functioning of the fileset.

   If a kernel fileset’s postinstall script returns FAILURE, the install process exits if in noninteractive mode, and suspends and waits for user input if in interactive mode.

   When the fileset is neither a kernel fileset nor a prerequisite for a kernel fileset, the installation of all other files proceeds even if a postinstall failure occurs.

2  WARNING. Installation succeeded, but a condition exists for which the user must 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 as follows:

- 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 “Required Actions” (page 542) are also applicable.

Configure Guidelines

A configure script performs 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 if the user invokes the swconfig command without the -u option. The configure script is not run when the user has invoked swcopy or swremove.

Execution

Because the configure scripts are run on the target system, $SW_ROOT_DIRECTORY is / (slash) in all cases when configure is run.

Environment

Script Execution Order

If one or more reboot filesets (including kernel) is successfully installed, then execution of the configure scripts is postponed until after the system reboots. If all of the selected filesets are nonreboot filesets, the configure scripts are invoked from within the swinstall session as
the last step of the execution phase. The configure scripts execute in product order, not install order.

The order is affected only by prerequisites and kernel filesets. Within any one software product, the configure script executes first for kernel filesets and their prerequisites, then for other nonkernel filesets. The configure scripts for a fileset’s prerequisites 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, it is called before any configure scripts belonging to that product’s filesets are called.

Available System Commands

More commands are available to configure scripts than to preinstall and postinstall scripts. These include the following:

- Posix.2 commands listed in “System Commands” (page 540)
- SW-DIST product commands
- Prerequisite fileset commands.
- Commands included in the script’s own fileset that do not need to be configured before using

Actions

Output:

See “Output” (page 541).

Exit Values:

The calling process acts upon the value returned by each configure script. The return values are as follows:

0  SUCCESS. Fileset configuration succeeded. The fileset state changes to 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 fileset state remains INSTALLED.

2  WARNING. Configuration succeeded, but a condition exists for which the user must be notified. The fileset state is CONFIGURED.

3  EXCLUDE. Fileset configuration must be run again to complete the configuration. The fileset state remains INSTALLED. The next invocation of the configure script by swconfig at system reboot completes the configuration process.

Permitted Actions

Typical actions valid from within a configure script at any time are as follows:

- 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 that were delivered to a location under a private directory (that is, one only known to the packager).

Prohibited Actions

The list of prohibited actions described in “Required Actions” (page 542) also applies here.
Unconfigure Guidelines

An unconfigure script undoes most configuration changes made to the system by the corresponding configure script. Typical actions in an unconfigure script remove device special files or change a system configuration parameter in a file under /etc/rc.config.d.

Order of Execution

The unconfigure script is invoked as part of the swremove process, or the user gives the -u option to the swconfig command. Each fileset's unconfigure script executes just after the user proceeds with removing the selected products and filesets, and just before the selected products and filesets are actually removed.

If a software product has an unconfigure script, it is called after any unconfigure scripts belonging to that product's filesets are called. The unconfigure scripts for any swremove session are invoked in reverse prerequisite order. The unconfigure script for all prerequisite filesets selected for removal is run after the unconfigure script for the current fileset. The presence of kernel or reboot filesets has no effect on the execution order.

Available System Commands

The commands available to the unconfigure script under all conditions include the following:

- Posix.2 commands listed in “System Commands” (page 540)
- SW-DIST product commands
- Prerequisite fileset commands
- This script's fileset commands
- Core system commands

Actions

Output:
See “Output” (page 541).

Exit Values:

The calling process acts upon the value returned by each unconfigure script. The return values are as follows:

- 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 proceeds. The install state is now INSTALLED.
- 2 WARNING. A nonfatal condition was detected and must be reported to the user. If called as part of a swremove process, the attempt to remove the files in the product or fileset proceeds. The install state is 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 as follows:

- Kill processes, including daemons, owned or spawned by files in the current fileset.
- Move or remove files and directories 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 described in “Required Actions” (page 542) also applies here.

Preremove Guidelines

The preremove script performs any necessary actions not done in the unconfigure script to prepare to remove the fileset's files. It undoes any actions taken in a postinstall script. For kernel filesets, it modifies the /stand/system file to delete entries such as driver names and configurable parameters with functionality that is part of the fileset.

Script Execution Order
The preremove scripts are called during the execution phase. Each fileset's preremove script executes just before the fileset files are actually removed. If a software product has a preremove script, it is called before any of that product's fileset level preremove scripts are called. The preremove scripts for any swremove session are invoked in reverse prerequisite order. The preremove script for all prerequisite filesets selected for removal runs after the preremove script for the current fileset. The presence of kernel or reboot filesets has no effect on the execution order.

Available System Commands
The commands available to the preremove script under all conditions include the following:
• Posix.2 commands listed in “System Commands” (page 540)
• SW-DIST product commands
• Prerequisite fileset commands
• This script's fileset commands, provided the unconfiguration has not rendered them unusable
• Core system commands

Actions
Output:
See “Output” (page 541).
Exit Values:
The calling process acts upon the value returned by each preremove script. The return values are as follows:
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 proceeds.
2 WARNING. A nonfatal condition was detected and must be reported to the user. The attempt to remove the files in the product or fileset proceeds.

Permitted Actions
Typical actions valid from within a preremove script are as follows:
• Move or remove files and directories.
• Modify the /stand/system file.

Prohibited Actions
The list of prohibited actions described in “Required Actions” (page 542) also applies here.

Postremove Guidelines

The postremove script performs any necessary cleanup actions after the fileset's files are removed. Any files created by the fileset and that might not have been added to the IPD must
be removed by the `postremove` script. A typical action in a `postremove` script is the removal of newly emptied directories when those directories are the exclusive property of the fileset.

**Script Execution Order**

The `postremove` script for a fileset is called in the execution phase immediately after that fileset’s files are removed from the system. If a software product has a `postremove` script, it is called after any `postremove` scripts belonging to that product’s filesets are called. The `postremove` scripts for any `swremove` session are invoked in reverse prerequisite order. The `postremove` script for all prerequisite filesets selected for removal runs after the `postremove` script for the current fileset. The presence of kernel or reboot filesets has no effect on the execution order.

**Available System Commands**

The commands available to the `preremove` script under all conditions include:

- Posix.2 commands listed in “System Commands” (page 540)
- SW-DIST product commands
- Prerequisite fileset commands
- Core system commands that have not been affected by the current removal

**Actions**

Output:

See “Output” (page 541).

Exit Values:

The calling process acts upon the value returned by each `postremove` script. The return values are as follows:

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 must be reported to the user.

**Permitted Actions**

Typical actions valid from within a `postremove` script at any time are as follows:

- Remove newly emptied directories owned and used exclusively by the fileset.
- Replace any files set aside by the installation of the fileset.
- Any other actions that bring the system closer to the condition it was in prior to the fileset’s installation.

**Prohibited Actions**

The list of prohibited actions described in “Required Actions” (page 542) also applies here.

**Files to Package**

To create a product, you need the object and source files, or document files on which the PSF and control scripts act to install, uninstall, and verify a product. These files can be drivers, libraries, tools, or a document. The path names of these files must be specified in the PSF both in the package and their final destinations when a product gets installed on a host.

**Creating a Software Depot**

Follow these steps to create an SD package:

- “Step 1: Designing an SD Structure”
- “Step 2: Selecting the Product Directory Structure” (page 552)
Step 1: Designing an SD Structure
As explained in “SD Package Components” (page 514), packaging can be done at the SD-Product level or at the Bundle level. Select the level that best suits your package. If you intend to have more than one product as part of a package, choose Bundle-level packaging. If not, omit the SD-Bundle and package at the SD-Product level. Depending upon the use of the SD depot, use SD-Subproduct objects; subproducts group logically related filesets. You must also decide whether the packaging requires any specific control scripts.

Step 2: Selecting the Product Directory Structure
After you decide the SD package structure, you must select the product directory structure (location). This is the directory structure where all the source/object/bin/docs files reside after installation. You must decide this before writing the PSF because directory information is required as one of the attributes in the SD-Product and SD-Fileset categories of the PSF.

Step 3: Writing a PSF
After you decide the SD structure and product directory structures, you can write a PSF. A PSF is a master file that has the configuration attributes of the package. These attributes take effect once the product is installed.

An overview and all the attribute policies are explained in detail in “Product Specification File” (page 515) and “SD Objects Attributes Classification and Flow” (page 516).

To omit the bundle and consider product-level packaging, do not include the bundle attribute for this package. When considering bundle-level packaging, both bundle and product attributes must be included. In both cases, fileset attributes are needed. The subproducts attributes are optional.

The attributes mentioned in “SD Objects Attributes Classification and Flow” (page 516) are required.

NOTE: HP recommends using policies. If you do not use policies, the PSF generates errors during package creation.

Writing a PSF Vendor Object
In the PSF, the vendor information of the package must be filled in first. This information, such as company name, must be unique.

You must fill in all of the following attributes:

- **tag**  The vendor's trademark.
- **title**  The vendor's trademark expansion or company name.
- **description**  Description of the vendor.

Following is the syntax of a vendor object:

```xml
vendor
tag          <....>
title        <.....>
description  <.....>
end
```

For example:
Writing a PSF Category Object

Category attributes are defined outside the SD-Bundle, SD-Product, or SD-Fileset. The category_tag attribute within the SD-Bundle, SD-Product, or SD-Fileset definition refers to the associated category object. Category class attribute definitions are global — a category tag and description applies to all referring objects in the depot or on the media.

For more information about category object attributes, see “Category Attributes” (page 517):

tag Identifies the category object.
title Detailed name for the category.
description A text string describing the category.

Following is the syntax for a category object:

category
  tag <.....>
title <.....>
description <.....>
end

For example:

#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 object follows the vendor object. All products and filesets in the bundle are mentioned in this object.

For more information about bundle object attributes, see “SD-Bundle Attributes” (page 522):
tag Bundle identifier.
title Detailed name of the bundle.
description A text string describing the bundle.
revision Bundle revision number.
architecture Target systems on which the bundle will be installed.
os_name The OS releases on which the bundle will run.
os_release The OS releases on which the bundle will run.
os_version (optional) The OS versions on which the bundle will run.
vendor_tag Short name of the vendor (see “Writing a PSF Vendor Object” (page 552).
contents Software contained in the bundle.

Following is the syntax for a bundle object:

bundle
  tag <.....>
title <.....>
description <.....>
revision <.....>
vendor_tag <.....>
architecture <......>
os_name <......>
os_release <......>
os_version <......>
contents <......>
end

For example:

#bundle information

bundle
tag              B11_23QLISP
title             "11.23 QLISP sample driver"
description      "11.23 QLISP sample driver version 1.0"
revision         11_23.1.0
vendor_tag       HP
architecture    HP-UX_B.11.23_IA/PA
os_name         HP-UX
os_release      ?.11.23
os_version      *
machine_type    *
contents        QLISP11_23_1_0.qlispl_0src,r=11_23.1.0,
a=HPUX_B.11.23_IA/PA,v=HP
end #bundle

Writing a PSF Product Object

The product object follows the bundle object. If you are creating product-level packaging, omit
the bundle information; the product object follows the vendor object. Include all filesets of the
products in this object.

NOTE: In this document, a fileset is considered a separate object.

The following are the product object attributes:
tag          Product name or identifier.
title         Detailed name of the product.
description   A text string describing the product.
revision      Product revision (release) number.
architecture  Target systems on which product will be installed.
oss            OSs on which the product will be installed.
os_release    OS release on which the product will be installed.
os_version    (optional), OS versions on which product will be handled.
directory     The default, absolute path name of the directory in which the product will
              be installed.
is_locatable  The product alternate directory.
vendor_tag     Short name of the vendor (see “Writing a PSF Vendor Object” (page 552).
preinstall     (optional) Path name of install preload product control script (holds the
                same for all other control scripts).

See also the policies mentioned in “SD-Product Attributes” (page 526).

NOTE: The revision, architecture, os_name, os_release, and os_version attributes
are the same as in the bundle object for bundle-level packaging.

Following is the syntax for a product object:
Writing a PSF Subproduct Object

The subproduct object is optional. The subproduct object follows the product object.

The following are attributes of the subproduct object:

tag Subproduct identifier.
title Detailed name of the subproduct.
description A text string describing the subproduct.

The policies are described in “SD-Subproduct Attributes” (page 530).

All subproduct object attributes are optional.

Following is the syntax for a subproduct object:

```
subproduct
tag <.......>
title <.......>
description <.......>
contents
end
```

For example:

```
#Subproduct information
subproduct
tag QLISP11_23_1_0
title "QLISP Sample drivers 1.0.0 for 11.23 DDK 1.0"
description "This subproduct contains a collection of all sources of QLISP 1.0"
contents QLISP11_23_1_0.qlispsrc1_0,r=11_23.1.0,
a=HPUX_B.11.22_IA/PA,v=HP
end #subproduct
```
Writing a PSF Fileset Object

The fileset object follows the product object, if you do not use subproduct objects. Because filesets are the atomic units, fill in all the attributes for files that are to be packaged here.

The following are the attributes of a fileset object:

- **tag**: Fileset identifier.
- **title**: The detailed name of the fileset.
- **description**: A text string describing the fileset.
- **revision**: Fileset revision number.
- **architecture**: Target systems on which the fileset will be installed.
- **os_name**: OSs on which the fileset will be installed.
- **os_release**: OS releases on which the fileset will be installed.
- **os_version**: (optional) OS versions on which the 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 preload fileset control script (holds the 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 file names are located (for this fileset only).
- **file**: Specifies the files to be packaged into a fileset.

See also “SD-Fileset Attributes” (page 531).

**NOTE:** The revision, architecture, os_name, os_release, and os_version attributes are the same as in product objects.

Following is the syntax for fileset objects:

```plaintext
fileset
  tag               <.......>
  description       <.......>
  title             <.......>
  revision          <.......>
  architecture      <.......>
  os_name           <.......>
  os_release        <.......>
  os_version        <.......>
  file_permissions  <.......>
  directory         <.......>
  file              <.......>
end #fileset
end #product (if this is the last fileset in the product)
```

For example:

```plaintext
fileset
tag             qlispsrc1_0
description      "DDK11.23_1.0 MS HBA sample drivers"
title            DDK11_23-qlisp-11_23_1_0
revision         11_23.1.0
architecture     HP-UX_B.11.23_IA/PA
os_name          HP-UX
os_release       ?.11.23
os_version       *
machine_type     *
```
Step 4: Writing Control Scripts

This section addresses the usage, templates, and samples of all the SD control scripts.

Writing a Checkinstall Script

A checkinstall script ensures that the target system has no product- or fileset-specific conditions that can 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, and inappropriate architecture.

Order of script execution, commands available for the checkinstall script, and its actions are explained in detail in “Checkinstall Guidelines” (page 543).

The actions of a checkinstall script must be unobtrusive because 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.

Use the checkinstall script 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.

The following is the checkinstall script template:

```
#!/sbin/sh
#
# Product: DEMO
# Fileset: LATEST
# checkinstall
# @(#) $Revision: 1.1 $
#
#
# (c) Copyright 2007, Hewlett-Packard Company
#
# UTILITY="/usr/lbin/sw/control_utils"
if [[ ! -f $UTILITY ]];
```
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 whether you are trying to install over a more recent version.
# If so, skip this installation. B.11.23 is hard-coded as the OS release.
if [[ -x $WHAT_BIN && -e $PROD_LIB ]]; then
  what_str='$WHAT_BIN $PROD_LIB'
  print $what_str | /usr/bin/grep -q 'B.11.23'
  if [[ $? = 0 ]]; then
    INST_VER='print $what_str | /usr/bin/awk '{print substr($0,index($0,"B.11.23.")+8,2);}'
    if [[ $INST_VER -gt $PROD_VER ]]; then
      echo "ERROR: A newer version of the PCI FDDI has already been installed"
      echo " on this system (perhaps from a patch)."
      echo " The version in the kernel is B.11.23.$INST_VER."
      echo " The version attempted is B.11.23.$PROD_VER."
      echo " To force the installation, you must first swremove this"
      echo " product from the kernel, then retry the installation."

      exit $exitval
    fi
  fi
fi

Writing a Preinstall Script

The preinstall script prepares the system for installation of the fileset. Before the preinstall script for a specific fileset runs, the actions of that fileset’s checkinstall script determine that there are no fileset-specific impediments to installing the fileset. Then, the analysis phase ensures that there are no known system-specific impediments to installing the current fileset. Next, the fileset’s preinstall script takes the steps to prepare the system for installation.

A typical preinstall script kills processes that can interfere with installation, for example daemons that keep an executable file opened. Preinstall scripts also remove obsolete software. The details such as order of script execution, commands available for the preinstall script, and its actions are explained in “Preinstall Guidelines” (page 545).

The following is the preinstall script template:

```bash
#!/sbin/sh

########
# Script information and what it is used for
########

Set up the SD environment by calling control utilities

Preinstall functions such as newconfig

The following is a sample preinstall control script:

#!/sbin/sh

# Product: DEMO
# Fileset: LATEST
# preinstall
# @(#) $Revision: 1.1 $
#
# (c) Copyright 2007, Hewlett-Packard Company
#

```
UTILS="/usr/lbin/sw/control_utils"
if [[ ! -f $UTILS ]]
then
  echo "ERROR: Cannot find $UTILS"
  exit 1
fi
.
$UTILS
exitval=$SUCCESS

# Do the newconfig preparation step for configuration files.
for file in /etc/rc.config.d/demo
do
  newconfig_prep $file
  retval=$?
  [[ $exitval -ne $FAILURE && $retval -ne $SUCCESS ]] &&
  exitval=$retval
  # newconfig_prep has already reported any warnings
done
exit $exitval

Writing a Postinstall Script

The postinstall script prepares for a kernel build when required by the install conditions. They also can drive events that must occur before the system reboots. 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.

The details such as order of script execution, commands available for the postinstall script, and its actions are explained in detail in “Postinstall Guidelines” (page 546).

Use the postinstall script to do the following:

- 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.

The following is the postinstall script template:

#!/sbin/sh
#
###
Script information and what it is used for
#
###
Set up the SD environment by calling control utilities
#
###
Preinstall script action, for example, copy a driver to a particular system location

The following is a sample postinstall control script:

#!/sbin/sh
#
###
# Product: DEMO
# Fileset: LATEST
# postinstall
# @(#) $Revision: 1.1 $
UTILS="/usr/lbin/sw/control_utils"
if [[ ! -f $UTILS ]]
then
  echo "ERROR: Cannot find $UTILS"
  exit 1
fi
$. $UTILS
exitval=$SUCCESS

# This example 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 Script

The configure script performs 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 runs swcopy or swremove.

Use the configure control script to do the following:

- Create device special 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 that were delivered to a location under a private directory (that is, one known only to the packager).

The following is the configure script template:

```
#!/sbin/sh

Script information and what it is used for

Set up the SD environment by calling control utilities

Configure script action, for example, configure nettl

The following is a sample configure control script:
```

```bash
#!/sbin/sh

# Product: DEMO
# Fileset: LATEST
# configure
# @(#) $Revision: 1.1 $
```

560 Creating a Software Depot
### UTILS="/usr/lbin/sw/control_utils"

if [[ ! -f $UTILS ]]
then
  echo "ERROR: Cannot find $UTILS"
  exit 1
fi

. $UTILS

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 Script

The unconfigure script undoes most configuration changes that were made to the system by the corresponding configure script. A typical action in an unconfigure script is the removal of device special files or the changing of a system configuration parameter in a file under /etc/rc.config.d.

Use the unconfigure script to do the following:

- 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.

The following is the unconfigure script template:

```bash
#!/sbin/sh

# Script information and what it is used for

# Set up the SD environment by calling control utilities

Configure script action, for example, remove nettl entry

The following is a sample unconfigure control script:

```bash
#!/sbin/sh

# Product: DEMO
# Fileset: LATEST
# configure
# @(#) $Revision: 1.1 $ 

# (c) Copyright 2007, Hewlett-Packard Company

UTILS="/usr/lbin/sw/control_utils"

if [[ ! -f $UTILS ]]
then
  echo "ERROR: Cannot find $UTILS"
```
Writing a Preremove Script

The preremove script performs any necessary actions not done in the unconfigure script in preparation for the removal of the fileset's files. It must undo any actions taken in a postinstall script. For kernel filesets, it must modify the /stand/system file to delete entries such as driver names and configurable parameters whose functionality is part of the fileset.

Use the preremove script to do the following:
• Move or remove files and directories.
• Modify the /stand/system file.

The following is the preremove script template:

```bash
#!/sbin/sh
######### Script information and what it is used for
Set up the SD environment by calling control utilities

Preremove script action, for example, remove system files for drivers

The following is a sample preremove control script:

#!/sbin/sh
#####
# Product: DEMO
# Fileset: LATEST
# preremove
# @(#) $Revision: 1.1$
#####
# (c) Copyright Hewlett-Packard Company 2007
#

UTILS=/usr/lbin/sw/control_utils
if [ -f $UTILS ];
then
  . $UTILS
else
  echo "ERROR: Cannot find $UTILS"
  exit 1
fi
exitval=$SUCCESS

```

```bash

```

562 Creating a Software Depot
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 Script

The postremove script performs any necessary cleanup actions after the fileset's files have been removed. Any files that might have been created by the fileset and that might not have been added to the IPD must be removed by the postremove script. A typical action in a postremove script is the removal of newly emptied directories when those directories are the exclusive property of the fileset.

Use a postremove script to do the following:

- Remove newly emptied directories that are owned and used exclusively by the fileset.
- Replace any files set aside by the installation of the fileset.
- Any other actions that bring the system closer to the condition it was in prior to the fileset's installation.

The following is the postremove script template:

```
#!/sbin/sh

######### Script information and what it is used for
#########
Setup the SD environment by calling control utilities

Postremove script action, for example, remove newly emptied directories

The following is a sample postremove control script:
```

```
#!/sbin/sh

# Product: DEMO
# Fileset: LATEST
# postremove
# @(#) $Revision: 1.1 $
#
# (c) Copyright Hewlett-Packard Company 2007
#

UTILS="/usr/lbin/sw/control_utils"

if [[ ! -f $UTILS ]]
then
   echo "ERROR: Cannot find $UTILS"
   exit 1
fi

$UTILS exitval=$SUCCESS

```

# Remove formatted manpages

for file in

Generating a Postremove Script

The postremove script performs any necessary cleanup actions after the fileset's files have been removed. Any files that might have been created by the fileset and that might not have been added to the IPD must be removed by the postremove script. A typical action in a postremove script is the removal of newly emptied directories when those directories are the exclusive property of the fileset.

Use a postremove script to do the following:

- Remove newly emptied directories that are owned and used exclusively by the fileset.
- Replace any files set aside by the installation of the fileset.
- Any other actions that bring the system closer to the condition it was in prior to the fileset's installation.

The following is the postremove script template:

```
#!/sbin/sh

######### Script information and what it is used for
#########
Setup the SD environment by calling control utilities

Postremove script action, for example, remove newly emptied directories

The following is a sample postremove control script:
```

```
#!/sbin/sh

# Product: DEMO
# Fileset: LATEST
# postremove
# @(#) $Revision: 1.1 $
#
# (c) Copyright Hewlett-Packard Company 2007
#

UTILS="/usr/lbin/sw/control_utils"

if [[ ! -f $UTILS ]]
then
   echo "ERROR: Cannot find $UTILS"
   exit 1
fi

$UTILS exitval=$SUCCESS

```

# Remove formatted manpages

for file in

Creating a Software Depot 563
ninstall \ninstallld
do
   rm -f /usr/share/man/cat1m.Z/${file}.1m
done

########
# Remove files that might not have been removed

for file in \
   /opt/demo/app/log* \n   /dev/demo
do
   rm -f $file >/dev/null 2>&1
done

########
# Remove empty directories that this fileset owned exclusively.

for dir in \
   /opt/demo/app \n   /opt/demo/config \n   /opt/demo/data \n   /opt/demo
do
   rm -rf $dir >/dev/null 2>&1
done

exit $exitval

Step 5: Packaging the Components

After you finish writing the PSF and control scripts, package the components. SD provides the swpackage command to package these components. This section focuses on some of the basic uses of swpackage. See swpackage(1M) for more information.

NOTE: All SD commands are provided as manpages in HP-UX.

SD Components Packaging

As explained in “Creating a Software Depot” (page 551), a package consists of a PSF file, the control scripts, and the source and object files of product that must be packaged for distribution. The command used to create a package into a target depot is swpackage.

Syntax

swpackage [-s *.*psf]

Inputs  PSF file, depot directory

Options  
-s  PSF file

-d  Depot directory

-p  Previews the package without creating a depot

Usage 1  swpackage [-s PSF file path]

Usage 2  swpackage [-s PSF file path][-d depot directory path]

In the following examples, the PSF file of the package is in the /home/qlisp/ directory and the depot directory is /home/qlisp/driversource.

# swpackage -s /home/qlisp/qlispdrv.psf

or

# swpackage -s /home/qlisp/qlispdrv.psf -d /home/qlisp/driversource

The default depot directory is /var/spool/sw. If you do not specify the depot directory, swpackage puts the depot into /var/spool/sw. Alternatively, if you specify a directory, swpackage puts the depot into the directory you specify.
The `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 are used. Then it does the following:

- Determines the product, subproduct, and fileset required for the structure.
- Determines which files are contained in each fileset.
- Determines the attributes associated with each objects.
- Checks PSF syntax and terminates the session if any errors are encountered.

Phase 2  Analysis

The `swpackage` command performs the following checks:

1. Checks for unresolved dependencies.
   
   For every fileset in each selected product, `swpackage` checks to see if a requisite of the fileset is also selected or already present in the target depot. Unresolved dependencies within the product generate errors. Unresolved dependencies across products produce notes.

2. Checks 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. Performs Disk Space Analysis (DSA).
   
   The `swpackage` command 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, `swpackage` writes a note to the log file about 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. Builds the software package.
   
   When packaging a product, if the target depot does not exist, `swpackage` creates it. If it does exist, `swpackage` merges new products 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 repackaging (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 or rebuilt.
   - It checks if the product is new or already exists. If it is new, it creates the product’s storage directory.
   - For each fileset in the product, copies the fileset’s files into their storage location (within the product’s storage directory), and creates the fileset’s catalog (database information) files.
   - After the individual filesets, creates the product’s informational files (metafiles).
NOTE: When creating a depot using swpackage, you must set is_locatable TRUE to enable users to change the target directory with the swinstall command. The swpackage command does not register the depot it creates; it asks you to register.

Step 6: Registering the Depot

After the depot is created, you must register the depot to install and manage the depot further. The command used to register the depot is swreg.

Syntax

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 depot.

Installing a Depot

After an SD depot is created and registered, it is ready for distribution. If a user wants to use the software, the depot must be installed first on a host. The command used to install a depot is swinstall.

The following are 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, such as all install and request scripts

Using swinstall

The swinstall command provides three interfaces.

For the swinstall Terminal User Interface (TUI), enter the following:

# /usr/bin/swinstall

For the swinstall Graphical User Interface (GUI), enter the following:

# /usr/bin/swinstall -i

The command-line interface has the following syntax:

Syntax

swinstall [-s] [depot source] [depot]

Options

- s source
- p
- x command_option=value
@target_selections

The following section describe to use swinstall.

Step 1: Selecting the Source Depot

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 follow these steps:

1. Click **Source Host Name**. 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. default depot path.

Optionally, to specify the path to the depot, type a new path, or follow these steps:

1. Click **Source Depot Path** 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.

**Step 2: Selecting the Software**

Use the Software Selection window to select the software to be installed.

**Step 3: Analyzing the Software**

SD-UX analyzes the software you 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.

**Step 4: Installing the Software**

In this step, SD-UX installs the software.

**Examples**

In the following command, `/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.

```
# swinstall -s /home/qlisp/ qlispdepot
```

The following command previews the installation, but does not install the depot:

```
# swinstall -p -s /home/qlisp/ qlispdepot
```

The following command changes the target install directory specified in the PSF file to `/home/elsewhere`:

```
# swinstall -s /home/qlisp/ qlispdepot:/home/elsewhere
```

The following command installs the depot into some other host:

```
# swinstall -s /home/qlisp/ qlispdepot:otherhost
```

**Command Options**

You can change the behavior of the command by changing its options. These can be set with a value of `true` or `false`. Some also require settings as attributes in the PSF files. The following are some `swinstall` options:

- **ask**
  - 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`: Does not execute request scripts. (Default for `swinstall` and `swconfig`)

- **autoreboot**
  - 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 enables installation of the software and automatically reboots the local host.

**reinstall**

Prevents SD-UX from reinstalling (overwriting) an existing revision of a fileset. If set to true, the fileset is reinstalled.

**allow_downdate**

Normally set to false, so installing an older version of software over a new one is disallowed. This prevents installing older versions by mistake. Additionally, many software products do not support downdating.

If set to true, a previous version can be installed, but SD-UX issues a warning message.

**allow_multiple_versions**

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, you can install and manage multiple versions of the same software.

---

### Managing the Depot Software

After the depot is installed, manage the software to suit your needs. SD provides the following `swlist` and `swcopy` depot managing commands.

#### The `swlist` Command

The `swlist` command displays registered depots installed on a host.

**Syntax**

```
swlist
```

**Options**

- `-l depot @ hostA` Lists all depots on remote host A.
- `-l product Bundle` Lists all products in the specific Bundle.
- `-l fileset Bundle` Lists all filesets in a specific Bundle.
- `-l fileset Product` Lists all filesets in a specific product.

For example:

```
swlist -l depot
swlist -l product QLISPBUNDLE
swlist -l fileset QLISPBUNDLE
swlist -l fileset QLISPDRVSRC
```

#### The `swcopy` Command

The `swcopy` command copies or merges selected software from a software source to one or more software depot targets. You can use `swcopy` from the GUI or the command line.

**Syntax**

```
swcopy [-s source] [@ target_selections]
```

**Options**

- `-s source` Specifies the source depot from which software is installed or copied. The default source type is `directory`. The syntax is as follows:
  
  `[host][[:]][/directory]`

- `@ target_selections` Specifies the target locations for the software. The syntax is as follows:
  
  `[host][[:]][/directory]`

---

568 Creating a Software Depot
In the following example, /home/ddk/depots/QLISP/QLISP_drvsrс is a depot path of QLISP driver sources. The `swcopy` command copies the QLISP drvsrс product to the /opt/qlisp depot path on local host:

```
# swcopy -s /home/ddk/depots/QLISP/QLISP_drvsrс /opt/qlisp
```

In the following example, `swcopy` copies the QLISP drvsrс product to the /opt/qlisp depot path on host A:

```
# swcopy -s /home/ddk/depots/QLISP/QLISP_drvsrс @ hostA:/opt/qlisp
```

### Removing the Installed Software

You can remove any installed software with the `swremove` command.

**Syntax:** `swremove Bundle/product/product`

Enter the following command to remove all software installed from the Bundle.

```
# swremove Bundle
```

To remove all software installed from the product contained in the bundle, enter the following command:

```
# swremove Bundle.product
```

To remove all software installed from the fileset contained in a product that is part of a bundle, enter the following command:

```
# swremove Bundle.product.fileset
```

To remove all product software, enter the following command:

```
# swremove product
```

To remove all driver sources, binaries, and documents from the QLISP bundle, enter the following command:

```
# swremove QLISPBundle
```

To remove only the driver sources, but not binaries and documents, enter the following command:

```
# swremove QLISPBundle.QLISPdrvsrс
```

If QLISPdrvsrс is not part of any bundle, enter the following command:

```
# swremove QLISPdrvsrс
```

---

**NOTE:** The `swremove` only removes the files (software). It does not remove the directory path where the files are installed.

**TIP:** HP recommends that you use the SD commands from command line instead of GUI. GUI commands are limited, for instance, user input cannot be obtained from a GUI.

### Command Messages Logging

The SD commands' messages are logged in `/var/adm/sw/swagent.log`.

### qlisp Driver Examples

This section contains examples for the `qlisp` driver.

### Creating a Package

This section contains the files necessary to create a package for the `qlisp` driver.

### PSF

The PSF file for the `qlisp` driver (`qlisp.psf`) is as follows:
vendor
tag       HP
title     Hewlett-pakard
description Hewlett-Packard Corporation
end
bundle
tag           B11_23QLISP
title         "11.23 QLISP sample driver Version 1.0"
description "11.23 QLISP sample driver Version 1.0 with only sources"
revision      11_23.1.0 ( suggested use - major:minor:release )
vendor_tag    HP
architecture HP-UX_B.11.23_IA/PA
os_name       HP-UX
os_release    ?.11.23
os_version    *
architecture   HP-UX_B.11.23_IA/PA
os_name       HP-UX
os_release    ?.11.23
os_version    *
architecture   HP-UX_B.11.23_IA/PA
os_name       HP-UX
os_release    ?.11.23
os_version    *
file_permissions -m 0555
directory    /sampldrv/ms/scsihba=/opt/ddk/11.23/sampldrvsmassstorage/qlisp/11.23.1.0
file *
end #fileset
end #product

Control Scripts Files

This section contains control scripts for the qlisp driver.

Preinstall

This script checks if there are any older versions of qlisp driver sources already existing on the system. If so, it removes them all and continues with its installation.

#!/sbin/sh

# Product: QLISP11_23_1_0
# Fileset: qlispsrcfiles1_0
# preinstall
# @(#) $Revision: 1.1 $

###
#
# (c) Copyright Hewlett-Packard Company 2007
#
###
#
set up the SD environment
[[ -z "$SW_ROOT_DIRECTORY" ]] && SW_ROOT_DIRECTORY="/
[[ -z "$SW_PATH" ]] && SW_PATH="/usr/lbin/sw/bin:/usr/bin:/usr/ccs/bin"
[[ -z "$SW_SYSTEM_FILE_PATH" ]] && SW_SYSTEM_FILE_PATH="/stand/system"
exitval=0                   # Anticipate success
UTILS="/usr/lbin/sw/control_utils"
if [[ ! -f $UTILS ]]
then
  echo "ERROR: Cannot find $UTILS"
  exit 1
fi
$. UTILS
echo checking preinstall working
if swlist -l filesset B11_*QLISP
then
  echo Found older version QLISP
  echo Removing older version QLISP
  swremove B11_*23QLISP*
fi
exit $exitval

Postinstall

This script installs the driver into kernel by adding qlisp into the system file.

#!/sbin/sh
###
#
# Product: QLISP11_23_1_0
# Fileset: qlispsrcfile1_0
# postinstall
# @(#) $Revision: 1.1 $###
#
# (c) Copyright Hewlett-Packard Company 2007
#
######
UTILS="/usr/lbin/sw/control_utils"
if [[ ! -f $UTILS ]]
then
  echo "ERROR: Cannot find $UTILS"
  exit 1
fi
$. UTILS
exitval=$SUCCESS# Anticipate success
############################################################################
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

This section contains commands to package, register, install, manage, and remove the qlisp driver software.

Creating a Package

The following command creates a software package under /home/user/qlispdepots:

```
# swpackage -s qlisp.psf -d /home/user/qlispdrvdepots
```

Issue the `ls` command for the `/home/user/qlispdepots` directory to display the following:

```
# ls /home/user/qlispdepots
B11_23QLISP
QLISP11_23_1_0
catalog
swagent.log
```

Issue the following command for the `/home/user/qlispdrvdepots/QLISP11_23_1_0` directory to display the qlisp sources under directory specified in the `target dir path` across the directory fileset of the PSF:

```
# ls /home/user/qlispdrvdepots/QLISP11_23_1_0
```

Issue the following command for the `/home/user/qlispdrvdepots/catalog` directory to display an INDEX file with creation time and all information set in PSF:

```
# ls /home/user/qlispdrvdepots/catalog
```

Change to the following directory to display the pfiles (product files). They in turn have INDEX and INFO files, which are self-explanatory:

```
# cd B11_23QLISP
```

Change to the following directory to display the pfiles, which are the same as for B11_23DDK:

```
# cd QLISP11_23_1_0
```

Registering the Depot

Issue the following command to register the qlisp depot:

```
# swreg -l /home/user/qlisdrvdepots
```

Installing the Depot

Issue the following command to install the qlisp sources under /opt/ddk/11.23/sampldrvs/massstorage/qlisp/11.23.1.0:

```
# swinstall -s /home/user/qlispdrv/depots B11_23QLISP
```

Listing the Product

Issue the following command to list B11_23QLISP with title and revision information:

```
# swlist
```

Issue the following command to list the QLISP11_23_1_0 product:

```
# swlist -l product B11_23QLISP
```

Issue the following command to list qlispsrcfiles1_0:

```
# swlist -l fileset B11_23QLISP
```

Removing the Sources from the Product

Issue the following command to remove all files under /opt/ddk/11.23/sampldrvs/massstorage/qlisp/11.23.1.0/:

```
# swremove B11_23QLISP
```

Issue the following command to remove the files from the same location:

```
# swremove B11_23QLISP.QLISP11_23_1_0
```
Issue the following command to not show B11_23QLISP and product and filesets under it:

```
# swlist
```

Software Package Builder

Software Package Builder (SPB) provides a visual method to create and edit software packages using HP-UX Software Distributor (SD-UX) package format. The SPB GUI provides user interfaces into software package structure, showing attributes that can be set for each package element. SPB loads packaging policies and validates software package attributes against these policies. SPB CLI can also validate software package attributes against policies and can be added to an automated process for editing and validation of a PSF.

The SPB is available at:

http://www.software.hp.com/products/SPB/
A

100BT 100BASE-T is the technical term for the Fast Ethernet or IEEE802.3u standard. See also, Fast Ethernet.

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.

adapter card Physical hardware, under software control, that 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 (for example, a disk controller with disk).

ANSI American National Standards Institute

Area Allocator The memory attribute based allocated in HP-UX kernel which replace the old MALLOC and /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.

autoload A capability made possible by the DLKM feature. It 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. During an autoload, the kernel also loads any modules that the module being loaded depends upon, just as it does during a demand load.

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 (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 An atomic unit of data representing either a 0 or a 1.

bit mask A pattern of binary values, typically used in bitwise operations.

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.

CB-CDIO
Central Bus CDIO. The BN-CDIO that is responsible for discovering and initializing CEC components.

ccNUMA
Cache-Coherent Non-Uniform Memory Architecture. See also, NUMA.

CD
Command Descriptor Block

CDIO
Context Dependent I/O module. A module in the GIO framework that contains all bus-specific or driver environment specific functionality.

CKO
Checksum Offload

class
A logical grouping of device or hardware modules by type. For instance the tape class 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 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 that 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

D

Data Link layer
The second layer of the seven-layer OSI model, responsible for frame delivery across a single link.

datagram
(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.

Device Driver Environment (DDE)
A defined set of services and entry points that allow a driver to function.

DLKM
Dynamically Loadable Kernel Module. A feature available in HP-UX 11.0 that supports dynamic loading and unloading of kernel modules, to avoid wasting kernel memory by keeping modules in core when they are not in use.

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.
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>driver</td>
<td>Software module that controls a device, interface card or bus-nexus. See also, device driver</td>
</tr>
<tr>
<td>DSAP</td>
<td>Destination Service Access Point</td>
</tr>
<tr>
<td>E</td>
<td>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.</td>
</tr>
<tr>
<td>ENET</td>
<td>The name of an HP-UX sample Native STREAMS DLPI network interface driver.</td>
</tr>
<tr>
<td>Ethernet</td>
<td>The popular name for a family of LAN technologies standardized by IEEE 802.3.</td>
</tr>
<tr>
<td>F</td>
<td>Fast Ethernet: An Ethernet system operating at 100 Mb/s.</td>
</tr>
<tr>
<td>filesets</td>
<td>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.</td>
</tr>
<tr>
<td>fragmentation</td>
<td>A technique whereby a packet is subdivided into smaller packets so they can be sent through a network with a smaller MTU. See also, reassembly.</td>
</tr>
<tr>
<td>frame</td>
<td>The Data Link layer encapsulation of transmitted or received information.</td>
</tr>
<tr>
<td>Frame Check Sequence</td>
<td>A block check code used to detect errors in a frame. Most LANs use a CRC-32 polynomial as their FCS.</td>
</tr>
<tr>
<td>full duplex</td>
<td>A mode of communication in which a device can simultaneously transmit and receive data across a communications channel. See also, half-duplex.</td>
</tr>
<tr>
<td>G</td>
<td>Gigabit Ethernet: An Ethernet system operating at 1000 Mb/s.</td>
</tr>
<tr>
<td>GIO</td>
<td>General I/O System</td>
</tr>
<tr>
<td>group address</td>
<td>Synonymous with multicast address.</td>
</tr>
<tr>
<td>H</td>
<td>High Availability. 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.</td>
</tr>
<tr>
<td>half-duplex</td>
<td>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.</td>
</tr>
<tr>
<td>HBA</td>
<td>Host Bus Adapter</td>
</tr>
<tr>
<td>header</td>
<td>A protocol-specific field or fields that precede the encapsulated higher-layer data payload (for example, the MAC addresses in a Data Link frame). See also, trailer.</td>
</tr>
<tr>
<td>HP-DLPI</td>
<td>An HP implementation of the DLPI layer.</td>
</tr>
<tr>
<td>I/O adapter</td>
<td>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.</td>
</tr>
<tr>
<td>I/O bus</td>
<td>Interconnect bus for I/O cards and devices. For example, PCI.</td>
</tr>
<tr>
<td>I/O node</td>
<td>An element of an I/O tree that includes all relevant information needed for configuring a single hardware module.</td>
</tr>
<tr>
<td>I/O PDIR</td>
<td>I/O Page Directory. An 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.</td>
</tr>
<tr>
<td>I/O tree</td>
<td>Data structure for recording the I/O subsystem configuration information.</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ICMP</td>
<td>Internet Control Message Protocol</td>
</tr>
<tr>
<td>IELAN</td>
<td>The name of a HP-UX sample HP-DLPI (non-native) network interface driver.</td>
</tr>
<tr>
<td>IHV</td>
<td>Independent Hardware Vendor</td>
</tr>
<tr>
<td>ILP32</td>
<td>A C language data model where int, long, and pointer data types are 32 bits in size.</td>
</tr>
<tr>
<td>init function</td>
<td>This is an attribute in driver modules modmeta file. An initfunc statement specified an initialization function, provided by the module, that the system should call during driver initialization.</td>
</tr>
<tr>
<td>init list</td>
<td>A linked list of device driver init routines (drv_init) that is built as the drivers configure themselves and run as the I/O system configuration is completed. These routines perform any device driver-specific initialization.</td>
</tr>
<tr>
<td>instance</td>
<td>A number assigned to an I/O tree node. The number is unique within a driver class.</td>
</tr>
<tr>
<td>Interface Service Routine</td>
<td>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.</td>
</tr>
<tr>
<td>Interrupt Service Routine</td>
<td>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.</td>
</tr>
<tr>
<td>IOVA</td>
<td>I/O Virtual Address. An 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.</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>IPD</td>
<td>Installed Product Database. SD uses the IPD to keep track of the software 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.</td>
</tr>
<tr>
<td>IPF</td>
<td>Itanium Processor Family</td>
</tr>
<tr>
<td>IRQ</td>
<td>Interrupt Request</td>
</tr>
<tr>
<td>ISC</td>
<td>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.</td>
</tr>
<tr>
<td>ISCSI</td>
<td>SCSI over IP.</td>
</tr>
<tr>
<td>ISR</td>
<td>Interrupt Service Routine. A driver-specific routine that handles interrupts from the device.</td>
</tr>
<tr>
<td>ISV</td>
<td>Independent Software Vendor</td>
</tr>
<tr>
<td>J</td>
<td>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.</td>
</tr>
<tr>
<td>K</td>
<td>Kcweb. The HP-UX Kernel Configuration tool user interface that uses as a web browser.</td>
</tr>
<tr>
<td>kernel module</td>
<td>A section of code responsible for supporting a specific capability or feature. Normally, such code is maintained in individual object files and/or archives, enabling modules to be conditionally included or excluded from the kernel, depending on whether or not the features they support are desired.</td>
</tr>
</tbody>
</table>
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
A 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
Logical Volume Manager. This 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 (for example, 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 an 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.

mbblk
Message block. A data structure used in networking stack.

mbuf
Message buffer. A data structure used in mass storage stack.

MemoryMapped I/O (MMIO)
I/O that occurs by mapping the device’s I/O to system memory.

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.

module type
A module type is distinguished by the mechanism used to maintain the modules of that type within the kernel. DLKM modules are classified according to a fixed number of supported module types.

modwrapper
The additional code and data structures added to a DLKM module to make it dynamic.

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 (for example, 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.

**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**

**Online Addition and Replacement (OLA/R)**

The ability to insert adapter cards and replace such cards while a system is being used (Hot Plug).

**Online Deletion**

The ability to delete adapter cards.

**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 (a single instruction). Typical cards map their registers into PCI memory space, meaning they can only be accessed by PCI memory cycles.

**Perl**
A procedural programming language created by Larry Wall. Its ancestors include awk, sed, and C.

**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 in which 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

**quiesce**
A networking term. To render quiescent, or temporarily inactive or disabled. For example, to quiesce a device (such as a digital modem). It is also a system command in MAX TNT software which is used to “Temporarily disable a modem or DS0 channel”.

**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 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.
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDTR</td>
<td>Synchronous Data Transfer Request</td>
</tr>
<tr>
<td>Semicoherent I/O</td>
<td>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.</td>
</tr>
<tr>
<td>Series 700</td>
<td>HP9000/7XX family of PA-RISC workstations.</td>
</tr>
<tr>
<td>Series 800</td>
<td>HP9000/8XX family of PA-RISC business servers.</td>
</tr>
<tr>
<td>SGML</td>
<td>Standardized General Markup Language. This template contains macros that convert your document into SGML.</td>
</tr>
<tr>
<td>SIO</td>
<td>Server I/O. I/O environment for port-server drivers with origins in S/800 systems.</td>
</tr>
<tr>
<td>SNAP</td>
<td>Sub-Network Access Point</td>
</tr>
<tr>
<td>Software Distributor (SD)</td>
<td>The software distributor tool for HP-UX operation system.</td>
</tr>
<tr>
<td>Software Objects</td>
<td>Can be a bundle, product, or filesets.</td>
</tr>
<tr>
<td>spinlock</td>
<td>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.</td>
</tr>
<tr>
<td>Stream</td>
<td>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.</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>target</td>
<td>Either a host (the host's file system) or a depot that resides on a host.</td>
</tr>
<tr>
<td>TBI</td>
<td>Transaction Based Interrupt</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>Token Ring</td>
<td>A LAN whose MAC algorithm uses token passing among stations on a logical ring topology (IEEE 802.5).</td>
</tr>
<tr>
<td>topology</td>
<td>The physical or logical layout of a network.</td>
</tr>
<tr>
<td>trailer</td>
<td>A protocol-specific field or fields that follow the encapsulated higher-layer data payload (for example, the FCS in a Data Link Frame). See also, header.</td>
</tr>
<tr>
<td>type encapsulation</td>
<td>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.</td>
</tr>
<tr>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>virtual address</td>
<td>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.</td>
</tr>
<tr>
<td>W</td>
<td>W</td>
</tr>
<tr>
<td>WDTR</td>
<td>Wide Data Transfer Request</td>
</tr>
<tr>
<td>WSIO</td>
<td>Workstation and Server I/O. A CDIO, also the HP-UX Driver Development Environment (DDE).</td>
</tr>
</tbody>
</table>
XML eXtensible Markup Language. A subset of SGML (Standardized General Markup Language), this template contains macros that convert your document into XML.
Index

A
ABORT status parameter, 166
acquisition rule
lock, 73
adapter (see I/O adapter)
adaptive lock, 70
address
I/O virtual, 46
physical, 46
virtual, 46
address probe
parameters, 95
registering, 95
registration example, 96
application layer, 214
arena
creating, 57, 58
deallocating memory, 57
fixed size allocation, 57
fixed-sized example, 60
handle, 58
initializing, 57
kmem_arena_attr_t structure, 58
kmem_handle_t structure, 58
object caching, 61
overriding default attributes example, 60
removing, 57
types, 58
variable size allocation, 57
variable-sized example, 60
Arena Allocator
overview, 56
routines, 57
ARP
configuration, 303
formatting information, 366
asynchronous event
codes, 442
sending to SCSI services, 407
attribute, 52
(see also WSIO system attribute service)
and PSF, 517
bundle, 522–526
category, 517–522
fileset, 531–535
modifying for an interface, 56
product, 526–529
subproduct, 530
vendor, 517
autoload stub, 149
B
base address register
using in PCI, 201
base module
module metadata file, 150
unloading, 151
bdevsw, 34, 35
beta semaphore (see mutex)
big endian, 50
block device, 34
block device switch table (see bdevsw)
block driver
entry point, 86
block I/O, 34
and driver strategy routine, 120
data structure, 401
request flow, 402
blocking transaction, 198
BOOT status parameter, 166
boot support
adding to SCSI interface driver, 439
buf structure, 38, 129
buffered raw I/O, 34
bundle, 512
attributes, 517, 522–526
naming, 513
bundle object
sample, 554
template, 553
writing, 553
Cc
C_ALLCLOSES flag, 106
cache coherence
driver requirements, 45
overview, 45
callback function, 131
canput routine, 303
card instance data, 240
category attribute, 517–522
category object
sample, 553
template, 553
writing, 553
cCc command, 169
CDB
status values, 442
cdevsw, 34, 35
CDIO
components, 30
description, 30
Cell Local Memory (see CLM)
character device, 34
character device switch table (see cdevsw)
character driver
entry point, 87
character I/O, 34
ccheckinstall script
guidelines, 543–545
sample, 557
template, 557
writing, 557
CheckSum Offload (see CKO)
CKO, 314
   and dl_hp_neg_ops_one_t structure, 314, 377
   structure, 382
   types, 382
cko_info_t structure, 382
class directive, 139
class instance number
   obtaining, 55
class statement, 163
CLI, 573
client module
   module metadata file, 151
CLM, 496
class
   driver, 310
   special device file, 310
CLONEOPEN flag, 310
close system call, 86
coherent I/O system, 47
coherent platform, 45
Command Line Interface, 573
class
   command word
      ioctl system call, 111
   condition variable, 73
      routines, 74
   configuration file
      sample for IELAN driver, 288
   configuration routine
      and drivers, 85
      guidelines, 86
      writing, 87–102
   configuration space
      accessing resources, 44
      obtaining a handle, 50
      PCI, 197
      PCI restrictions, 203
      reading from, 50
      releasing a handle, 50
      writing to, 50
   configure script
      guidelines, 547–548
      sample, 560
      template, 560
      writing, 560
   constant
      and SCSI interface drivers, 405
      and writing drivers, 77
      error, 141
      KRS, 178
      PCI, 206
   constraint handler, 188
   constructor function, 62
   Context-Dependent I/O (see CDIO)
   control function, 310
      ENET driver lower half, 321
   control request
   processing, 250
   types, 251
   control script
      and commands, 540
      checkinstall, 557
      configure, 560
      environment variables, 538
      exit values, 542
      format, 537
      input actions, 541
      output actions, 541
      overview, 536
      postinstall, 559
      postremove, 563
      preinstall, 558
      preremove, 562
      required actions, 542
      standard functions, 543
      unconfigure, 561
      writing, 557–564
   control_utils library, 543
controller
   initializing for HBA, 419
   launching probe operations, 427
   parallel SCSI driver registration example, 422
   probing sequence, 421
   registering with SCSI services, 422
   registration sequence, 421
   setting the health attribute, 418
   unregistering, 429
   copyin kernel routine, 34
   counting semaphore, 74
      routines, 74
CPU
   migrating interrupts, 42, 495–509
CRA, 477
   and modprep script, 143
   and WSIO driver, 143
   example, 485
   flow examples, 487–494
   header file, 481
   infrastructure module, 477
   locked analysis, 485
   return flags, 480
   shared libraries, 481
   using nwmgr, 340
CRA log file
   and OL* operations, 485
   conventions, 483
   examples, 484
      from subsystem CRA modules, 482
   cra routine, 478
      input parameters, 481
   cra_ret_t return type, 480
CRA_TRACE_LOG environment variable, 483
Critical Resource Analysis (see CRA)
csema_alloc routine, 74
csema_attr_init routine, 74
csema_attr_setdata routine, 74
csema_dealloc routine, 75
  csema_decrement routine, 75
  csema_decrement_sig routine, 75
  csema_increment routine, 75
  csema_timeddecrement routine, 75
  csema_timeddecrement_sig routine, 75
  csema_trydecrement routine, 75
  csema_value routine, 75
  cv_alloc routine, 74
  cv_attr_init routine, 74
  cv_attr_setdata routine, 74
  cv_broadcast routine, 74
  cv_dealloc routine, 74
  cv_signal routine, 74
  cv_timedwait routine, 74
  cv_timedwait_sig routine, 74
  cv_wait routine, 74
  cv_wait_sig routine, 74

D
  d_ioctl entry point, 439
data
    buffering using uiomove, 109
caching, 34
data header, 380
      (see also OOP)
data link layer, 215
Data Link Provider Interface (see DLPI)
Data Link Service Definition, 301
data path
  Fastpath, 312
  inbound, 312
  outbound, 312
  regular, 312
data structure (see structure)
    and NIC tool, 394
data template, 380
data transfer, 34
data tuple, 380
datagram
  processing fragments, 385
DDE, 399
deadlock, 73
debug information
  embedding in kernel modules, 164
dependency directive, 138
dependency statement, 160
depot
  directory, 511
  installing for qlisp, 572
  registering for qlisp, 572
tape, 511
desc field, 142, 183
desc statement, 158
description service (see WSIO description service)
destructor function, 62
device
  asynchronous discovery, 433
  block, 34
directive
  class, 139
dependency, 138
  flags, 139
  initfunc, 148
  loadtimes, 137, 139
  states, 139
  struct drv_info, 139
type, 139
  unloadable, 139, 141
directory depot, 511
disk driver
  required header files, 78
Disk Space Analysis, 565
dispatch point, 161
early_boot_load, 162
tunable_init, 162
DL_ENABMULTI primitive, 256
dl_hp_create_info_t structure, 242
DL_HP_GET_DRV_PARAM_IOCTL ioctl, 255
DL_HP_GET_MIBSTATS_REQ control request, 283
dl_hp_neg_ops_one_t enumeration, 314
dl_hp_neg_ops_one_t structure, 377
DL_HP_NOTIFY_EVENT primitive, 379
DL_HP_OOP_PRESENT flag, 383
DL_LINK_DOWN_IND primitive, 379
DL_LINK_UP_IND primitive, 379
driver
  installation example, 445
  installing, 445
  listing information, 445
  major number, 34
  minor number, 34
  removing, 446
device special file
  and I/O command shared library, 447
  command flow, 446
  creating, 35, 446
  installation example, 445
  installing, 445
  listing information, 445
  major number, 34
  minor number, 34
  removing, 446
device switch table, 35
Direct Memory Access (see DMA) (see DMA)
directive
  class, 139
dependency, 138
  flags, 139
  initfunc, 148
  loadtimes, 137, 139
  states, 139
  struct drv_info, 139
type, 139
  unloadable, 139, 141
directory depot, 511
disk driver
  required header files, 78
Disk Space Analysis, 565
dispatch point, 161
early_boot_load, 162
tunable_init, 162
DL_ENABMULTI primitive, 256
dl_hp_create_info_t structure, 242
DL_HP_GET_DRV_PARAM_IOCTL ioctl, 255
DL_HP_GET_MIBSTATS_REQ control request, 283
dl_hp_neg_ops_one_t enumeration, 314
dl_hp_neg_ops_one_t structure, 377
DL_HP_NOTIFY_EVENT primitive, 379
DL_HP_OOP_PRESENT flag, 383
DL_LINK_DOWN_IND primitive, 379
DL_LINK_UP_IND primitive, 379
character, 34
getting swap partition size, 128
I/O memory, 64
queue management, 129
raw, 34
setting up for PCI, 201
device driver, 33, 77
(see also driver)
driver_install routine example, 89
interface management, 129
interrupt driven routines, 129
sample header file, 82
Device Driver Environment (see DDE)
device interrupt, 123
device special file
  and I/O command shared library, 447
  command flow, 446
  creating, 35, 446
  installation example, 445
  installing, 445
  listing information, 445
  major number, 34
  minor number, 34
  removing, 446
device switch table, 35
Direct Memory Access (see DMA) (see DMA)
DLKM, 137
   adding support in kernel modules, 164
   and DMA services, 49
   and ENET driver, 331
   and SCSI interface driver, 415
   data structure, 139
   destructor function caution, 63
   driver details, 139
   writing a driver, 137–156
DLPI, 301
   infrastructure, 213
   primitives, 313
dlpi_eventp routine, 274
DLPI_GET_SRC_ROUTE_FLAG ioctl, 316
DLPI_HP_GET_DRV_PARAM_IOCTL ioctl, 316
DLPI_HP_SET_DRV_PARAM_IOCTL ioctl, 316
DLPI_IOC_HDR_INFO ioctl, 313
DLPI_NO_SRC_ROUTING ioctl, 316
dlpi_propp routine
   and HP-DLPI drivers, 233
   and native DLPI LAN driver, 300
   guidelines, 234
   prototype, 233
   return values, 234
DLPI_SET_NOLOOPBACK ioctl, 316
DLPI_SET_SRC_ROUTE_FLAG ioctl, 316
dlpi_wakeup routine, 256
DLS provider
   and transport layer, 377
   ATM, 377
   HyperFabric, 377
   X.25, 377
DLSAP, 326
DMA, 45, 399
   and HP-DLPI LAN drivers, 223
   and PCI drivers, 204
   flags, 408
   mapping, 46
   mapping services, 408
   retrieving data packet information, 266
DMA object
   programming considerations, 47
DMA service (see WSIO DMA service)
dma_sync_IO routine
   rules, 46
driver, 33, 77
   (see also device driver)
   (see also disk driver)
   (see also interface driver)
   (see also LAN driver)
   adding CRA support, 477–494
   adding DLKM support, 137–156
   adding I/O command support, 445–454
   and kernel recognition, 34
   and major numbers, 34
   choosing a name, 77
   coherency requirements, 45
   configuration overview, 88
   defining data structures, 78–85
DLMI details, 139
drv_ops_t fields, 79
environment, 29–32
event handler example, 134
event handling, 51
handling device interrupts, 123
identifying routines, 85
interface, 33
interface with system calls, 86
interrupt context, 36
interrupt service, 51
LBI, 495
loosely coupled, 213
lower half, 36
memory allocation, 51
monolithic, 33
MSI, 495
MSI-X, 495
nwmgr routines, 341–350
obtaining options from, 377
OL* scripts, 469
probe parameters, 94
protection, 301
pseudo, 33
required header files, 78
resume routine, 465
sample (PCI), 207
synchronization, 301
TBI, 495
tightly coupled, 213
types of, 33–34
upper half, 36
writing, 77–135
writing attach routines, 91–92
writing callback routines, 115
writing configuration routines, 87–102
writing driver_close routines, 105
writing driver_dev_init routines, 92
writing driver_if_init routines, 92
writing driver_ioctl routines, 110
writing driver_minor_build routines, 102
writing driver_minphys routines, 116
writing driver_open routines, 102
writing driver_read routines, 107
writing driver_select routines, 117
writing driver_write routines, 107
writing entry point routines, 102–123
writing for SCSI interfaces, 405–444
writing install routines, 89–91
writing probe routines, 93
driver entry point, 217
Driver Entry Table
   and I/O commands, 447
driver event
   receiving from HP-DLPI, 276
   reporting to HP-DLPI, 274
types, 275
driver link state
   reporting to HP-DLPI, 274
driver property
reporting changes, 275

driver registration
additional address probe function, 95
address probe function, 93
event handler, 130
event mask, 132
OL* event handler, 458
OL* event mask, 459
driver startup script, 287
arguments, 287
guidelines, 287
IELAN driver example, 288
driver statement, 163
driver_addr_probe routine, 96
driver_attach routine
and HBA controllers, 416
and HP-DLPI driver, 238
and HP-DLPI drivers, 236
and native DLPI LAN driver, 297
diagnostics, 91
element, 91, 133
PCI driver example, 209
prototype, 238
return value, 91
writing, 91
driver_callback routine, 115
example, 116
driver_close routine
C_ALLCLOSES flag, 106
example, 106
exclusive open device, 105
multiple open device, 105
shared open device, 105
writing, 105
driver_control routine
description, 249
HP-DLPI driver guidelines, 251
request types, 251
driver_dev_init routine
writing, 92
driver_event_handler routine, 276
driver_if_ctlr_probe routine, 403
driver_if_init routine
example, 126
writing, 92
driver_if structure, 320
driver_inboundp routine, 264
driver_init routine
and HP-DLPI driver, 239
and HP-DLPI drivers, 236
and native DLPI LAN driver, 298
driver_install dispatch point, 162
driver_install routine, 89–91
and HP-DLPI drivers, 236, 237
device driver example, 89
example, 131
interface driver example, 90
monolithic driver example, 90
non-WSIO pseudo driver example, 90
PCI driver example, 208
SCSI interface driver example, 414
WSIO pseudo driver example, 89
driver_ioctl routine
example, 113
prototype, 118
writing, 110
driver_isr routine, 87
and HP-DLPI driver, 261
example, 126
writing, 123
driver_minor_build routine, 102
driver_minphys routine, 87, 116
example, 117
driver_open routine
and native DLPI LAN driver, 310
example, 104
syntax, 103
writing, 102
driver_ops_t structure
and native DLPI LAN driver, 314
driver_read routine
using physio routine, 108
writing, 107
driver_select routine
example, 118
writing, 117
driver_strategy routine, 87
example, 122
using for read operations, 121
using for write operations, 121
writing for block device, 120
driver_write routine
using uiomove routine, 109
writing, 107
drv_cb_info_t structure, 116
drv_cb_opcode_t structure, 116
drv_info_t structure, 80
and HP-DLPI LAN drivers, 229
and major numbers, 34
drv_ops_t structure, 79
and HP-DLPI LAN drivers, 229
fields, 79
DSA, 565
dynamic load function
native DLPI driver, 147
STREAMS driver, 146
STREAMS module, 148
WSIO driver, 143
dynamic load initialization
miscellaneous module, 149
STREAMS driver, 146
STREAMS module, 148
WSIO driver, 143
dynamic unload function
miscellaneous module, 149
native DLPI driver, 147
no PCI online deletion support, 144
PCI online deletion support, 144
STREAMS driver, 146
STREAMS module, 148
Dynamically Loadable Kernel Module (see DLKM)

E
End-of-Options (see EOO)
END_MODULE macro, 152
enet, 372
ENET driver, 296
(see also native DLPI LAN driver)
adding DLKM support, 331
DLKM data structure examples, 332
DLPI ioctls used, 317
DLPI primitives used, 317
driver_attach routine example, 298
driver_install routine example, 297
driver_load example, 333
driver_open routine, 310
driver_release_all example, 334
driver_unload example, 334
drv_info structure example, 296
entry points for DLKM, 333
ift data structure fields, 320
inbound data path, 324
initializing data structures example, 296
initializing statistics, 330
lower part, 318
NIC tool shared library example, 393
nwmgr shared library example, 350
outbound data path, 323
processing inbound packets, 331
processing outbound packets, 331
statistics primitives, 331
subformatter example, 367
upper part, 306
wakeup routine example, 307
ENET Native STREAMS DLPI, 302
enet_dlpi_data_t structure, 308, 309
ENOLINK, 268
entry point
and block drivers, 86
and character drivers, 87
and drivers, 85
and interface drivers, 87
description, 86
for SCSI interface drivers, 433
entry point routine
writing, 102–123
environment variable
and CRA log, 483
and subformatter message catalog, 373
in SD control scripts, 538
EOO, 380
error code, 351
error constant, 141
error message
writing in a tunable handler, 187
escsi_ifsw_t structure
tentry points, 433
fields, 434
escsi_list_t structure, 410
ESCSI_TP_OFFLINE event, 442
ESCSI_TP_ONLINE event, 442
event
and NIC tool, 388
log types, 357
posting for I/O node, 55
setting timeout values, 132
event flag
LBI, 497
MSI, 502
MSI-X, 502
TBI, 499
event handler
example, 134
HP-DLPI LAN driver, 276
IELAN driver example, 277
registering, 130
registering for OL*, 458
rules for HP-DPLI driver, 277
event handling function
OL*, 460
event handling service (see WSIO event handling service)
event mask
registering for driver, 132
registering for OL*, 458
exclusive open, 103
exports statement, 161
F
FAIL status parameter, 165
Fastpath, 312, 377, 378
negotiation acknowledgment message format, 378
negotiation message formats, 313
negotiation request message format, 378
operation, 216
Fastpath stream, 313
FC, 399
Fibre Channel (see FC)
fileset, 512
attributes, 517, 531–535
corequisite, 537
file definitions, 535
kernel, 537
naming, 514
prerequisite, 537
reboot, 537
fileset object
sample, 556
template, 556
writing, 556
flag
KRS, 177
flags directive, 139
flags statement, 163
formatter
types of routines, 365
utility routines, 365, 366
fragment chain, 385
frame
inbound processing, 264
outbound processing, 268
function
mod_einval, 152
mod_enoload, 152
mod_minus, 152
fuser command, 143

G
General I/O (see GIO)
generic function switch, 39
GIO
and WSIO, 53
description, 29
information flow, 29

H
HA event, 444
hardware path
converting from a string, 55
converting to a string, 55
mapping to node, 55
retrieving for a node, 55
hardware state
notifying HP-DLPI, 276
HBA, 399
HBA controller
configuring, 416
initializing, 416
header file
KRS, 178
required for drivers, 77
sample for device driver, 82
sample for interface driver, 83
sample for pseudo driver, 84
SCSI interface driver requirements, 412
health attribute, 418
high availability (see HA)
Host Bus Adapter (see HBA)
HP-DLPI, 213
 infrastructure, 216
 infrastructure component, 215
 supported NICs, 213
HP-DLPI driver, 235
(see also IELAN driver)
64-bit MIB statistics, 254
and LAN commands, 230
and nwmgr command, 230
driver_control guidelines, 251
handling a primitive, 251
handling loopback mode, 272
handling promiscuous mode, 272
initializing, 236–249
ioctl processing, 254
ISR, 261
optional capabilities, 227
processing inbound frames, 264
processing outbound frames, 268
receiving from HP-DLPI, 276
reporting driver link state changes, 274
reporting hardware state changes, 276
required data structures, 253
required primitives, 252
updating inbound MIB statistics, 265
updating outbound MIB statistics, 269
WSIO interaction, 228
HP-DLPI LAN driver
adding nwmgr command support, 292
adding OLARD support, 285
data link architecture, 217
dlpi_propp routine guidelines, 234
reporting driver property changes, 275
required steps, 223
sending frame to NIC, 270
HP-UX Software Distributor
SD-UX, 573
HyperFabric, 377
I
ICOD, 507
ICS, 69
IELAN driver, 235
(see also HP-DLPI driver)
DL_HP_SET_DRV_PARAM_IOCTL example, 258
driver_attach routine example, 238
driver_init routine example, 240–248
driver_install example, 237
driver_isr routine example, 265
driver_load example, 237
event handler example, 277
freeing multicast address list, 235
initializing data structures, 236
NIC tool shared library example, 393
outbound processing example, 269
processing DL_ENABMULTI primitive, 256
sample configuration file, 288
sample ISR, 262
sample module metadata file, 248
sample startup script, 288
ielan_if_t structure, 231
IELAN_LINK_SPEED ioctl, 260
if_ctlr_probe routine, 427
if_ioctl entry point, 438
if_task_mgmt entry point, 436
IHW, 301
inbound data path, 312
inbound frame
IELAN driver example, 265
processing, 264
processing guidelines, 264
inbound processing
statistics, 282
Independent Hardware Vendor (see IHV)
Independent Software Vendor (see ISV)
infrastructure CRA module, 477
inheritance
  KRS, 177
initfunc directive, 148
initfunc statement, 161
insf command
  description, 445
  function prototype, 449
Installed Product Database (see IPD)
Instant Capacity on Demand (see ICOD)
intctl command, 495
interface
  card management data structure, 130
  claiming, 54
  converting hardware path to a string, 55
  converting string to hardware path, 55
  creating a new property, 55
  destroying (legacy), 54
  destroying all properties, 56
  destroying an property, 56
  function switch, 39
  legacy, 54, 55
  mapping hardware path to node, 55
  modifying an property, 56
  obtaining a reference, 54
  obtaining class instance number, 55
  obtaining information (legacy), 54
  obtaining isc structure, 55
  posting events, 55
  registering, 54
  registering (legacy), 54, 55
  registering with WSIO, 53
  retrieving a property, 56
  retrieving hardware path, 55
  retrieving the property size, 56
  saving a reference, 54
  select code table, 38
interface driver, 33
  driver_install routine example, 90
entry point, 87
error recovery, 470
sample header file, 83
interface management
  device driver, 129
Interface Select Code (see ISC)
Internet protocol
  ARP, 215
  TCP/IP, 215
  UDP/IP, 215
interrupt, 51
  (see also WSIO interrupt service)
  handling, 41
  handling in drivers, 129
  line based, 495
  message signalled, 495
  migrating for LBI, 508
  migrating for MSI, 508
  migrating for TBI, 508
  migrating to another CPU, 42
  ordered, 52
  services, 41
  transaction based, 495
interrupt context
  driver, 36
Interrupt Control Stack (see ICS)
interrupt line sharing, 496
interrupt migration, 495
  flow, 507
Interrupt Request (see IRQ)
Interrupt Request Routine (see IRR)
Interrupt Service Routine (see ISR)
I/O, 52
  (see also WSIO I/O synchronization service)
  block, 34
  character, 34
  coherent system, 47
  completion handling, 442
  flow in mass storage stack, 402
  flow of a request, 37
  high-priority, 436
  noncoherent system, 47
  PDIR, 47
  raw, 34
  read request example, 37
I/O adapter, 47
I/O address
  obtaining physical address example, 65
I/O card
  obtaining physical address example, 65
I/O memory
  kernel mapping routines, 64–67
  mapping to virtual address, 64
  obtaining physical I/O address example, 65
  user process-mapping guidelines, 64
  write coalescing, 66
I/O node
  setting the health attribute, 418
  start interface routine, 435
  target path callback, 430
I/O object
  registering, 443
I/O port space, 44
  (see also WSIO I/O port space service)
  accessing, 44
I/O space, 198
I/O subsystem, 29
I/O switch table, 39
I/O system call, 36
I/O tree
  setting the node description, 52
I/O virtual address (see IOVA)
  io_get_instance routine, 55
  io_hw_path_to_node routine, 55
  io_hw_path_to_str routine, 55
  io_node_to_hw_path routine, 55
  io_post_event routine, 55
  io_post_event_req_t structure, 40
  io_str_to_hw_path routine, 55
ioctl
  and ENET driver, 317
  processing in HP-DLPI driver, 254
  required for boot support, 439
  transparent, 255
ioctl system call, 86, 110
  command words, 111
  LP64 considerations, 112
IODC, 201
iomap_enable_wc routine, 64
iomap_enter_shared_acc routine
deprecation statement, 64
iomap_exit_shared_acc routine
deprecation statement, 64
iomap_pagesize routine, 64
IOPDIR, 49
ioscan command, 143
IOVA
  translation, 47
  view, 46
iovec structure, 39
IP
  configuration, 303
  formatting information, 366
IP header, 384
IPD, 538
IRQ, 42, 202
ISC, 38, 202
  and WSIO, 31
isc_table_type structure, 38
  interface management, 130
ISO 8802/2, 301
ISO 8886, 301
ISR, 37, 199, 222
  and native DLPI LAN driver, 298, 326
  and spinlocks, 71
  HP-DLPI LAN driver example, 262
  IELAN driver example, 262
  rules, 261
isrlink routine
deprecation note, 496
isrunlink routine
deprecation note, 496
ISV, 301
K
  kcmdmodule command, 170
  use in SD control script, 171
  kctune command
and new tunables, 195
KDT
  and GIO, 29
KEN_BACKOUT reason code, 185
KEN_DONE reason code, 186
KEN_EVENT reason code, 185
KEN_REGISTER reason code, 185
KEN_UNREGISTER reason code, 185
kernel
  building, 195
  data structures, 38
  identifying device drivers, 34
  testing, 195
Kernel Device Table (see KDT)
kernel memory
  dynamic allocation, 56
kernel module
  adding DLKM support, 164
  adding tunable parameter support, 164
  adding tunable parameters, 183–195
  additional information, 163
  building, 167
  changing tunable parameter attributes, 195
  changing tunable parameter default values, 195
  creating a module metadata file, 157
  creating modprep scripts, 165
  declaring initialization functions, 161
  declaring tunable parameters, 161
  description, 158
  embedding debug information, 164
  embedding version information, 164
  exporting interfaces, 161
  installing, 170
  load sequence, 138
  load time, 160
  name, 158
  specifying capabilities, 157
  specifying characteristics, 157
  specifying dependencies, 160
  specifying dispatch points, 161
  specifying driver class, 163
  specifying driver flags, 163
  specifying driver structures, 163
  specifying driver type, 163
  state, 158
  testing, 170
  type, 158
  unloading, 160
  updating an older version, 171
  using, 170
  version, 158
  writing, 157–171
  writing source code, 164
Kernel Registry Services (see KRS)
kern_iomap routine, 64
kern_iounmap routine, 64
key
  KRS, 174
kfree routine, 63
Index

kget_log_instance routine, 361
KLOG_CK macro, 361
klogg_write routine, 361
kmalloc routine, 63
kmem_arena_alloc routine, 57
kmem_arena_attr_init routine, 58
kmem_arena_create routine, 57
kmem_arena_destroy routine, 57
kmem_arena_varalloc routine, 57
kr_close_node routine, 181
kr_delete_node routine, 181
kr_delete_value routine, 179
kr_flag_field_t structure, 178
kr_flags_t structure, 178
kr_get_mod_time routine, 181
kr_get_node_info routine, 180
kr_get_node_names routine, 180
kr_get_value routine, 179
kr_get_value_names routine, 180
kr_get_vinfo routine, 180
kr_ket_t structure, 178
kr_link_node routine, 179
kr_linkid_t structure, 178
kr_open_node routine, 178
kr_release_reference, 181
kr_set_node_flags routine, 180
kr_set_value routine, 179
kr_set_value_flags routine, 180
kr_size_t structure, 178
kr_type_t structure, 178
KRS, 173
  action flags, 177
  attribute flags, 177
  data structures, 178
  header file, 178
  inheritance, 177
  keys, 174
  link, 175
  persistence, 176
  tree reference, 176
  tree structure, 173
  value type, 174
KRS node
  and links, 175
  closing, 181
  deleting, 181
  deleting a value, 179
  getting a value, 179
  getting information, 180
  getting node names, 180
  getting time information, 181
  getting value information, 180
  getting value names, 180
  linking, 179
  opening, 178
  releasing a reference, 181
  setting a value, 179
setting node flags, 180
setting value flags, 180
structure, 174
krs.h header file, 178
kte_flags structure field, 186
kte_op structure field, 186
kte_tuneid structure field, 186
kte_txnid structure field, 186
KTRC_CK macro, 358
ktrc_write routine, 359
ktune_event_t structure, 186
ktune_register_handler function, 194
ktune_simple_dynamic routine, 196
LAN command
  and native DLPI LAN driver, 327
LAN driver
  creating NIC tool shared library, 393
  logging, 353
  NIC tool requirements, 393
  tracing, 353
lanadmin command, 337
LANG environment variable, 373
lanscan command, 337
LBI, 495
  and driver_isr routine, 124
  event flags, 497
  migrating interrupts, 508
LBI driver, 495
Least Significant Byte (see LSB)
  legacy device
    hardware path support, 431
Line Based Interrupt (see LBI)
  line sharing
    interrupt, 496
  link
    KRS, 175
    Link Layer Header template, 378, 379
    link state, 275
    linkloop command, 337
    little endian, 50
    LLC, 301
      specifying in ISR, 304
    LLC header
      stripping off inbound packets, 305
    LLC Type 1, 216
    LLC Type 2, 216
    load function, 141, 143
    STREAMS driver sample, 156
    WSIO driver sample, 155
load routine
  SCSI interface driver example, 415
load sequence, 138
LOAD status parameter, 165
loadtimes directive, 137, 139
loadtimes statement, 160
lock
  acquisition rules, 73
adaptive, 70
deadlock, 73
order, 73
locked analysis, 478
log data
  formatting, 355
logging (see network logging)
Logical Link Control (see LLC)
Logical Volume Manager (see LVM)
loopback mode, 272
loosely coupled driver (see native DLPI LAN driver)
LP64 data model
  and the ioctl system call, 112
LSB, 200
lsdev command, 35
lssf command
  description, 445
  function prototype, 449
LUN device
  controller management support, 433
lunpath, 403
  closing, 435
  legacy hardware support, 431
  opening, 434
LVM, 33, 80
M
  macro
    END_MODULE, 152
    MODULE, 151
    STUB, 151
    USTUB, 152
    WSTUB, 152
  major number
    and drivers, 34
    defined, 34
    dynamic assignment of, 34
    in device special files, 34
    range, 34
  Makefile file, 168
  Makefile.bld file, 168
  Management Information Base (see MIB)
  mapping
    I/O Space, 43
  mass storage stack
    components, 397
    data structures, 400
    features, 397
    I/O flow, 402
    probe functions, 403
  MAXNOCLOSEOPEN flag, 310
  MAXOPENS flag, 310
  mblk, 217, 228, 302
  MBLK chain
    and ioctl, 254
    and primitives, 252
  memory, 56
    (see also kernel memory)
  memory allocation, 43
  (see also WSIO memory allocation service)
  memory allocation service (see WSIO memory allocation service)
  memory mapped I/O
    write coalescing, 66
  memory space, 197
  message block (see mblk)
  message block function
    and HP-DLPI LAN drivers, 228
    and native DLPI LAN driver, 302
  Message-Signaled Interrupt (see MSI)
  metadata file (see module metadata file)
  MIB
    32-bit statistics, 277
    64-bit statistics, 279
    64-bit statistics support, 254
    getting statistics, 283
    resetting statistics, 284
    updating inbound statistics, 265, 267
    updating outbound statistics, 269
  mib_ifEntry structure, 328
    fields, 278
    prototype, 278
  migration
    interrupt, 495
  minor number
    defined, 35
    in device special files, 34
  miscellaneous module
    dynamic load data structure, 148
    dynamic load function, 149
    dynamic load initialization, 149
    dynamic unload function, 149
  mknod command, 35
  mksf command
    description, 446
    function prototype, 449
  ml_ops field, 140
  ml_type field, 140
  mod_einval function, 152
  mod_enload function, 152
  mod_minus function, 152
  mod_wsio_attach_list_add function, 143
  mod_wsio_attach_list_add routine, 333
  mod_zero function, 152
  modlink structure, 140
    and dynamic loading, 145
    and STREAMS modules, 147
  modmeta (see module metadata file)
  modprep script
    creating, 165
    naming convention, 170
    postload operation, 142
    postunload operation, 143
    preload operation, 142
    preunload operation, 142
    sample, 166
    status parameters, 165
  MODREV, 140
module (see kernel module)
  automatic loading, 139
  dependency, 138, 153
dynamically loadable, 137
load function, 141
load sequence, 138
load time, 137
static, 137
stub, 150
unload function, 141
MODULE macro, 151
module metadata file, 157
  adding a tunable parameter, 185
  and tunable parameters, 183
base module, 150
client module, 151
creating for HP-DLPI driver, 248
fields, 158–163
sample, 163
stub module, 150
module preparation script (see modprep script)
modwrapper structure, 139
monolithic driver, 33
driver_install routine example, 90
Most Significant Byte (see MSB)
  and native DLPI LAN driver, 302
  and PCI drivers, 206
  safe for OL*, 462
MSB, 200
MSI
  and driver_isr routine, 125
event flags, 502
  migrating interrupts, 508
MSI driver, 495
MSI-X
event flags, 502
MSI-X driver, 495
mtd_info field
  and miscellaneous module, 148
  and STREAMS driver, 145
  and STREAMS module, 147
  and WSIO driver, 142
mtd_pdata field
  and miscellaneous module, 148
  and STREAMS driver, 145
  and STREAMS module, 147
  and WSIO driver, 142
multicast address list
  freeing, 235
  multicast entry, 322
multiple open, 103
multiprocessing, 69
  compared to uniprocessing, 69
  synchronization mechanisms, 69
mutex, 71
deadlock, 73
  routines, 71
mutex_alloc routine, 71
mutex_attr_init routine, 71
mutex_attr_setflag routine, 71
mutex_dealloc routine, 71
mutex_lock routine, 71
mutex_owned routine, 71
mutex_trylock routine, 71
mutex_unlock routine, 71
mw_conf_data field, 140
mw_halt field, 140
mw_load field, 140
mw_modlink field, 140
mw_rev field, 140
mw_unload field, 140
N
  native DLPI driver
    dynamic load data structure, 147
dynamic load function, 147
dynamic unload function, 147
  native DLPI LAN driver, 296, 377
    (see also ENET driver)
    adding nwmgr support, 337–351
    and LAN commands, 327
    and NetTL, 327
    and nwmgr command, 327
    and SMH, 327
    and STREAMS functions, 303
    control functions, 310
    data link architecture, 218
displaying driver statistics, 327
dlpi_propp routine guidelines, 300
driver_install routine, 296
  initializing, 295
  ISR, 326
  registering with HP-DLPI, 300
  required steps, 293
Native Language Support (see NLS)
etfmt command, 355
  features, 364
NetTL, 217
  and HP-DLPI LAN drivers, 217
  and native DLPI LAN driver, 327
nettl command, 355
nettlconf command
  configuring a subsystem, 372
description, 354
  registering subformatter with NetTL, 364
nettlgen.conf file, 354
  network
    classifying trace data, 356
descriptor, 356
  network logging, 353
  assigning subsystem ID, 355
network driver (see LAN driver)
Network Interface Card (see NIC)
network interface driver
  architecture, 213
network logging, 353
  assigning subsystem ID, 355
code example, 362
defining a log time, 357
defining log events, 357
ENET driver subformatter example, 367
formatting data, 355
formatting messages, 364
guidelines, 355
nice formatted output example, 369
raw formatted output example, 370
registering subformatter, 364
required header files, 358
terse formatted output example, 370
testing support, 375
types of events, 357
network protocol layer, 215
network tracing, 353
(see also NetTL)
assigning subsystem ID, 355
classifying data, 356
code example, 360
ENET driver subformatter example, 367
formatting data, 355, 356
formatting messages, 364
formatting routines, 366
guidelines, 355
nice formatted output example, 369
raw formatted output example, 370
registering subformatter, 364
required header files, 358
terse formatted output example, 370
testing support, 375
Network Tracing and Logging (see NetTL)
networking stack
  transferring data within, 377
newconfig directory, 543
newconfig function, 543
NIC, 213
  configuration, 287
displaying information, 339
initializing card instance data, 240
modifying attributes, 339
performing CRA, 340
resetting, 276, 340
troubleshooting problems, 340
NIC tool
  adding support to HP-DLPI LAN drivers, 291
  and shared library, 388
  APIs, 394
data structures, 394
  ENET driver shared library example, 393
  events, 388
  IELAN driver shared library example, 393
LAN driver requirements, 393
return values, 392
starting, 387
user actions, 388
NLS, 372
NLSPATH environment variable, 373
node
  KRS, 174	node description, 52
  non-blocking transaction, 198
  non-WSIO pseudo driver
driver_install routine example, 90
noncoherent I/O system, 47
noncoherent platform, 45
NOSYNC
  and native DLPI LAN drivers, 302
nwmgr command, 303
adding support to HP-DLPI LAN drivers, 292
adding support to LAN drivers, 337–351
and HP-DLPI drivers, 230
and native DLPI LAN driver, 327
architecture, 338
building the shared library, 351
community flow, 337
compiling the shared library, 351
displaying help, 341
displaying NIC information, 339
displaying statistics, 327
displaying subsystem information, 339
error codes, 351
installing the shared library, 351
modifying NIC attributes, 339
performing CRA, 340
resetting a NIC, 340
resetting a subsystem, 340
routines for drivers, 341–350
shared library, 341–350
syntax, 338
troubleshooting NIC problems, 340
writing a shared library, 350

O
object caching
  overview, 61
OL*
  adding CRA support, 477–494
  addition and HP-DLPI, 285
  addition operation, 462
  and HP-DLPI LAN driver, 285
delete failures, 468
delete operation, 466
delete processing steps, 466
deletion and HP-DLPI, 286
driver requirements, 455, 461
driver support for, 130
event handling function, 460
handling, 462
miscellaneous driver changes, 461
MP safe, 462
overview, 455
postdelete phase, 466
predelete phase, 466
registering driver event handler, 458
registering driver event mask, 459
replacement and HP-DLPI, 285
replacement operation, 463
resource allocation failure, 462
resource allocation services, 463
resume request, 465
scripts, 469
suspend event, 463
tools, 462
WSIO Event Handling structures, 456
olrad command
and driver scripts, 469
Online Addition, 455
(see also OL*)
online addition (see OL*)
Online Addition and Replacement (see OL*)
Online Addition, Replacement, and Deletion (see OL*)
Online Deletion, 455
(see also OL*)
online deletion (see OL*)
Online Replacement, 455
(see also OL*)
online replacement (see OL*)
OOP, 314, 378
data template, 379
OOP data
inbound processing, 384
location in packet, 380
outbound processing, 383
skipping, 384
OOP data header, 380
OOP data template, 380
OOP data tuple, 380
open
exclusive, 103
multiple, 103
shared, 103
open system call, 86
operation
postload, 142
postunload, 143
preload, 142
preunload, 142, 143
option negotiation, 314
OSF/Encore spinlock, 301
OSI protocol suite, 215
OTHERQ routine, 303
Out-Of-Packet (see OOP)
Out-of-Packet (see OOP)
outbound data path, 312
outbound frame
IELAN driver example, 269
processing, 268
processing guidelines, 268
outbound processing
statistics, 283
packet
creating for qlisp, 572
inbound processing, 330
outbound processing, 331
packet header template, 380
packet train, 385
Page DIREctory (see PDIR)
parallel SCSI (see pSCSI)
parallelism
elsewhere, 302
global, 302
module, 302
NOSYNC, 302
queue, 302
queue pair, 302
PATH environment variable, 538
PCI
and DMA, 204
automatic IRQ determination, 202
blocking transaction, 198
byte swapping, 200
card mastered reads to host memory, 199
card mastered writes to host memory, 199
configuration space, 197
configuration space restrictions, 203
constants, 206
data structures, 206
device operation, 203
device setup, 201
driver entry point routines, 211
endian issues, 200
I/O base registers, 202
I/O space, 198
interleaved transactions, 199
mapping a memory base register, 202
masters and coherence, 205
memory base registers, 201
memory space, 197
multiprocessor safety, 206
non-blocking transaction, 198
PCI_LITTLE_ENDIAN_ONLY flag, 203
pre-swapping, 200
processor mastered read, 198
processor mastered write transaction, 198
register space, 197
sample driver, 207
sample driver_attach, 209
sample driver_install, 208
summary of services, 205
transaction ordering, 198
using base address registers, 201
write side effects, 199
PCI driver
writing, 197–211
PCI Error Recovery
overview, 470
PCI OLAR (see OL*)
PDC, 201
PDIR, 47
percentage_tune flag, 184
persistence
KRS, 176

P
package
creating for qlisp, 572
packet
inbound processing, 330
physical address, 46
view, 46
Physical Point of Attachment (see PPA)
physical processor, 46
physio routine, 34
using, 108
PIO, 399
postinstall script
guidelines, 546–547
qlisp sample, 571
sample, 559
template, 559
writing, 559
postload operation, 142
postremove script
guidelines, 550–551
sample, 563
template, 563
writing, 563
postunload operation, 143
PPA, 301
preinstall script
guidelines, 545–546
qlisp sample, 570
sample, 558
template, 558
writing, 558
preload operation, 142
preremove script
guidelines, 550
sample, 562
template, 562
writing, 562
preunload operation, 142, 143
primitive
processing by HP-DLPI driver, 251
required for HP-DLPI driver, 252
private flag, 184
privilege
determining, 36
probe
SCSI controller sequence, 421
probe parameter
address, 95
driver, 94
probe routine
and SCSI targets, 403
writing, 93
probe type
PROBE_ADDRESS, 93
PROBE_FIRST, 93
PROBE_NEXT, 93
Processor Dependent Code (see PDC)
processor mastered read transaction, 198
processor mastered write, 198
product, 512
attributes, 517, 526–529
listing for qlisp, 572
naming, 514
relocating, 539
product object
sample, 555
template, 554
writing, 554
Product Specification File (see PSF)
Programmed I/O (see PIO)
promiscuous level
bound, 305
inbound, 305
inbound processing, 326
outbound, 305
outbound processing, 324
PROMISC_MULTI, 305
PROMISC_PHY, 305
PROMISC_SAP, 305
unbound, 305
promiscuous mode, 272
promiscuous stream
bound, 324, 326
unbound, 324, 326
prop_create routine, 55
prop_destroy routine, 56
prop_destroy_all routine, 56
prop_get routine, 56
prop_modify routine, 56
prop_size routine, 56
property
creating for an interface, 55
destroying all for an interface, 56
destroying for an interface, 56
modifying for an interface, 56
retrieving for an interface, 56
retrieving size, 56
protocol, 303
binding, 304
demultiplexing, 304
determining packet format, 304
unbinding, 304
protocol format
Ethernet, 304
IEEE 802.2, 304
LLC, 304
SNAP, 304
protocol interface layer, 215
protocol suite
OSI, 215
SNA, 215
X.25, 215
pSCSI, 399
specifying parameters, 420
pSCSI driver
controller registration example, 422
pseudo driver, 33
sample header file, 84
PSF, 515
and swpackage command, 514
attribute types, 517
bundle attributes, 522–526
category attributes, 517–522
fileset attributes, 531–535
product attributes, 526–529
qlisp sample, 569
subproduct attributes, 530
vendor attributes, 517
writing, 552–557
writing bundle object, 553
writing category object, 553
writing fileset object, 556
writing product object, 554
writing subproduct object, 555
writing vendor object, 552
putnext routine, 303
putq routine, 303

Q
qreply routine, 303

R
race condition
  avoiding in tunable handler, 189
raw device, 34
raw I/O
  buffered, 34
  unbuffered, 34
read system call, 34, 86
reader-writer lock, 72
deadlock, 73
  routines, 72
reader-writer spinlock, 72
deadlock, 73
  routines, 72
Real Time Extension (see RTE)
register
  base address, 201
discovering, 50
I/O base, 202
mapping, 50
  mapping memory base, 202
memory base, 201
register space, 197
registration
  SCSI controller, 421
resource allocation failure, 462
resource analysis, 478
resume request, 465
rmsf command
description, 446
routine
  identifying for drivers, 85
  secondary, 71
rsvd2p, 300
rsdvlp, 300
RTE, 507
run dispatch point, 162
rwlock_alloc routine, 72
rwlock_attr_init routine, 72
rwlock_attr_setflag routine, 72
rwlock_dealloc routine, 72
rwlock downgrade routine, 72
rwlock owned routine, 72
rwlock rdlock routine, 72
rwlock有的LOCK routine, 72
rwlock tryrdlock routine, 72
rwlock tryupgrade routine, 72
rwlock trywrlock routine, 72
rwlock unlock routine, 72
rwlock upgrade routine, 72
rwlock wrlock routine, 72
rwlock wrowned routine, 72
rwspin Alloc routine, 72
rwspin_attr_init routine, 72
rwspin_attr_setflag routine, 72
rwspin dealloc routine, 73
rwspin_rdlock routine, 73
rwspin_rduunlock routine, 73
rwspin_wlock routine, 73
rwspin wrunlock routine, 73

S
SA
  in outbound packet processing, 269
SAM, 327
deprecation notice, 387
SAP
data fields, 321
SAS, 399
script, 536
  (see also control script)
and NIC initialization, 287
and nwmgr command, 338
checkinstall, 543–545
data, 547–548
driver startup, 287
modprep, 142
OL*, 469
postinstall, 546–547
postremove, 550–551
preinstall, 545–546
preremove, 550
subformatter configuration example, 373
unconfigure, 549–550
SCSI
addressing, 402
SCSI class drivers, 398
SCSI interface driver, 399
driver startup, 439
and asynchronous events, 442
and DLKM, 415
conditional entry points, 406
ccontoller registration example, 422
description, 403
detecting a target port ID change, 444
DMA mapping services, 408
external data structures, 411–412
handling high-priority I/O, 436
installing, 414
ioctl interfaces, 438
load routine example, 415
migration guidelines, 404
registering I/O object, 443
registering the controller, 422
required entry points, 405
required header file, 412
sending asynchronous events, 407
unload routine example, 415
writing, 405–444

SCSI probe
registration example, 100

SCSI Services
available functions, 406–411

SCSI services
task management functions, 436

SCSI transport drivers (see SCSI interface driver)
scsi_probe routine, 100

SD
and logging, 372
and tracing, 372
controller, 514
creating, 511–573
designing the structure, 552
overview, 511
package structure, 515
packager, 514
packaging components, 564–566
selecting the product directory structure, 552
structure, 512

SD-UX
HP-UX Software Distributor, 573
select code
interface, 38
select system call, 86

semaphore (see counting semaphore) (see mutex) (see reader-writer lock) (see reader-writer spinlock)

semicoherent platforms, 45
Serial Attached SCSI (see SAS)
service
WSIO I/O Port Space, 44
WSIO interrupt, 41
WSIO memory allocation, 43
WSIO register, 43, 50
sf_info_t structure, 447

shared library
and NIC tool, 388
and nwmgr, 341–351
and SMH, 388
building for NIC tool, 395
building for nwmgr, 351
compiling for I/O commands, 449
compiling for nwmgr, 351
CRA, 481
creating for I/O commands, 447
creating for NIC tool, 393
I/O command example, 450–454
I/O command interfaces, 449
installing for nwmgr, 351

SMH, 395
(see also NIC tool)
and native DLPI LAN driver, 327
and shared library, 388
required ioctls, 255
SNA protocol suite, 215
Software Depot (see SD)
Software Package Builder
SPB, 573
Source Address (see SA)

SPB
Software Package Builder, 573
special file (see device special file)
spin_alloc routine, 71
spin_attr_init routine, 71
spin_dealloc routine, 71
spin_lock routine, 71
spin_locks_held routine, 71
spin_owned routine, 71
spin_trylock routine, 71
spin_unlock routine, 71
spinlock, 70
c characteristics, 70
deadlock, 73
routines, 71
startup script
sample for IELAN driver, 288
states directive, 139
states statement, 158
static keyword, 146
statistic
initializing in ENET driver, 330
status parameter
ABORT, 166
BOOT, 166
FAIL, 165
LOAD, 165
UNLOAD, 165

STREAMS driver
dynamic load data structure, 145
dynamic load function, 146
dynamic loading initialization, 146
dynamic unload function, 146
sample load function, 156
sample unload function, 156

STREAMS function
and native DLPI LAN driver, 303

STREAMS module, 377
dynamic load data structure, 147
dynamic load function, 148
dynamic load initialization, 148
dynamic unload function, 148
null
system call
  I/O, 36
  interface with driver, 86
  read, 34
  write, 34

T
  tape depot, 511
  tape driver
    required header files, 78
  target
    probe failure example, 429
    registration data structures, 426
    unregistering, 429
  target path
    I/O node callback, 430
  target path element
    offline events, 442
    online events, 442
  target port
    detecting an ID change, 444
  target port identifier, 403
  target port name, 403
  TBI, 495, 498
    and driver_isr routine, 124
    event flags, 499
    migrating interrupts, 508
  TBI driver, 495
  TCP
    formatting information, 366
  template
    data, 380
    Link Layer Header, 378, 379
    packet header, 380
  tightly coupled driver (see HP-DLPI LAN driver)
    timeout, 40
      releasing, 326
    timeout routine, 40
  TLB, 202
  trace data
    formatting, 355
  tracing (see network tracing)
  transaction
    blocking, 198
    card mastered reads to host memory, 199
    card mastered writes to host memory, 199
    interleaved, 199
    nonblocking, 198
    ordering, 198
    processor mastered read, 198
    processor mastered write, 198
  Transaction Based Interrupt (see TBI)
  Translation Lookaside Buffer (see TLB)
  transmit descriptor
    setting up, 271
  transparent ioctl, 255
  transport
    protection, 301
    synchronization, 301
  transport layer
    and DLS provider, 377
  tree reference
    KRS, 176
  tunable handler
    capabilities, 186
    example, 190
    prototype, 185
    reason codes, 185
    registering, 193
    writing, 185
  tunable parameter, 183
    accepting percentage values, 184
    adding support in kernel modules, 164
    adding to kernel modules, 183–196
    adding to the modmeta file, 185
    automatic, 189
    avoiding race conditions, 189
    changing attributes, 195
    changing default values, 195
    constraint handlers, 188
    creating documentation, 195
    default value, 184
    default values, 188
    description, 183
    dynamic changes to values, 188
    failsafe value, 184
    handler capabilities, 186
    initialization functions, 193
    making it dynamic, 195
    making it self-tuning, 195
    maximum value, 184
    minimum value, 184
    name, 183
    notification of changes, 190
    percentage, 190
    self-tuning, 189
    sign, 184
    validating values, 187
    visibility, 184
    writing handler functions, 185
    writing teardown code, 194
    zero value, 184
  tunable statement, 161
  tunable_init dispatch point, 162
  tunables
    how to change attributes, 195
    how to change default values, 195
  type directive, 139
  type statement, 158, 163

U
  UDP
    formatting information, 366
  uio structure, 39
  uiomove routine, 34
  buffering data, 109
  using, 109
  unbuffered raw I/O, 34
unconfigure script
guidelines, 549–550
sample, 561
template, 561
writing, 561
uniprocessing
compared to multiprocessing, 69
unload function, 141
STREAMS driver sample, 156
WSIO driver sample, 155
unload routine
UNLOAD interface driver example, 415
UNLOAD status parameter, 165
unloadable directive, 139, 141
unloadable statement, 160
untimeout routine, 41
user action
and NIC tool, 388
user process
mapping device I/O memory, 64
user_iomap routine, 64
errors, 65
user_iomap_private routine, 64
errors, 65
user_iounmap routine, 64
USTUB macro, 152
V
value type
KRS, 174
typep, 300
vendor
attributes, 517
vendor object
sample, 552
template, 552
writing, 552
version information
embedding in kernel modules, 164
virtual address, 46
in PCI macros, 198
view, 46
W
wakewup kernel call, 307
what string, 164
Workstation and Server I/O (see WSIO)
write coalescing
guidelines, 66
write system call, 34, 86
WSIO, 30
and driver operation, 31
event handling structures, 456
interaction with HP-DLPI driver, 228
service, 49
WSIO configuration space service
routines, 44
WSIO description service
overview, 52
routine, 52
WSIO DMA service
overview, 47, 50
routines, 48
WSIO driver
dynamic load function, 143
dynamic loading data structure, 141
dynamic loading initialization, 143
dynamic unload function, 144
dynamic unloading function, 143
sample load function, 155
sample unload function, 155
WSIO endian service
routines, 50
WSIO event handling service
overview, 51
routines, 51
WSIO I/O port space service
overview, 50
routines, 44
WSIO I/O synchronization service
routine, 52
WSIO interface registration service
overview, 53
routines, 53
WSIO interrupt service
CLM routines, 496
overview, 51
routines, 41
WSIO memory allocation service
routines, 43, 51
WSIO ordered interrupts service
overview, 52
routine, 52
WSIO pseudo driver
driver_install routine example, 89
WSIO register service
overview, 50
routines, 43
WSIO system attribute service
routines, 52
wsio_activate_probe function, 143
wsio_alloc_mem routine, 43, 51
wsio_alloc_mem_handle routine, 43, 51
wsio_allocate_dma_handle routine, 48
wsio_allocate_shared_mem routine, 48
wsio_async_scan routine, 55
WSIO_BIG_ENDIAN routine, 50
wsio_cfg_inXX routine, 44
wsio_dma_pass_thru routine, 48
wsio_dma_set_device_attributes routine, 48
wsio_completion_callback routine, 497, 499, 503
wsio_completion_cb routine, 501
wsio_destroy_legacy routine, 54
wsio_dma_pass_thru routine, 48
wsio_dma_set_device_attributes routine, 48
wsio_data structure
and HP-DLPI LAN drivers, 229
wsio_drv_data_t structure, 81
wsio_drv_info_t structure, 82
and HP-DLPI LAN drivers, 229
WSIO_EVENT_LBI_INTR_MIGR event, 497
WSIO_EVENT_MSI_INTR_MIGR event, 502, 506
WSIO_EVENT_OFFLINE_CPU event, 499, 502, 503
WSIO_EVENT_ONLINE_CPU event, 501, 502, 507
WSIO_EVENT_REMOVE event, 444
WSIO_EVENT_RESUME event, 130, 444
WSIO_EVENT_SUSPEND event, 130, 444
wsio_fastmap_dma_buffer routine, 48
wsio_flush_shared_mem routine, 48
wsio_free_dma_buffer routine, 48
wsio_free_mem routine, 43, 51
wsio_free_mem_handle routine, 43, 51
wsio_free_shared_mem routine, 48
wsio_generic_event_t structure, 130, 456, 497, 499, 501
wsio_get_active_processor_count routine, 52
wsio_get_all_registers routine, 43
wsio_get_drv_priv routine, 54
wsio_get_iports routine, 44
wsio_get_legacy routine, 54
wsio_get_parm routine, 132
wsio_get_processor_count routine, 52
wsio_get_system_params routine, 52
WSIO_HW_RESUME_TIMEOUT, 133
WSIO_HW_SUSPEND_TIMEOUT, 133
wsio_init_map_context routine, 48
wsio_install_driver function, 143
wsio_install_drv_event_handler routine, 51, 130, 502
wsio_intr_activate routine, 41, 501, 508
wsio_intr_alloc routine, 41, 508
wsio_intr_assign_cpus routine, 496
wsio_intr_deactivate routine, 41, 501, 508
wsio_intr_deactivate_nowait routine, 41
wsio_intr_enable routine, 501
wsio_intr_free routine, 41, 508
wsio_intr_get_assigned_cpu routine, 41, 501, 509
wsio_intr_get_irq_line routine, 41, 509
wsio_intr_get_loc routine, 496
wsio_intr_get_txn_info routine, 41, 501, 509
wsio_intr_migr_info_t structure, 497
wsio_intr_req_loc routine, 496
wsio_intr_set_cpu_spec routine, 41, 501, 509
wsio_intr_set_irq_line routine, 41
wsio_io_sync routine, 52
wsio_iova_to_phys routine, 48
wsioLegacy_info_t structure, 40
WSIO_LITTLE_ENDIAN routine, 50
wsio_map_cfg_handle routine, 44
wsio_map_dma_buffer routine, 48
wsio_map_port routine, 44
wsio_map_reg routine, 43, 64
wsio_msi_alloc routine, 41
wsio_msi_assign routine, 41
wsio_msi_capability routine, 41
wsio_msi_disable routine, 41
wsio_msi_enable routine, 41
wsio_msi_free routine, 41
wsio_msi_get_cpus routine, 41
wsio_msi_query routine, 41
wsio_msi_resize routine, 41
wsio_node_get_isc routine, 55
WSIO_ORDERED_INTERRUPTS routine, 52
wsio_port_inXX routine, 44
wsio_port_outXX routine, 44
wsio_put_drv_priv routine, 54
wsio_read_regXX routine, 43
wsio_reg_drv_capabilities_mask routine, 132
wsio_reg_drv_capability_mask routine, 51, 497, 499, 501, 502
wsio_reg_interface routine, 54
wsio_reg_legacy routine, 54
wsio_reg_node_t structure, 40
wsio_register_addr_probe routine, 95
wsio_register_dev_probe function, 143
wsio_register_dev_probe routine, 93
wsio_register_probe_func function, 143
wsio_remap_dma_func routine, 48
wsio_set_description routine, 52
wsio_set_dma_attributes routine, 48
wsio_set_dma_callback routine, 48
wsio_set_parm routine, 132
wsio_unmap_cfg_handle routine, 44
wsio_unmap_dma_buffer routine, 48
wsio_unmap_port routine, 44
wsio_unmap_reg routine, 43
wsio_write_regXX routine, 43
WSTUB macro, 152
X
X.25 protocol suite, 215
XPORT option, 377