Legal Notices

This document contains information which is protected by copyright. All rights are reserved. Reproduction, adaption, or translation without prior written permission is prohibited, except as allowed under the copyright laws.

Restricted Rights Legend. Use, duplication or disclosure by the U.S. Government Department of Defense is subject to restrictions as set forth in paragraph (b)(3)(ii) of the Rights in Technical Data and Software clause in FAR 52.227-703.

Copyright ©, 1988, The Santa Cruz Operation


Copyright ©, 1980, 1984 AT&T, Inc

Copyright ©, 1986, 1992 Sun Microsystems, Inc


This software and documentation is based in part on the Fourth Berkeley Software Distribution under license from the Regents of the University of California

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

Hewlett-Packard makes no warranty of any kind with regard to this manual, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.

Hewlett-Packard shall not be liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.
Printing History

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

**Edition/Part Number/Date**

First/B2355-90669/September 1999
Second/B2355-90727/August 2000
## Contents

### 1. Introduction
- The Intended Audience .............................................. 18
- Support/Compatibility Disclaimers .................................. 19
- Using This Manual ..................................................... 20
- Internet and E-mail Resources ....................................... 21
- Reference Documentation .............................................. 22

### 2. Overview of the Driver Environment
- How the I/O Subsystem is Structured ................................ 27
- General I/O System (GIO) ............................................. 27
- Context-Dependent I/O Modules (CDIOs) ............................. 28

### 3. Understanding HP-UX I/O Subsystem Features
- General Driver Topics ............................................... 35
- Overview of Driver Types .......................................... 36
- Device, Pseudo, and Interface Drivers .............................. 36
- Block and Character Devices ...................................... 36
- Major and Minor Numbers ........................................... 39
- Assigning Major Numbers ............................................ 39
- Using Minor Numbers .................................................. 40
- Device-Special Files ............................................... 40
- Device Switch Tables .................................................. 41
- System Calls .......................................................... 42
- The Two Levels of a Driver ......................................... 42
- Flow of an I/O Request .............................................. 43
- A Sample I/O Read Request .......................................... 44
- Kernel Data Structures Used for I/O ............................... 46
- The buf Structure ................................................... 46
- The isc_table_type Structure .................................... 46
- The uio Structure ................................................... 48
- The iovec Structure ................................................ 48
- Timeout Mechanisms .................................................. 49
- How Timeouts Work .................................................. 49
- The timeout() Routine .............................................. 50
- The untimeout() Routine ........................................... 50
- Interrupt Handling ................................................... 51
- The Software Trigger Mechanism .................................. 51
- Memory Allocation and Mapping ................................... 54
Contents

Cache Coherence ...................................................... 55
Driver Requirements for Coherency .................................. 56
Rules For Using dma_sync_IO() ........................................ 57
DMA Mapping ............................................................ 58
I/O Adapters ............................................................. 59
WSIO Mapping Services ................................................ 60
Programming Considerations ......................................... 61

4. Multiprocessing
Comparing Uniprocessing to Multiprocessing ......................... 65
Synchronization Mechanisms ............................................ 66
Timing Hazards and Idle Time .......................................... 66
Spinlocks ................................................................. 67
Spinlock Routines ....................................................... 67
Beta Semaphores ........................................................ 68
Beta Semaphore Routines .............................................. 68
Deadlocks ................................................................. 69
Rules for Lock Acquisition ............................................ 69
Lock Orders .............................................................. 69
Synchronization Using sleep() and wakeup() ......................... 70

5. Writing a Driver
Suggested Driver Writing Methodology ............................... 75
Writing a Driver: Step-by-Step ....................................... 75
Step 1: Choosing a Driver Name ..................................... 76
Step 2: Choosing System Header Files ................................. 77
System-Defined Header Files ......................................... 77
Step 3: Defining Installation Structures ............................... 80
The drv_ops_t Structure Type ....................................... 80
The drv_info_t Structure Type ...................................... 83
The wsio_drv_data_t Structure Type ................................. 85
The wsio_drv_info_t Structure Type ................................ 86
Sample Header for a Device Driver ................................ 86
Sample Header for an Interface Driver .................................. 88
Sample Header for a Pseudo Driver .................................. 89
Step 4: Identifying Routines for Your Driver ....................... 91
## Contents

- Data Link Interface Layer ......................................................... 182
- Network Protocol Layer .......................................................... 182
- Protocol Interface Layer, Device File, and STREAMS Head .............. 182
- Streams Environment Layer ....................................................... 183
- Streams Model Network Driver .................................................. 183
- Data Structures and Interfaces ..................................................... 184
  - hw_ift_t Structure Description and Initialization ......................... 184
  - hw_dlpi Structure Description and Initialization ......................... 188
- Protection and Synchronization for Driver and Transport ................... 189
  - hw_ift Structure Protection .................................................. 189
  - Driver Structure Protection ............................................... 190
- Initializing Networking Device Drivers ........................................ 191
  - STREAMS DLPI Driver Initialization ....................................... 191
  - WSIO Network Driver Initialization ....................................... 192
  - Calling driver_install() .................................................... 194
  - Calling driver_attach() ..................................................... 196
- Networking Driver-Specific Initialization ................................... 199
- Protocol Binding and Demultiplexing ......................................... 201
  - Protocol Binding and Unbinding ............................................ 202
  - Protocol Demultiplexing .................................................... 202
  - Distinguishing Packet Protocol Format .................................... 202
- Message block and queue_t functions and macros ................................ 204
- DLPI Sequence ........................................................................ 205
  - Device file, Interface name and PPA number ................................ 205
  - IP and ARP Configuration .................................................... 206
  - STEAMS DLPI and network driver overview .................................. 207
- STREAMS DLPI Driver ............................................................... 211
  - Major Data Structures ......................................................... 211
  - Opening and Closing a Driver ................................................. 217
  - Control Functions .................................................................. 218
  - IOCTL Processing .................................................................. 221
  - Transmission of Message blocks ............................................. 222
  - Reception of Message blocks .................................................. 223
  - Summary of DLPI Primitives and IOCTLs ................................... 223
- WSIO network driver ................................................................. 226
  - Data Structures ................................................................. 226
## Contents

- logged_info, logged_link .................................................. 231
- IOCTLs ............................................................................. 232
- Outbound Path .................................................................... 234
- Inbound Path ...................................................................... 235
- DLKM drivers ...................................................................... 237
- Platform Specifics ............................................................... 238
- V Class ................................................................................. 238
- Network Management Support ............................................. 240
- Configuring a Networking Driver Through SAM .................. 245
- Network Monitoring Commands:
  - driveradmin, driverlinkloop, and lanscan ......................... 246
  - driveradmin command ...................................................... 246
  - lanscan Command ......................................................... 249
- Network Tracing and Logging Support for Troubleshooting .... 253
  - Introductory Overview of HP-UX Tracing and Logging ........ 253
  - Using HP-UX Logging and Tracing for Troubleshooting Support 257
  - Passing Data to HP-UX Tracing/Logging ......................... 262
  - Formatting Networking Trace/Log Messages ..................... 271
- Alternative Means of Development Debugging ..................... 274

### 8. Writing PCI Device Drivers

- Introduction ....................................................................... 278
- Terms and Definitions ......................................................... 279
- PCI Overview ...................................................................... 283
  - PCI Register Spaces ......................................................... 283
  - PCI Transaction Ordering ................................................ 284
  - PCI Endian Issues .......................................................... 288
- PCI Device Setup ............................................................... 290
  - Mapping Base Address Registers into PCI Memory and I/O Space 290
  - Using the Base Address Registers ..................................... 290
  - Automatic IRQ Determination ......................................... 292
  - Mapping the Memory Base Register .................................. 292
  - PCI Configuration Space Restrictions ............................... 293
- PCI Device Operation .......................................................... 295
  - The PCI_LITTLE_ENDIAN_ONLY Flag ................................. 295
  - Direct Memory Access (DMA) ........................................... 296
- PCI Masters and Coherence .................................................. 298
- Leveraging Existing Drivers .................................................. 299

---

9
# Contents

Multibus Drivers ........................................ 299  
PCI Services Summary .................................. 301  
Unsupported WSIO Functions ............................ 302  
Multiprocessor (MP) Safety ............................ 303  
Constants and Data Structures ......................... 304  
  User Visible PCI-Specific Data Structures .......... 304  
  Defined Constants .................................. 304  
A Sample PCI Driver .................................... 306  
Sample WSIO Setup and Structures .................... 306  
Sample WSIO Routines .................................. 308  
Other Driver Entry Point Routines ..................... 313

## 9. Writing SCSI Device Drivers

SCSI Driver Development, Step 1: Include Header Files .......................... 317  
  General Header Files ................................ 317  
  Header Files for SCSI Drivers ....................... 317  
  Header Files for Device Classes .................... 317  
SCSI Driver Development, Step 2: Set Up Structures ......................... 319  
SCSI Driver Development, Step 3:  
  Create the driver_install Routine .................. 322  
SCSI Driver Development, Step 4:  
  Create the driver_dev_init() Routine ............... 323  
    Setting up the Device-Switch Table (scsi_ddsw) .... 323  
SCSI Driver Development, Step 5:  
  Analyze Multiprocessor Implications ............... 327  
SCSI Driver Development, Step 6:  
  Create the Entry-Point Routines ................... 328  
    driver_open() Routine ................................ 328  
    driver_close() Routine ............................. 328  
    driver_read() Routine ............................. 328  
    driver_write() Routine ............................ 329  
    driver_strategy() Routine ......................... 329  
    driver_psize() Routine ............................ 329  
    driver_ioctl() Routine ............................. 329  
SCSI Driver Development, Step 7: Error Handling ......................... 331  
SCSI Driver Development, Step 8: Underlying Routines .................... 332  
  dd_close Routine .................................. 333
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Installing the Module’s Component Files</td>
<td>394</td>
</tr>
<tr>
<td>Preparing the Module for Configuration</td>
<td>395</td>
</tr>
<tr>
<td>Tuning the Module</td>
<td>396</td>
</tr>
<tr>
<td>Configuring the Module into the System</td>
<td>397</td>
</tr>
<tr>
<td>Updating the Module’s Image</td>
<td>399</td>
</tr>
<tr>
<td>Updating the Kernel’s Image</td>
<td>400</td>
</tr>
<tr>
<td>Loading the Module</td>
<td>401</td>
</tr>
<tr>
<td>Loading the Kernel</td>
<td>402</td>
</tr>
<tr>
<td>Querying the Module</td>
<td>402</td>
</tr>
<tr>
<td>Creating Device Special Files for the Module</td>
<td>403</td>
</tr>
<tr>
<td>Testing the Module</td>
<td>405</td>
</tr>
<tr>
<td>Unloading the Module</td>
<td>406</td>
</tr>
<tr>
<td>Correcting Errors in the Module’s Component Files</td>
<td>406</td>
</tr>
<tr>
<td>Updating the Module’s Components</td>
<td>407</td>
</tr>
<tr>
<td>Removing the Module’s Components from the System</td>
<td>407</td>
</tr>
<tr>
<td>Sample DLKM WSIO Class Driver</td>
<td>409</td>
</tr>
<tr>
<td>Sample Component Files</td>
<td>411</td>
</tr>
<tr>
<td>Sample Makefile</td>
<td>418</td>
</tr>
</tbody>
</table>

### 11. How To Make Pre 11.0 Drivers 64-Bit Safe

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>425</td>
</tr>
<tr>
<td>Important Terms and Concepts</td>
<td>425</td>
</tr>
<tr>
<td>The 64-bit Data Model</td>
<td>426</td>
</tr>
<tr>
<td>LP64 Considerations</td>
<td>427</td>
</tr>
<tr>
<td>General Guidelines</td>
<td>429</td>
</tr>
<tr>
<td>64-bit Clean Headers</td>
<td>429</td>
</tr>
<tr>
<td>Useful Data Types</td>
<td>430</td>
</tr>
<tr>
<td>Hardware Considerations</td>
<td>432</td>
</tr>
<tr>
<td>Software Considerations</td>
<td>433</td>
</tr>
<tr>
<td>ANSI Function Prototypes</td>
<td>434</td>
</tr>
</tbody>
</table>

### A. Example Drivers

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ram Disk Static Module Device Driver</td>
<td>441</td>
</tr>
<tr>
<td>Ram Disk Static Device Driver: Program Source</td>
<td>441</td>
</tr>
<tr>
<td>Ram Disk Static Device Driver: 64-bit makefile</td>
<td>446</td>
</tr>
<tr>
<td>Ram Disk Static Device Driver: 32-bit makefile</td>
<td>449</td>
</tr>
</tbody>
</table>
Contents

Ram Disk Static Device Driver: master file and
general information ........................................... 454
Ram Disk WSIO DLKM Driver ..................................... 455
Ram Disk DLKM Device Driver: Program Source .............. 455
Ram Disk DLKM Device Driver: Makefile ....................... 462
Ram Disk DLKM Device Driver: mod32.mk ..................... 463
Ram Disk DLKM Device Driver: mod64.mk ..................... 464
Ram Disk DLKM Device Driver: master and system files ....... 465
Ram Disk DLKM Device Driver: miscellaneous information .... 466
PCI Monolithic Hardcopy Interface Driver ....................... 468
IKON PCI Monolithic Driver: Header File ..................... 468
IKON PCI Monolithic Driver: Program Source ................. 470
IKON PCI Monolithic Driver: Master File ...................... 476
IKON PCI Monolithic Driver: 32-bit Makefile .................. 476
IKON PCI Monolithic Driver: 64-bit Makefile ................. 479
PCI Combined User and Kernel Driver ......................... 482
PCI Combined Driver: Header File ............................. 482
PCI Combined Driver: Kernel Program Source ................ 482
PCI Combined Driver: User Program Source ................... 488
PCI Combined Driver: 32-bit Makefile ......................... 490
PCI Combined Driver: 64-bit errata
and both 32-bit and 64-bit usage warning ..................... 495

B. Data Structures, Defines,
Routines, Flags, and Code Examples

Interface Group MIB Objects .................................. 499
Standard MIB II ..................................................... 499
Extended MIB II for 802.3 and 802.5 ......................... 499
MAC Types and Protocol Types ................................ 502
Message Types for DLPI Primitive and Acknowledgment .... 503
MIB Event and Event Record ................................... 504
MTU Values ......................................................... 505
Packet Headers ..................................................... 506
Packet Types ....................................................... 509
Protocol Kinds and Values ..................................... 510
SAP Values for IEEE 802.2 LLC Packets ...................... 511
TYPE Values for Ethernet and SNAP Protocols ............... 512
## Contents

**C. How to Design a Networking Trace/Log Subformatter and a Sample Subformatter**
- Designing a Subformatter ................................................................. 515
- Alternative Subformatter Implementations ........................................ 516
- Network Trace/Log Subsystem Installation Testing .......................... 549

**D. Shared Library Examples for the driveradmin and lanscan Commands**
- The lanscan's shared library sample code ........................................ 553
- The driveradmin's shared library sample code ................................... 556
- The driveradmin sample code .......................................................... 578
- The enetlinkloop's sample code ....................................................... 678

**Glossary**
1 Introduction
This manual describes how to write I/O driver routines for the WSIO interface on HP 9000 workstations (Series 700 and others) and servers (Series 800 and others). Except where noted, this manual is applicable to all workstations and servers using Release 11.0. The manual is organized as follows:

**Chapter 1: Introduction**
- How to use this manual and other references.

**Chapter 2: Overview of the Driver Environment**
- The I/O subsystem's structure and how drivers fit into this environment.

**Chapter 3: Understanding HP-UX I/O Subsystem Features**
- Features of the I/O subsystem, such as types of drivers, memory mapping, flow of I/O requests, data structures, and interrupt handling.

**Chapter 4: Multiprocessing**
- Multiprocessor issues.

**Chapter 5: Writing a Driver**
- A step-by-step strategy for writing drivers. It includes descriptions of routines used by device drivers, interface drivers, and combined drivers.

**Chapter 6: Installing Your Driver**
- Installing your driver in the kernel and configuring it to communicate with the hardware.

**Chapter 7: Creating Networking Device Drivers**
- Designing and writing networking device drivers.

**Chapter 8: Writing PCI Device Drivers**
- PCI bus driver routines.

**Chapter 9: Writing SCSI Device Drivers**
- SCSI bus driver routines.

**Chapter 10: Developing Dynamically Loadable Kernel Modules**
• Adding a kernel module to a running UNIX system without rebooting the system or rebuilding the kernel.

**Chapter 11: How To Make Pre 11.0 Drivers 64-Bit Safe**

• How to modify a Release 10.20 32-bit driver to run in a Release 11.0 32-bit or 64-bit environment.

**Appendix A: Example Drivers**

• Driver examples.

**Appendix B: Data Structures, Defines, Routines, Flags, and Code Examples**

• Networking related data structures, defines, routines, flags, and code examples.

**Appendix C: How to Design a Networking Trace/Log Subformatter and a Sample Subformatter**

• Generic subformatter code for networking drivers.

**Appendix D: Shared Library Examples for the lanadmin and lanscan Commands**

• Shared libraries and message catalogs that provide code examples to be used with network administration.
The Intended Audience

Porting an existing device driver is not a trivial task. Writing a device driver is even more complex. Using this manual to port or write a driver assumes that you know how to:

- Write programs in the C language.
- Understand the basic concepts of writing a driver.
- Understand the functionality of the hardware for which you are writing the driver.
- Read the HP-UX System Administration Tasks manual and perform system administration.
- Understand the virtual memory, I/O, and file system areas in the HP-UX and/or UNIX operating systems.

These assumptions are not meant to discourage anyone, but you should not plunge onward unless you know the HP-UX (UNIX) operating system, the C language, and the implications of writing drivers. The “Support/Compatibility Disclaimers” section describes the support provided by Hewlett-Packard Company.
Support/Compatibility Disclaimers

Since drivers function at the level of the kernel, Hewlett-Packard Company (HP) reminds you of the following:

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

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

• Should difficulties arise during the development and test phases of writing a driver, HP may provide assistance in isolating problems to determine if:
  — HP hardware is not at fault; and
  — HP software (firmware) is not at fault by removing user-written kernel drivers.

• When HP hardware, software, and firmware are not at fault, you should seek help from the third party from whom you obtained software or hardware.
Using This Manual

How you read this manual depends on the tasks you need to perform. The steps you need to take will differ depending on whether you are writing a new kernel driver or porting an existing driver.

Chapter 5, “Writing a Driver,” describes general routines you will need to use regardless of the type of driver you are writing. These routines include the normal kernel driver entry points such as driver_open, driver_close, driver_read, and others. Tasks such as mapping of card registers, usage of DMA service routines, and linkage of interrupt service routines are usually bus-specific. For example, Chapter 8, “Writing PCI Device Drivers” contains the bus specific routines for PCI.

Chapter 11, “Developing Dynamically Loadable Kernel Modules”, describes the additional routines and tools you will need to use for a DLKM driver.

NOTE
This book contains many examples of C programs to help you 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.
Internet and E-mail Resources

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

- **Interface Program at** //www.hp.com/visualize/programs/netio/index.html
- **Interface Program E-mail at** interface@fc.hp.com
- **Developer Resource at** http://devresource.hp.com/
Reference Documentation

- Hewlett-Packard Company
  - HP-UX 10.20 Driver Development Reference, HP Part No. B2355-90067
  - HP-UX 11.0 Driver Development Reference, HP Part No. TBA
  - Dealer Configuration File Creation Guide, HP Part No. D2230-90001
  - EISA Configuration Guide for HP-UX, HP Part No. B2370-90000
  - HP-UX Managing Systems and Workgroups, HP Part No. B2355-90664
  - HP-UX Reference, HP Part No. B2355-90052
  - HP-UX System Administration Tasks, HP Part No. B2355-90079
  - HP C Programmer’s Guide, HP Part No. 92434-90002
  - Configuring HP-UX for Peripherals, HP Part No. B2355-90053
  - Installing and Updating HP-UX, HP Part No. B2355-90078
  - PA-RISC 1.1 Architecture and Instruction Set Reference Manual, HP Part No. 09740-90039
  - PA-RISC Procedure Calling Conventions Reference Manual, HP Part No. 09740-90015
  - Managing HP-UX Software with SD-UX, HP Part No. B2355-90044
J 2237-90005

• **Other References**
  
  — Edward Solari, EISA Bus Design, Annabooks
  
  — EISA Specification Version 3.10 or later, BCPR Services, Inc.
  
  — PCI Local Bus Specification, Revision 2.1, PCI Special Interest Group
  
  
  — PCI System Design Guide, Revision 1.0, PCI Special Interest Group
  
  — Data Link Provider Interface Specifications, Unix International
2 Overview of the Driver Environment
Overview of the Driver Environment

This book is intended for individuals writing interface and device drivers for HP-UX Workstations. Much of the information applies to HP-UX Servers, however the main focus is on Workstation I/O.

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

The I/O subsystem provides a uniform interface for user processes to use in reading information from and writing 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) also play their parts.

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

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:

- Management of data structures used for I/O configuration

  Data structures that can be manipulated by system-administration utilities or that are global to the system must be maintained by the GIO. These include:
  - I/O tree node
  - block and character switch tables
  - the 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.
Overview of the Driver Environment

How the I/O Subsystem is Structured

- The system administration interface
  System administration utilities must see a consistent view of the system that is independent of individual drivers' views.

**Context-Dependent I/O Modules (CDIOS)**

CDIOS contain functionality specific to a particular bus or driver environment (the context is defined by the bus or driver environment). Individual, independent CDIOS provide a way to hide any driver-environment specifics from other environments. This, in turn, allows individual component CDIOSs 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, which provide bus-dependent services to other CDIOSs. They may have bus-nexus drivers to control bus adapters or bus converters. A kernel can contain the following bus-nexus CDIOS:
  - CORE CDIO (Core Context Dependent I/O Module)
  - PA CDIO (Precision Architecture Context Dependent I/O Module)
  - EISA CDIO - optional (EISA Context Dependent I/O Module)
  - PCI CDIO - optional (PCI Context Dependent I/O Module)

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

**Basic Components of a CDIO**

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

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

How the I/O Subsystem is Structured

• Driver Services

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

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

• Drivers

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

• Management of I/O resources

A CDIO controls resources specific to a device. For example, the PA CDIO controls resources specific to Precision Architecture, such as interrupt bits and I/O PDIR entries. The WSIO CDIO manages common structures like the Interface Select Code (ISC) table.

How the Driver Environment Works

The WSIO CDIO was originally designed for the workstation single-processor environment. With HP-UX Release 10.20, its functionality was expanded to encompass the server multiprocessor environment as well. 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 EISA CDIO, or with the system’s I/O bus (see Figure
Overview of the Driver Environment

How the I/O Subsystem is Structured

2-1, “Same Driver Can Operate in Many Configurations.” To the driver, all configurations look the same. It is the task of the WSIO CDIO to interpret the service calls and to take the appropriate actions for the given configuration.

Figure 2-1  Same Driver Can Operate in Many Configurations

![Diagram of driver configurations]

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

Refer to the HP-UX Driver Development Reference for the reference pages for WSIO CDIO routines, services, and data structures.
Figure 2-2  WSIO CDIO as a Buffer Zone

PA module

Core

wsio
drv_x

wsio
drv_x

wsio
drv_x

EISA
Overview of the Driver Environment
How the I/O Subsystem is Structured
3 Understanding HP-UX I/O Subsystem Features
This chapter explains software concepts associated with the HP-UX I/O subsystem. It also provides information and models that help you understand how to write a driver.
General Driver Topics

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

WSI O supports three types of drivers: device, pseudo, and interface; and two device access types: block and character.

Device, Pseudo, and Interface Drivers

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

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

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

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

Block and Character Devices

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

- The driver of a **block device** accesses a structured device, such as a disk or magnetic tape that supports a file system. These drivers generally allow buffered, random access. Block devices are accessed through the file system buffer cache.
• The driver of a character device accesses a device as a character device (everything not accessed in blocks). They are also called raw devices.

Features of Block Devices
Block devices have file system support and are random access devices.

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

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

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

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

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

No File System Support  The kernel does not cache data for character devices. On accessing a device as a character device, data is transmitted in units of one or more bytes.

Raw I/O: Buffered or Unbuffered  There are two cases for raw I/O with character devices. First, you can use physio() for direct (unbuffered) data transfer by the character-device driver. Second, you can have a character device driver set up its own buffer using copyin()
Understanding HP-UX I/O Subsystem Features

Overview of Driver Types

or uioMove(). This method is useful for a small amount of data or if you need 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 (such as a SCSI disk driver and an optical autochanger) have both a block-major number and a character-major number. Devices that support only character-mode access have only a character-major number.

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

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

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

Assigning Major Numbers

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

- In your driver's master file in the /usr/conf/master.d directory, specify -1 in both the char major and block major fields in the $DRIVER_INSTALL table.
- Specify -1 in both the b_major and c_major fields of the drv_info_t structure in your driver's header. See “Step 3: Defining Installation Structures” on page 80, Chapter 5, for more information about the drv_info_t structure.
- Also in the drv_info_t structure, set the following bit values in the flags field. If you have a block driver, set the DRV_BLOCK value. If you have a character driver, set the DRV_CHAR value. If your driver is both a block and a character driver, set both values.
Understanding HP-UX I/O Subsystem Features

Major and Minor Numbers

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

Using Minor Numbers

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

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

For more information, you can also consult the device driver manpages in section 7 of the HP-UX Reference.

Device-Special Files

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

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

The two fields of importance are the major number (193) and the minor number (0x00080). The major and minor numbers are combined to form a numerical designation for the device driver, in what is called the `dev_t` format. This format consists of:

- **Bits 0-7**: The major number, which can range from 0 to 255. Character and block major numbers are separate ranges.

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

Device Switch Tables

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

The kernel maintains two switch tables for devices:

- bdevsw for block devices.
- cdevsw for character devices.

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

The kernel automatically uses your driver's installation routine, entries in your driver's dev_ops_t and dev_info_t structures, and information from your driver's master file in /usr/conf/master.d, to construct the switch tables. See “Step 5: Writing Configuration Routines” on page 96, Chapter 5, “Step 3: Defining Installation Structures” on page 80, Chapter 5, and “Step 2: Create a Master File” on page 160, Chapter 6.
A user process performs I/O by making system calls. The kernel executes the system calls on behalf of the user process. The processing performed by the kernel depends on:

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

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

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

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

**The Two Levels of a Driver**

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

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

Interrupts are handled by an interrupt service routine (ISR) and supporting routines in the interface driver. This chapter describes interrupts in "Interrupt Handling"; Chapter 5, "Writing a Driver," describes the interface driver routines.
Flow of an I/O Request

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

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

1. In the user process
   - Invoke the kernel.
     A user process makes an I/O system call and invokes the kernel through the system-call interface. The kernel does some processing related to the management of the process and of resources for the request.

2. In the kernel
   - Get major and minor numbers.
     If the file specified in the I/O system call is a device-special file, the kernel gets the major and minor numbers from the file's inode. The kernel uses the major number to index into a device switch table and sets up parameters, if any, to be passed to the driver.

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

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

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

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

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

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

6. In the kernel
   - Return control to user process.
   The kernel interprets the return value from the device driver, and sets the return value of the system call accordingly. It then returns control to the user process.

### A Sample I/O Read Request

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

1. A user runs a program that executes a \texttt{read()} system call on a character device file.
2. The \texttt{read()} system call:
   a. Performs necessary preprocessing of the request, such as verifying
that the file is a character device, checking permissions.

b. Extracts the major and minor numbers from the inode for the special file.

c. Uses the major number to index into the cdevsw table to obtain the name of the device driver’s driver_read() routine.

d. Invokes the driver_read() routine.

3. The driver_read() routine calls physio() to invoke the driver_strategy() routine.

4. The driver_strategy() routine gets the minor number to access the correct device and decodes the device options.

5. The driver_strategy() routine queues a request on the interface driver for a read from the device.

6. The interface driver is currently processing another request, so the driver_strategy() routine returns and physio() sleeps while waiting for the I/O request to complete.

7. The device interrupts when it completes the previous I/O request.

8. The interface driver processes the interrupt and starts the next request waiting on the queue (our request).

9. The interface driver tells the hardware to perform our read request.

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

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

12. Control returns to the driver_read() routine, which completes any final device-specific processing of the request.

13. Control returns to the read() system call, which completes the request and returns control to the user process.
Kernel Data Structures Used for I/O

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

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

**The `buf` Structure**

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

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

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

**The `isc_table_type` Structure**

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

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

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

See `isc_table_type(KER4)` for details about fields in the `isc_table_type` structure.
The I/O Switch Tables

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

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

- `isc->gfsw.init` points to a driver-defined interface initialization routine that is run by the system during the boot process. See Chapter 5, "Writing a Driver."
- `isc->gfsw.diag` points to a driver-defined interface diagnostic routine whose usage is currently undefined.

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

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

The interface drivers use identical structures to specify their operations and put the addresses of the structures in their respective `isc->ifsw` fields. The device driver obtains the ISC structure for the appropriate interface driver (see `wsi0_get_isc(WSIO3)`) and then calls the interface driver through the switch table to perform the operation.

This allows the device driver to trigger interface-specific routines without actually having to know which interface driver is configured with it. Multiple interface drivers could be configured interchangeably with this device driver, as long as they share the same `ifsw` type.
Understanding HP-UX I/O Subsystem Features

Kernel Data Structures Used for I/O

definition.

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

For an example of a switch table that you could adapt for ifsw, look at
the drv_table_type structure, defined in <sys/hpib.h>.

“Legacy” Function Switch  For backward compatibility, the ISC
structure also supports the iosw field, which points to a drv_table_type
structure.

The uio Structure

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

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

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

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

The iovec Structure

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

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

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

How Timeouts Work

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

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

The `timeout()` routine works as follows:

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

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

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

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

The `untimout() ` Routine

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

- **func**: Address of the function passed to `timeout()`.
- **arg**: Argument passed to the `timeout()`.
Interrupt Handling

Drivers can process I/O transactions using polled I/O or interrupts. The choice will depend on the device.

With polled I/O, the device does not interrupt the processor. The driver starts a transfer on the device, then polls the device to see when it is ready to send or receive the next byte. The driver establishes the polling interval by setting a timeout.

With interrupts, the driver starts a transfer on the device, then waits for it to interrupt the processor (indicating the I/O operation is complete and the device is ready to transfer more data). Interrupts can occur at any time, so be sure code segments that cannot be interrupted are protected by spinlocks (see Chapter 4, “Multiprocessing”).

A driver registers its Interrupt Service Routine (ISR) by calling `isrlink()`. See Chapter 5, "Writing a Driver", for details.

The Software Trigger Mechanism

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

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

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

Software Trigger Routine

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

```c
void sw_trigger(struct sw_intloc *intloc, int (*proc)(),
               caddr_t arg, int level, int sublevel);
```
intloc  A pointer to a sw_intloc structure to be added to the queue of software triggers. The driver allocates the structure, zero-filled. The sw_trigger() routine initializes its fields.

proc  The address of a routine to be called when the software trigger is executed.

arg  The argument to be passed to proc.

level  The priority level of the software trigger.

sublevel  Currently, sublevels are not implemented. Drivers should use 0 as the last argument.

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

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

The level value has the following restrictions:

- Your driver cannot set a software trigger higher than your current processor priority level.
- You cannot call sw_trigger() with level set to 7.

**A Skeletal Driver Fragment**

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

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

static struct sw_intloc mycard_intloc;

static void
mycard_isr(void)
{
    caddr_t reason;
```
Understanding HP-UX I/O Subsystem Features

Interrupt Handling

Chapter 3

/* stop card from interrupting */
mycard->control = ......;

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

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

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

HP-UX provides the following routines to dynamically allocate and free a block of host memory: kmalloc() and kfree(). The kmalloc() routine allocates a block of host memory that is mapped in kernel virtual address space and contiguous in virtual address space, but may be discontiguous in physical address space. For example, a block of host memory that is 8K bytes in size occupies two 4K pages, where the pages may not be physically contiguous in host memory. The largest allocation size a driver can make with kmalloc() that is guaranteed to be physically contiguous in host memory is 4K bytes.

If a driver needs to allocate a block of memory that is larger than 4K bytes and physically contiguous (contiguous from the view of the card), then the driver uses the wsio_allocate_shared_memory() routine. This routine is also used to allocate a block of memory that is used by the device as control or scratch pad memory. Such memory is typically accessed and updated by the device on a continuous basis to keep track of I/O transactions that are in progress.

Memory allocated by wsio_allocate_shared_memory() is automatically mapped for device access. Memory allowed by kmalloc(), however, must be mapped for device access by the driver using wsio_map() or wsio_fastmap().

Interface drivers may need to map device registers into kernel virtual address space for the driver to access the device. Drivers use the map_mem_to_host() routine to accomplish this mapping.

See the HP-UX Driver Development Reference for details on using these routines.
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.

Coherent platforms implement DMA such that accesses to data in host memory by I/O devices are consistent with accesses by processor caches. Hardware in the platform maintains the consistent view of data in host memory as DMA transactions flow through the hardware.

Semicoherent platforms implement DMA similar to coherent systems. 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.

Noncoherent platforms implement DMA such that 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.
Driver Requirements for Coherency

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

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

PCI buses have special coherency exceptions which are discussed in Chapter 9 “Writing PCI Device Drivers”. They are also discussed in
pci_errata(PCI5), wsio_map(WSI03), and wsio_fastmap(WSI03) in the HP-UX Driver Development Reference.

**Rules For Using dma_sync_IO()**

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

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

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

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

There are three address views to host memory: physical, 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 termed the physical address view. HP-UX supports platforms where the physical address width of the processor/memory interconnect is 32 bits wide or wider.

• Virtual Address View

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

• 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 the physical address width of the processor memory interconnect is 32 bits wide, implement only the physical address view to I/O devices. Platforms that are semi-coherent or coherent, or where the processor/memory interconnect is greater than 32 bits wide, generally implement the IOVA view.

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

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

Address translation is assisted by the I/O PDIR (Page Directory) 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
DMA Mapping

The kernel maintains a table to provide mappings between IOVAs and physical addresses.

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

**Coherent I/O Systems**

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

**Noncoherent I/O Systems**

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

**WSIO Mapping Services**

The following table lists the WSIO mapping services provided by the kernel. Refer to the HP-UX Driver Development Reference for details.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>init_map_context()</td>
<td>Initialize the mapping context used by wsio_map().</td>
</tr>
<tr>
<td>wsio_fastmap()</td>
<td>Map all or part of a host address range into an IOVA range contained in a page.</td>
</tr>
<tr>
<td>wsio_map()</td>
<td>Map all or part of a host address range into an IOVA range.</td>
</tr>
<tr>
<td>wsio_remap()</td>
<td>Map a range of host addresses into a premapped range of IOVAs.</td>
</tr>
<tr>
<td>wsio_set_attributes()</td>
<td>Set attributes of wsio_map().</td>
</tr>
<tr>
<td>wsio_unmap()</td>
<td>Unmap a range of IOVAs.</td>
</tr>
</tbody>
</table>

Table 3-1 WSIO Mapping Services
Programming Considerations

The WSIO mapping services present a programming model that assumes the existence of I/O adapters and IOVAs. These same services are also used for platforms that do not implement I/O adapters, where physical addresses are used in lieu of IOVAs.

Using \texttt{wsio\_map()}

Drivers must call \texttt{wsio\_map()} to map a virtual address range to an IOVA range. Each call to \texttt{wsio\_map()} reserves one or more I/O PDIR entries and is done by the driver to map a buffer in preparation for DMA transactions by the device. The I/O PDIR entries are released when \texttt{wsio\_unmap()} is called by the driver after the device DMA transactions have completed.

I/O PDIR entries are a limited resource, so \texttt{wsio\_map()} may return an error if resources are not available, or only partially map a virtual address range. Drivers must be prepared to handle this.

Each I/O PDIR entry provides a translation for a whole page of data. Drivers should avoid mapping many small buffers (each reserving an I/O PDIR entry), and map PAGESIZE or larger buffers when possible.

A driver must call \texttt{wsio\_unmap()} once for each call made to \texttt{wsio\_map()}. The same IOVA and length returned by \texttt{wsio\_map()} must be passed to \texttt{wsio\_unmap()}

Using \texttt{wsio\_init\_map\_context()}

IOVAs with a mapping context made through \texttt{init\_map\_context()} form a single mapping that must be unmapped together. A context is used by the mapping services to bundle IOVAs together. Do not unmmap IOVAs mapped with a mapping context until the driver is ready to unmap all IOVAs mapped with that context; then unmap all IOVAs mapped with that context together.

Each call to \texttt{wsio\_map()} with a new mapping context reserves a set of contiguous I/O PDIR entries. This allows a buffer to be mapped into a single contiguous IOVA range. It is important that drivers keep track of the current context when they map in multiple pages. If each page is mapped with a new context, many entries in the I/O PDIR will be wastefully reserved.
Understanding HP-UX I/O Subsystem Features

DMA Mapping

**Using `wsio_remap()`**

The `wsio_remap()` function allows I/O PDIR entries to be reused. A driver can avoid the cost of calling `wsio_unmap()` to return an I/O PDIR entry, only to call `wsio_map()` at a later time. Because `wsio_remap()` does not take hints, use it to remap the same type of data for which the IOVAs were originally mapped.

Drivers should not use the `wsio_remap()` function as a method for reserving or hoarding I/O PDIR entries to guarantee their availability for that driver.

The `wsio_remap()` function may return a different IOVA than the IOVA the driver passes as a parameter. When this happens, the driver must use the new IOVA appropriately; the original IOVA is no longer valid.

**Using `wsio_set_attributes()`**

Use the `wsio_set_attributes()` function to give information to the mapping services about the type of I/O the driver does. This allows them to optimize their allocation method for the type of I/O. Specifically, this allows them to allocate I/O PDIR entries out of a pool of entries that is optimized for the type of driver you specify in the `wsio_set_attributes()` call.

The `wsio_set_attributes()` function provides two attributes, `IO_INTERLEAVED_DMA` and `IO_NONINTERLEAVED_DMA`.

Drivers for devices, such as SCSI controllers, which can stream data from several disks at a time that interleave many I/O requests should use the `IO_INTERLEAVED_DMA` attribute. This is the default.

Drivers for devices that are likely to satisfy one I/O request at a time should use the `IO_NONINTERLEAVED_DMA` attribute. However, if this attribute is set and the driver calls `wsio_map()` with a non-NULL `map_cb` argument, the services behave as if the driver is interleaving its requests.

Hybrid drivers, which can fit both the interleaved and noninterleaved models, should choose the `IO_NONINTERLEAVED` attribute. This becomes their default behavior which they can override when setting up an interleaved DMA transfer.
4 Multiprocessing
Multiprocessing

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

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

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

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

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

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

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

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

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

Timing Hazards and Idle Time

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

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

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

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

Spinlock Routines

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

- alloc_spinlock() - Allocate and initialize a spinlock resource.
- cspinlock() - Conditionally lock a spinlock if the spinlock is not owned.
- dealloc_spinlock() - Deallocate a spinlock resource.
- owns_spinlock() - Check if the processor owns a spinlock.
- spinlock() - Acquire (lock) a spinlock.
- spinunlock() - Release (unlock) a spinlock.
Beta Semaphores

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

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

Beta Semaphore Routines

HP-UX provides the following beta semaphore routines. Refer to the HP-UX Driver Development Reference for detailed descriptions.

- **b_cpsema()** - Conditionally acquire (lock) a beta semaphore if it is not currently locked.
- **b_initsema()** - Initialize a beta semaphore.
- **b_owns_sema()** - Test whether a beta semaphore is owned by the current thread.
- **b_psema()** - Acquire (lock) a beta semaphore.
- **b_vsema()** - Release (unlock) a beta semaphore.
Deadlocks

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

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

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

Rules for Lock Acquisition

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

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

Lock Orders

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

In addition to spinlocks and beta semaphores, HP-UX provides another synchronization mechanism using the system services sleep() and wakeup(). Typically, the upper half of a driver will start an asynchronous activity and wait for the lower half to complete the activity. The system service sleep() is called by the driver’s upper half to block the kernel thread and put it on a sleep queue. The driver’s lower half calls wakeup() to take the kernel thread off the sleep queue and to awaken the thread.

A race condition exists between the time a kernel thread calls sleep() and the time wakeup() is called. The wakeup() routine can be called before the kernel thread has been put on a sleep queue. To handle this race condition, a call to get_sleep_lock() must be made before calling sleep().

The routine get_sleep_lock() acquires a sleep queue spinlock that is later released by sleep() after the kernel thread has been put on the sleep queue. The routine wakeup() acquires the same sleep queue spinlock before taking the kernel thread off the sleep queue and awakening the sleeping thread. Drivers typically call get_sleep_lock(), start an asynchronous activity, then call sleep() as shown below:

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

When the asynchronous activity completes, the driver’s async_completion() routine calls wakeup() as follows:

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

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

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

Notice that the incrementing of activity_count is protected by a sleep queue spinlock. When the asynchronous activity completes, the driver's async_completion() routine must call get_sleep_lock() before it decrements activity_count and calls wakeup().

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

The routine wakeup() has a special provision to allow the sleep queue lock to be acquired and held across a call to wakeup(). After the call to wakeup() in the example above, the sleep queue lock must be unlocked.
Multiprocessing

Synchronization Using sleep() and wakeup()
5 Writing a Driver
This chapter describes the code most device drivers need to include, and the things you need to do to write a driver.
Suggested Driver Writing Methodology

Read this chapter, and then read the chapter that provides details about your specific driver. For example, if you are writing PCI Drivers, read Chapter 8, “Writing PCI Device Drivers.” If you are writing a loadable driver module, also read Chapter 10, “Developing Dynamically Loadable Kernel Modules.”

Writing a Driver: Step-by-Step

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

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

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

The name is required in four places:

- In a master file in the /usr/conf/master.d directory, described in “Step 2: Choosing System Header Files”.
- In the system configuration file, /stand/system, described in “Step 3: Modify the System File” on page 166, Chapter 6.
- In the name field of the drv_info_t structure, described in “The drv_info_t Structure Type”.
- As the prefix of the name of the installation routine, driver_install, described in “Writing a driver_install() Routine”.

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

For example, if your company is Wonderful Software, and you are writing a MUX driver, you could give your driver the name wonderMUX. When it installs your driver, the kernel will call a routine named wonderMUX_install().

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

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

System-Defined Header Files

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

NOTE

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

Header Files for All Drivers

- /usr/include/sys/buf.h
  
  The buf I/O buffer structure.

- /usr/include/sys/conf.h
  
  Device switching tables and the drv_ops_t and drv_info_t structures.

- /usr/include/sys/errno.h
  
  Error numbers returned to applications.

- /usr/include/sys/file.h
  
  Flags for the open() system call.

- /usr/include/sys/io.h
  
  The ISC table.

- /usr/include/sys/malloc.h
  
  Things needed for acquiring and releasing memory.
Writing a Driver

Step 2: Choosing System Header Files

- `/usr/include/sys/sysmacros.h`
  Some commonly used fields in some drivers’ minor numbers.

- `/usr/include/sys/uio.h`
  The `uio` structure and its elements.

- `/usr/include/sys/user.h`
  Things used by some kernel routines.

- `/usr/include/sys/wsio.h`
  Data and macros used in the WSIO context, including the `wsio_drv_info` and `wsio_drv_data` structures. The header of each driver and each WSIO dependent pseudo driver must include this header file.

Header Files for Disk Drivers

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

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

Header Files for Tape Drivers

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

When you examine some system header files, you will find that they include different files, depending on the presence of a kernel flag. For example, with the flag, they specify “#include “../h/buf.h””; without the flag, they specify “#include <sys/buf.h>”.

For historical reasons, driver modules were built using a separate set of system header files. The driver modules were built in parallel directories, and the header files were referenced with relative path names, such as “#include “../h/buf.h””. Currently, the system header files used by driver modules are distributed in the standard location, `/usr/include`, for which the corresponding reference is “#include <sys/buf.h>”. Since this conversion is not complete (a number of headers in `/usr/include` still use relative path names to include other headers), those header files and others you will need are also distributed in the `/usr/conf` directory tree.
In order to facilitate the conversion, we recommend that you reference system header files with the <...> notation. Also see “Step 1: Compile Your Driver” on page 157, Chapter 6.

NOTE

Some kernel header files include “../wsio/eeprom.h”. Unfortunately, this file is not shipped in that (“/usr/conf/wsio”) location. You can avoid problems by simply copying /usr/conf/machine/eeprom.h to /usr/conf/wsio.
Step 3: Defining Installation Structures

You must include some data structures in your driver. Which data structures your driver requires depends on what your device or interface-card driver does. They can appear in either the .c file of the driver, or in a header file that you include in the driver.

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

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

The **drv_ops_t** Structure Type

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

```c
typedef struct drv_ops {
    int (*d_open)(); /* block and character */
    int (*d_close)(); /* block and character */
    int (*d_strategy)(); /* block */
    int (*d_dump)(); /* NULL (obsolete) */
    int (*d_psize)(); /* block */
    int (*reserved0)(); /* NULL */
    int (*d_read)(); /* character */
    int (*d_write)(); /* character */
    int (*d_ioctl)(); /* character */
    int (*d_select)(); /* character */
    int (*d_option1)(); /* NULL */
    pfilter_t *pfilter; /* block and character */
    int (*reserved1)(); /* NULL */
    int (*reserved2)(); /* NULL */
    int (*reserved3)(); /* NULL */
    int d_flags; /* block and character */
} drv_ops_t;
```
The relevant fields are described in Table 5-1. All other fields in `drv_ops_t` should be `NULL`. Except as noted, entry points that do not apply to your driver or that your driver does not provide (for example, `d_psize()` has no meaning for a printer) should be `NULL`.

**Table 5-1** Device Driver Fields in the `drv_ops_t` Structure Type

<table>
<thead>
<tr>
<th>Field</th>
<th>Device Type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>d_open()</code></td>
<td>both</td>
<td>Pointer to your <code>driver_open()</code> routine, which enables a device for subsequent operations. If the device is off line, does not exist, or cannot be accessed, an error should be returned. When appropriate, it is permissible for the open routine to do nothing.</td>
</tr>
<tr>
<td><code>d_close()</code></td>
<td>both</td>
<td>Pointer to your <code>driver_close()</code> routine, which disables interrupts, resets a device, frees resources, and performs other tasks required when a device is closed.</td>
</tr>
<tr>
<td><code>d_strategy()</code></td>
<td>block</td>
<td>Pointer to your <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><code>d_psize()</code></td>
<td>block</td>
<td>Pointer to your <code>driver_psize()</code> routine. For a swapping device, it should return the size of the swap partition. Consider writing this routine only if your device is used for swapping.</td>
</tr>
<tr>
<td><code>d_read()</code></td>
<td>character</td>
<td>Pointer to your <code>driver_read()</code> routine, which should return the requested data transferred from the device.</td>
</tr>
</tbody>
</table>

Chapter 5 81
Step 3: Defining Installation Structures

Table 5-1 Device Driver Fields in the `drv_ops_t` Structure Type

<table>
<thead>
<tr>
<th>Field</th>
<th>Device Type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>d_write()</td>
<td>character</td>
<td>Pointer to your <code>driver_write()</code> routine, which should write the requested data to the device.</td>
</tr>
<tr>
<td>d_ioctl()</td>
<td>character</td>
<td>Pointer to your <code>driver_ioctl()</code> routine, which sends control information to, or gets it from, a device. You can also use it to provide driver-dependent functions that are not implemented by other routines.</td>
</tr>
<tr>
<td>d_select()</td>
<td>character</td>
<td>Pointer to your <code>driver_select()</code> routine, which you can use to test for I/O completion and driver-dependent exception conditions. If your device is always ready for reading or writing, you can put <code>seltrue</code> in the <code>d_select()</code> field. If you do, calls to <code>select()</code> always return true without invoking your driver.</td>
</tr>
<tr>
<td>pfilter</td>
<td>both</td>
<td>Pointer to a <code>pfilter_t</code> structure. Use the <code>&amp;cpd_pfilter</code> 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 <code>&amp;cpd_pfilter</code> pointer is required for such disks; it is ignored under other conditions (or you can use <code>NULL</code>).</td>
</tr>
</tbody>
</table>
The `drv_info_t` Structure Type

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

---

**Table 5-1** Device Driver Fields in the `drv_ops_t` Structure Type

<table>
<thead>
<tr>
<th>Field</th>
<th>Device Type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>d_flags</td>
<td>both</td>
<td>The bitwise OR of flag values that indicate special features of the device. The flags give information about the device to the kernel. Drivers receive this information, but usually only validate it. Use 0 if no flags are set. The flags used by drivers are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1. <code>C_ALLCLOSES</code> flag forces a call to <code>driver_close()</code> on every closing of the device. The default action is to call the driver's close routine only on the last close of the device.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. <code>C_NODELAY</code> tells the kernel to not wait for a write request to complete on this device. The default action is to wait for a write request to complete before returning control to the calling process.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3. <code>C_MGR_IS_MP</code> identifies the driver as safe for use in a multiprocessing environment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4. <code>C_MAP_BUFFER_TO_KERNEL</code> indicates the device driver needs <code>physio()</code> to remap a user buffer to kernel space prior to calling the <code>driver_strategy()</code> routine. This is the pre 10.0 behavior of <code>physio()</code> .</td>
</tr>
</tbody>
</table>
### Writing a Driver

**Step 3: Defining Installation Structures**

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

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Pointer to a string containing the name of the driver. This is the name you defined in “Step 1: Choosing a Driver Name”.</td>
</tr>
<tr>
<td>class</td>
<td>Pointer to a string containing the name of the class that the driver is in. For interface drivers, instances of a card are enumerated within each class as they are identified by the kernel at boot time.</td>
</tr>
<tr>
<td>flags</td>
<td>The bitwise OR of flag values that describe the driver.</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 it is not a block device.</td>
</tr>
<tr>
<td>c_major</td>
<td>The major number if this is a character device. Set it to</td>
</tr>
</tbody>
</table>
Writing a Driver
Step 3: Defining Installation Structures

NOTE
The values you specify above for b_major and c_major override the values you enter in a master file in /usr/conf/master.d (see “Step 2: Create a Master File” on page 160, Chapter 6.)

The *wsio_drv_data_t* Structure Type

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

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

The fields are described below.

Field                  Purpose

**drv_path**            Follow these guidelines:

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

**drv_type**          One of the following values:

- **T_DEVICE** - The driver controls a hardware device.
Writing a Driver

Step 3: Defining Installation Structures

- **T_INTERFACE** - The driver controls an interface card, or is monolithic.

**drv_flags** One of the following values:

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

**drv_minor_build** Pointer to your minor number formatter.

**drv_minor_decode** Pointer to your minor number interpreter.

The `wsio_drv_info_t` Structure Type

The `wsio_drv_info_t` structure type, defined in `<sys/wsio.h>` and shown below, contains pointers to the three preceding data structures.

typedef struct wsio_drv_info
{
    drv_info_t     *drv_info;
    drv_ops_t      *drv_ops;
    wsio_drv_data_t *drv_data;
} wsio_drv_info_t;

Sample Header for a Device Driver

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

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

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

static drv_ops_t skel_ops =
{
    skel_open,
    skel_close,
```
skel_strategy,
NULL,
skel_psize,
NULL,
skel_read,
skel_write,
skel_ioctl,
NULL,
NULL,
NULL,
NULL,
NULL,
 NULL,
C_ALLCLOSES | C_MGR_IS_MP
};

static drv_info_t skel_info =
{
"skel",
"disk",
DRV_CHAR | DRV_BLOCK | DRV_SAVE_CONF | DRV_MP_SAFE,
-1,   /* dynamic major number assignment */
-1,   /* for block and character devices */
NULL,
NULL,
NULL,

};

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

};

static wsio_drv_info_t skel_wsio_info =
{
&skel_info,
&skel_ops,
&skel_data,

};
Writing a Driver

Step 3: Defining Installation Structures

Sample Header for an Interface Driver

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

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

```c
#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,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
};

static wsio_drv_data_t skel_data =
{
    "skel",
    T_INTERFACE,
    DRV_CONVERGED,
    NULL,
};
```
Step 3: Defining Installation Structures

Sample Header for a Pseudo Driver

Pseudo drivers do not control hardware, but do have character and/or block entry points in a `drv_ops_t` structure. Generally, a pseudo driver preprocesses information that it then passes to the file system or to a device driver that does control hardware. Normally, it is installed in the CDIO where the device driver is installed (generally, WSIO). In some cases, a pseudo driver works with device drivers that reside in different CDIOs. Then, it must be installed outside the specific CDIO environments. For example, LVM is a pseudo driver for both workstation (WSIO) and server (SIO) disk drives.

In the multiple CDIO case (e.g., SIO and WSIO), the pseudo driver would define the following header and structures and use the `install_driver()` installation function in `driver_install()` (see “Writing a driver_install() Routine”). Note the use of the `pseudo` class and the `DRV_PSEUDO` flag.

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

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

static drv_ops_t my_ops =
{
  my_open,
```
Writing a Driver

Step 3: Defining Installation Structures

In the WSIO case, the driver adds the following header and structure to the preceding and uses the `wsio_install_driver()` installation function in `driver_install()` (see "Writing a driver_install() Routine").

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

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

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

Table 5-2, “Configuration and Entry Point Routines,” lists the configuration and entry point routines for the various types of drivers.

<table>
<thead>
<tr>
<th>Driver Type</th>
<th>Configuration Routines</th>
<th>Entry Points</th>
<th>Other Routines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block a Device</td>
<td>driver_install()</td>
<td>driver_open()</td>
<td>driver_strategy()</td>
</tr>
<tr>
<td></td>
<td>driver_dev_init()</td>
<td>driver_close()</td>
<td>driver_strateg()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_strategy()</td>
<td></td>
</tr>
<tr>
<td>Block Pseudo</td>
<td>driver_install()</td>
<td>driver_open()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>driver_dev_init()</td>
<td>driver_close()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_strategy()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_psize()</td>
<td></td>
</tr>
<tr>
<td>Character Device</td>
<td>driver_install()</td>
<td>driver_open()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>driver_dev_init()</td>
<td>driver_close()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_read()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_write()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_ioctl()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_select()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_strategy()</td>
<td></td>
</tr>
<tr>
<td>Character Pseudo</td>
<td>driver_install()</td>
<td>driver_open()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>driver_dev_init()</td>
<td>driver_close()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_read()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_write()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_ioctl()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_select()</td>
<td></td>
</tr>
<tr>
<td>Interface</td>
<td>driver_install()</td>
<td>None c</td>
<td>driver_isr()</td>
</tr>
<tr>
<td></td>
<td>driver_attach()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>driver_if_init()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>driver_probe()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block Monolithic</td>
<td>driver_install()</td>
<td>driver_open()</td>
<td>driver_isr()</td>
</tr>
<tr>
<td></td>
<td>driver_attach()</td>
<td>driver_close()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>driver_dev_init()</td>
<td>driver_strategy()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>driver_strateg()</td>
<td>driver_psize()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>driver_probe()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

*Step 4: Identifying Routines for Your Driver*

Table 5-2, “Configuration and Entry Point Routines,” lists the configuration and entry point routines for the various types of drivers.

<table>
<thead>
<tr>
<th>Driver Type</th>
<th>Configuration Routines</th>
<th>Entry Points</th>
<th>Other Routines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block a Device</td>
<td>driver_install()</td>
<td>driver_open()</td>
<td>driver_strategy()</td>
</tr>
<tr>
<td></td>
<td>driver_dev_init()</td>
<td>driver_close()</td>
<td>driver_strateg()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_strategy()</td>
<td></td>
</tr>
<tr>
<td>Block Pseudo</td>
<td>driver_install()</td>
<td>driver_open()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>driver_dev_init()</td>
<td>driver_close()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_strategy()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_psize()</td>
<td></td>
</tr>
<tr>
<td>Character Device</td>
<td>driver_install()</td>
<td>driver_open()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>driver_dev_init()</td>
<td>driver_close()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_read()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_write()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_ioctl()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_select()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_strategy()</td>
<td></td>
</tr>
<tr>
<td>Character Pseudo</td>
<td>driver_install()</td>
<td>driver_open()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>driver_dev_init()</td>
<td>driver_close()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_read()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_write()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_ioctl()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_select()</td>
<td></td>
</tr>
<tr>
<td>Interface</td>
<td>driver_install()</td>
<td>None c</td>
<td>driver_isr()</td>
</tr>
<tr>
<td></td>
<td>driver_attach()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>driver_if_init()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>driver_probe()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block Monolithic</td>
<td>driver_install()</td>
<td>driver_open()</td>
<td>driver_isr()</td>
</tr>
<tr>
<td></td>
<td>driver_attach()</td>
<td>driver_close()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>driver_dev_init()</td>
<td>driver_strategy()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>driver_strateg()</td>
<td>driver_psize()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>driver_probe()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Choosing the Routines You Need

Your driver may not need all the routines shown in Table 5-2, on page 91. Choose the ones you need from the following description.

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

Driver Type

These are the principal driver types. The block and character types can be combined into the same driver, in which case the driver can specify the entry points for both types. (See Table 5-3 and Table 5-4, below.)

Configuration Routines

These routines are executed when the system boots.

- Every driver requires a `driver_install()` routine.
Writing a Driver

Step 4: Identifying Routines for Your Driver

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

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

Step 4: Identifying Routines for Your Driver

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

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

- `driver_isr()` is the interrupt service routine for a device or interface, established by the interface driver.
- `driver_strategy()` for a character driver can be

<table>
<thead>
<tr>
<th>Table 5-3 For a Block Driver</th>
<th>System Call</th>
<th>Executes</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>open()</td>
<td><code>driver_open()</code></td>
<td>Open the device</td>
<td></td>
</tr>
<tr>
<td>close()</td>
<td><code>driver_close()</code></td>
<td>Close the device</td>
<td></td>
</tr>
<tr>
<td>read()</td>
<td><code>driver_strategy()</code></td>
<td>Perform block read</td>
<td></td>
</tr>
<tr>
<td>write()</td>
<td><code>driver_strategy()</code></td>
<td>Perform block write</td>
<td></td>
</tr>
<tr>
<td>kernel</td>
<td><code>driver_psize()</code></td>
<td>Specify swap partition size</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 5-4 For a character Driver</th>
<th>System Call</th>
<th>Executes</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>open()</td>
<td><code>driver_open()</code></td>
<td>Open the device</td>
<td></td>
</tr>
<tr>
<td>close()</td>
<td><code>driver_close()</code></td>
<td>Close the device</td>
<td></td>
</tr>
<tr>
<td>read()</td>
<td><code>driver_read()</code></td>
<td>Perform character read</td>
<td></td>
</tr>
<tr>
<td>write()</td>
<td><code>driver_write()</code></td>
<td>Perform character write</td>
<td></td>
</tr>
<tr>
<td>ioctl()</td>
<td><code>driver_ioctl()</code></td>
<td>Perform special command</td>
<td></td>
</tr>
<tr>
<td>select()</td>
<td><code>driver_select()</code></td>
<td>Test I/O completion</td>
<td></td>
</tr>
</tbody>
</table>
accessed from the `driver_read()` and `driver_write()` routines to transfer data in blocks rather than byte by byte using `physio()`. In a combination block and character driver, the two `driver_strategy()` routines can often be combined.

- `driver_minphys()` is used for a character driver which calls `physio()` and requires that subsequent `driver_strategy()` calls be made with different size buffers than that provided by the standard `minphys()` routine.
Step 5: Writing Configuration Routines

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

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

Configuration Overview

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

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

1. For each driver, the kernel calls the driver's install routine.

   (The `driver_dev_init()`, `driver_if_init()`, and `driver_probe()` routines are optional for all drivers. A `driver_attach()` routine is required for any interface driver that intends to claim an interface card.)

   The install routines usually perform the following operations:

   a. All install routines call `wsio_install_driver()` or `install_driver()`, which registers the driver with the system.

   b. If a `devd_dev_init()` routine exists, `devd_install()` puts the routine on the applicable chain of init routines for that type of...
Writing a Driver

Step 5: Writing Configuration Routines

device.

c. If a pseudod_dev_init() routine exists, pseudod_install() puts the routine on the applicable chain of init routines for that type of device.

d. ifd_install() puts ifd_attach() on the applicable chain(s) of attach routines for supported interfaces (e.g., the eisa_attach() and the pci_attach() routines for a driver which supports both EISA and PCI bus interface cards.) If it has an ifd_probe() routine, it registers the routine with the system.

e. monod_install() puts monod_attach() on the applicable chain of attach routines. If it has a monod_dev_init() routine, it puts the routine on a chain of init routines. If it has a monod_probe() routine, it registers the routine with the system.

2. The kernel has installed all the drivers.

   The driver’s static or dynamically assigned major number is known at this point.

3. The kernel identifies the hardware on the system. For each interface card, the kernel calls the first entry on an attach chain that contains ifd_attach() or monod_attach(). Each entry on the chain is responsible for calling the next entry on the chain.

   When called, ifd_attach() or monod_attach() lays claim to the hardware it controls. For each card an interface driver claims, the kernel allocates an ISC structure, and the driver can set up its isc->gfsw->init to point to its card-initialization routine, ifd_if_init() or monod_if_init().

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

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

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

Step 5: Writing Configuration Routines

Writing a driver_install() Routine

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

The driver_install() routine has the following tasks:

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

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

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

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

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

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

Those values are:

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

If it fails, the appropriate message below appears on the system console and in the system's error-log file. *driver* is the name of your driver.

- *wsio_install_driver*: Install of driver *driver* failed.
- *install_driver*: Install of driver *driver* failed.

**driver_install() Example for Device and WSIO Pseudo Drivers**

Installation routine for a driver named *skel*.

The *skel_dev_init()* device init routine is added to the head of the *dev_init()* global chain.

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

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

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

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

**driver_install() Example for Non-WSIO Pseudo Drivers**

Installation routine for a pseudo driver named *pseu* that must be installed outside the WSIO environment.

The *pseu_dev_init()* device init routine is added to the head of the
Writing a Driver  

Step 5: Writing Configuration Routines

dev_init() global chain.

int (**pseu_saved_dev_init)(); /* save previous init head */

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

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

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

driver_install() Example for Interface Drivers

Installation routine for a driver named skel.

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

int (**skel_saved_attach)(); /* save previous attach head */

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

    skel_saved_attach = pci_attach; /* save head pointer */
    pci_attach = skel_attach; /* make my attach the head */

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

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

Chapter 5

Step 5: Writing Configuration Routines

driver_install() Example for Monolithic Drivers

Installation routine for a driver named skel.

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

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

int
skel_install(void)
{
    /* ptr to pci attach chain */
    extern int (*dev_init)(); /* head pointer for init chain */

    skel_saved_attach = pci_attach; /* save head pointer */
    pci_attach = skel_attach; /* make my attach the head */

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

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

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

Writing a driver_attach() Routine

The driver_attach() function is provided by the driver writer. It can have any unique name. The driver registers its attach routine with the WSIO by inserting it in the appropriate attach chains during driver installation. Commonly, driver is replaced by your driver’s name.
The kernel searches the I/O backplane for hardware. When it finds a device, it first does preliminary initialization. Then it calls the driver_attach() routine at the head of the corresponding attach chain, e.g., pci_attach(), that was created by the driver_install() routine.

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

This algorithm allows multifunction EISA cards to be claimed by more than one interface driver.

If the isc->ftn_no field is not -1 in the received ISC for a multifunction EISA card, the driver_attach() routine should call get_new_isc() to allocate a new ISC structure for the driver’s functions and set the isc->ftn_no field of the new ISC structure to the function number for its portion of the card. Then it should pass the new ISC on to the next driver in the attach chain. See get_new_isc(WSIO3) in HP-UX Driver Development Reference.

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

**driver_attach() Return Value**

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

**driver_attach() Diagnostics**

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

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

  The driver_attach() routine has access to the card registers via the isc pointer passed into it. The routine may verify that data any way it sees fit.
Bad driver. A `driver_attach()` routine returned without calling the next `driver_attach()` routine in the chain. The system will panic and display the message:

```
bad driver in kernel
```

**driver_attach() Examples**

```c
static int skel_attach(int product_id, struct isc_table_type *isc_ptr) {
    int MY_PRODUCT_ID = 0x1234;
    /* the four-byte value for this driver */

    /*
    * Check and make sure the product_id belongs to me.
    */
    if (product_id != MY_PRODUCT_ID) /* not mine */
        return (*skel_saved_attach)(product_id, isc_ptr);

    /*
    * memory allocation, save pointers, other housekeeping.
    * ...
    */

    /*
    * Claim the driver (sets INITIALIZED in
    * isc_ptr->if_info->flags)
    */
    isc_claim(isc_ptr, &skel_wsio_info);

    /*
    * Save my initialization routine (if I have one).
    */
    isc_ptr->gfsw->init = skel_if_init;

    /*
    * Chain to the next driver’s attach routine.
    */
    return (*skel_saved_attach)(product_id, isc_ptr);
}
```

**Writing a driver_dev_init() Routine**

The `driver_dev_init()` routine is provided by the driver writer. It can have any unique name. You pass the name to WSI O Services by linking it.
Writing a Driver

Step 5: Writing Configuration Routines

into an *init chain from your driver_install() routine.*

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

int
sdisk_install(void)
{
    sdisk_link();

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

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

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

**Writing a driver_if_init() Routine**

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

```c
static int
mydriver_attach(uint32_t card_id, struct isc_table_type *isc)
{
    if (card_id == my_card_id) {
        /* if mydriver has an init routine */
        isc->gfsw->init = mydriver_init;
        ...}
    }
```
static int
mydriver_init(struct isc_table_type *isc)
{
    ...
    return 0; /* success initializing interface */
}
Writing Driver Probe Routines

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

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

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

Most drivers will only use the service `wsio_register_dev_probe()` to register a probe function based on either the class of devices the driver controls or the driver’s name as specified in the name field of the `dev_info_t` structure.

The second service `wsio_register_addr_probe()` is used to associate an additional probe function based on the name of the driver as specified in the driver’s `drv_info_t` structure.

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

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

### `wsio_register_dev_probe()`

The calling semantics for `wsio_register_dev_probe()` are:

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

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

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

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

```c
int sctl_install(void)
{
    /*
    * register the scsi probe function
    * with the wsio cdio.
    */
    wsio_register_dev_probe(IF_CLASS, scsi_probe, "scsi");

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

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

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

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

<table>
<thead>
<tr>
<th>Table 5-5 Driver Probe Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
</tr>
<tr>
<td>drv_info</td>
</tr>
<tr>
<td>probe_id</td>
</tr>
</tbody>
</table>
Drivers can register an additional address probe function by calling the WSIO service `wsio_register_addr_probe()`. This service is used to associate an additional probe function based on the driver's name. The name must match the "name" field of the driver's "drv_info_t" structure.

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

```c
int c720_install(void) {
    int ret;

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

<table>
<thead>
<tr>
<th>hw_path</th>
<th>IN</th>
<th>The hardware path of the last device found</th>
</tr>
</thead>
<tbody>
<tr>
<td>isc</td>
<td>IN</td>
<td>A pointer to the isc_table_type structure</td>
</tr>
<tr>
<td>probe_type</td>
<td>IN</td>
<td>The type of hardware probe. There are three types described above.</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>desc</td>
<td>OUT</td>
<td>A pointer to a string with the device description. Thi is driver dependent</td>
</tr>
</tbody>
</table>
Step 5: Writing Configuration Routines

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

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

**Table 5-6 Address Probe Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
<td>IN</td>
<td>A pointer to the io_tree_node</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>
<tr>
<td>drv_info</td>
<td>IN</td>
<td>A pointer to the drv_info_t structure</td>
</tr>
<tr>
<td>probe_id</td>
<td>OUT</td>
<td>A unique identifier for the device found</td>
</tr>
<tr>
<td>hw_path</td>
<td>IN</td>
<td>The hardware path of the last device found</td>
</tr>
<tr>
<td>hw_path</td>
<td>OUT</td>
<td>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>probe_type</td>
<td>IN</td>
<td>The type of hardware probe. There are three types defined above</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>desc</td>
<td>OUT</td>
<td>A pointer to a string with the device description. This is driver dependent</td>
</tr>
</tbody>
</table>
Writing a Driver

Step 5: Writing Configuration Routines

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

Normally a driver would only register a single probe function by calling `wsio_register_dev_probe` to register either a probe function based on the driver's name or class. Certainly this will be the case for monolithic drivers. However there are some driver stacks which are very modular such that a device driver can be paired with several different interface drivers where each interface driver supports a different range of addresses. This is when the two WSIO services can be used to register separate probe functions that will work together.

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

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

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

Step 5: Writing Configuration Routines

Example of a driver_addr_probe() Routine

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

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

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

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

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;
```

Chapter 5  111
int node_type, parent_type;
struct io_tree_node *parent_node;
int lun, target;
struct wsio_probe_dev_info probe_dev;

/*
 * The hardware path should only contain two
 * elements at the most. A target and an LUN.
 * If first_index == last_index, we have only a
 * target address. If first_index+1 == last_index
 * we have both a target and lun, otherwise we do
 * not know how to interpret the hardware path so
 * just return.
 */
if((hw_path->first_index+1) < hw_path->last_index)
  return(0);
if(probe_func == NULL)
  return(0);
probe_dev.instance = wsio_isc_to_instance(isc,hw_path);
probe_dev.opt_1 = 0;
probe_dev.opt_2 = 0;
probe_dev.opt_3 = 0;

if(probe_type == PROBE_NEXT) {
  /*
   * If only a target was passed in then the last thing
   * found was a target so the next thing to look for
   * is the first lun of that target, so get the target
   * address and set lun to -1 so that it gets
   * automatically incremented to 0 in the switch
   * statement below. Else get the previous values for
   * target and lun from the path.
   */
  if(hw_path->first_index == hw_path->last_index) {
    target = (int)hw_path->addr[hw_path->last_index];
    hw_path->last_index++;
    lun = -1;
  } else {
    lun = (int)hw_path->addr[hw_path->last_index-1];
    target = (int)hw_path->addr[hw_path->last_index-1];
  }
  looking_for = NEXT_LUN;
}
found = FALSE;
do {
    switch(probe_type) {
    case PROBE_FIRST:
        target = lun = 0;
        hw_path->last_index++;
        looking_for = NEXT_TARGET;
        probe_type = PROBE_NEXT;
        break;
    case PROBE_NEXT:
        /*
        * If we didn’t find a device (target) or the last
        * lun used was invalid (out of range) then go
        * to the next target and scan starting
        * with lun 0.
        */
        if((found == NO_DEV) || (found == INVAL_LUN)) {
            target++;
            lun = 0;
            hw_path->last_index = hw_path->first_index;
            looking_for = NEXT_TARGET;
        } else {
            lun++;
            looking_for = NEXT_LUN;
        }
        break;
    case PROBE_ADDRESS:
        if(hw_path->first_index == hw_path->last_index) {
            target = (int)hw_path->addr[hw_path->last_ix];
            lun = 0;
            looking_for = NEXT_TARGET;
        } else {
            lun = (int)hw_path->addr[hw_path->last_index];
            target =
                (int)hw_path->addr[hw_path->last_index-1];
            looking_for = NEXT_LUN;
        }
        break;
    default:
        return 0;
    }
    probe_dev.target = target;
    probe_dev.opt_1 = lun;
    found = probe_func(isc, NULL, &probe_dev, probe_type, 
        probe_id, dev_class, desc );
    /*
Step 5: Writing Configuration Routines

* We need to keep looking if we are not looking for a particular address (probe_type == PROBE_ADDRESS) and while we haven’t run out of valid targets and either we are looking for the next target and we haven’t found a device or we are looking for the next lun and we have found a device but not a valid lun for it. */

```c
} while ((probe_type != PROBE_ADDRESS) &&
        (found != INVAL_TGT) &&
        ((looking_for == NEXT_TARGET) &&
         (found == NO_DEV)) ||
        ((looking_for == NEXT_LUN) &&
         (found != VALID_LUN)));

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

Example of the `scsi_probe()`

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

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

Chapter 5
static wsio_drv_data_t sctl_data = {
    "scsi_ctl", T_DEVICE, DRV_CONVERGED, NULL, NULL,
};

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

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

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

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

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

/*
 * Get the major number of the driver
 * and build a dev_t structure.
 */
major_num = sctl_wsio_info.drv_info->c_major;
dev = ((major_num << 24) & 0xff000000);
dev |= ((probe_dev->instance << 16) & 0x00ff0000);
dev |= ((probe_dev->target << 12) & 0x0000f000);
dev |= ((probe_dev->opt_1 << 8) & 0x00000f00);
found = NO_DEV;
if ((sctl_open(dev)) == 0) {
    if ((sctl_ioctl(dev, SIOC_INQUIRY, &iqr_data, 0)) == 0) {
        dev_type = iqr_data.inq2.dev_type;
        if (dev_type != NO_SCSI_DEV) {
            found = VALID_TGT;
            /*
             * Get first four bytes of the inquiry data
             * for dev_id and the vendor and product id for
             * desc. and set dev class according to the
             * dev_type.
             */
            *(int *)dev_id = *((int *)&iqr_data);
            strncpy(desc, iqr_data.inq2.vendor_id, 8);
            strncpy(&desc[8], iqr_data.inq2.product_id, 16);
            desc[24] = '\0';
            switch(dev_type) {
            case SCSI_PROCESSOR:
                strcpy(dev_class, "processor");
                break;
            case SCSI_DIRECT_ACCESS:
            case SCSI_CDROM:
            case SCSI_MO:
            case SCSI_WORM:
                strcpy(dev_class, "disk");
                break;
            case SCSI_SEQUENTIAL_ACCESS:
                strcpy(dev_class, "tape");
            } // End switch
        } // End if
    } // End if
} // End if
Writing a driver

Step 5: Writing Configuration Routines

break;
case SCSI_AUTOCHANGER:
    strcpy(dev_class, "changer");
    break;
case SCSI_PRINTER:
    strcpy(dev_class, "printer");
    break;
case SCSI_SCANNER:
    strcpy(dev_class, "scanner");
    break;
case SCSI_COMMUNICATIONS:
    strcpy(dev_class, "comm");
    break;
} /* switch */

/*
 * We know we found a valid device now lets
 * make sure that it is also a valid LUN.
 */
if (iqr_data.inq2.periph_qualifier == 0)
    found = VALID_LUN;

} /* dev_type != NO_SCSI_DEV */
} /* sctl_ioctl */
sctl_close(dev);
} /* sctl_open */

return found;
} /* sctl_probe */

Writing a driver_minor_build() routine

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

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
Writing a Driver

Step 5: Writing Configuration Routines

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

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

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

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

Writing a `driver_open()` Routine

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

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

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

- Opens the device.
- Returns an error if the device is an exclusive-open device and this is not the first open.
- Allocates a set of data structures, if the device is a multiple-open device.
- Finds the device (determines the hardware location, based on the `dev_t` value passed to it).
- Sets a flag indicating the device is open.
- Initializes data structures as needed.
- Initializes hardware (puts the device in a known state).
- Clears the open flag and returns an error if an error occurs during initialization.
Step 6: Writing Entry Point Routines

- Returns zero if the open was successful.

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

- **Exclusive Open**

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

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

- **Shared Open**

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

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

- **Multiple Open**

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

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

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

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

- **dev**

  The dev_t device number of the file to be opened. (See “Major and Minor Numbers” on page 39, Chapter 3.) The driver_open() routine can extract the major and minor numbers from the device number. (See
Writing a Driver

Step 6: Writing Entry Point Routines

major(KER2) and minor(KER2) in the HP-UX Driver Development Reference.) See NOTE below.

flag

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

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

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

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

NOTE

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

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

If the driver_open() routine is successful, the kernel’s open() call returns a file descriptor to the user. If it is unsuccessful, the kernel returns -1 to the user and sets errno to the value returned by the driver_open() routine. The user’s process can check the returned value
Writing a Driver

Step 6: Writing Entry Point Routines

and `errno` to determine whether an error occurred. See the `<errno.h>` header file for possible values for `errno`.

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

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

Sample `driver_open()` Routine for a Device Driver

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

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

    wsio_get_isc(dev, &isc, &CharDrv_wsio_info);

    /* our device? */
    if (isc == NULL || isc->if_id != CENT_SV_ID)
        return ENXIO;
    if (DevIsOpen) /* is the flag nonzero? */
      return EACCES;

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

    /* reserve the device */
```
Writing a driver _close() Routine

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

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

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

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

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

- **flag** A value corresponding to the flag field in the driver _open() call. (See the parameter description in “Writing a driver _open() Routine” for values that can appear in the flag parameter.)

The kernel always returns success (0) to the caller of close(), ignoring the return value it gets from the driver. Therefore, the driver _close() routine need not return a valid value. However, to avoid problems (as with strict compiler return value checking), the driver _close() routine could return some integer value (such as 0).

Writing a driver _close() Routine for an Exclusive-Open Device

The driver _close() routine for an exclusive-open is simple. Because only one process at a time can open the device, the kernel invokes the

```c
DevIsOpen++;
return 0;
}
Writing a Driver

Step 6: Writing Entry Point Routines

driver_close() routine each time a process calls the close() system call. The driver_close() routine for such a device does the following:

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

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

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

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

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

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

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

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

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

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

Step 6: Writing Entry Point Routines

**Chapter 5**

`driver_close()` once for each number.

3. A file must be unmounted before a block-device file is closed. If the file is mounted, the `close()` system call returns control to the calling process without calling the `driver_close()` routine.

4. The device must remain open until all active inodes for the device are closed. If more than one active inode can exist for the same device, (if two device files with the same major number can access the same device), the driver should set the `C_ALLCLOSES` flag and maintain its own count of how many processes have the device open. Note that setting this flag does not guarantee that the kernel will call the `driver_close()` routine for every `close()` call for the device, but only when the link count in the inode is 0 or 1. It will not call the `driver_close()` routine, for instance, when a forked child process calls `close()` for an open-file descriptor it has inherited.

**Using the C_ALLCLOSES Flag**

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

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

Without the `C_ALLCLOSES` flag, the kernel only calls your `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 your `driver_close()` routine on every `close()` system call. This allows you to keep track of complex device structures, such as a device with two device files having the same major number.

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

**Sample `driver_close()` Routine**

This example is for a character driver that controls a Centronics interface. See “Sample driver_open() Routine for a Device Driver” on page 122, for the corresponding `driver_open()` routine.

```c
static int CharDrv_close(dev_t dev) {
```

```c
```
Writing a Driver

Step 6: Writing Entry Point Routines

```c
struct isc_table_type *isc;
CentIfSwitch_t *ifsw;

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

/* invoke interface driver shutdown routine */
wsio_get_isc(dev, &isc, &CharDrv_wsio_info);
ifsw = (CentIfSwitch_t *)isc->ifsw;
if (((*ifsw->dev_end)(isc) != 0) {
    return EIO;
}

/* release the device */
DevIsOpen--;
return 0;
```

Writing a `driver_read()` or `driver_write()` Routine

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

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

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

Character devices typically need different processing for read requests and write requests, so they can have separate routines for reading and writing operations. If character devices share a great deal of code, common code can be combined into a single “strategy” routine that both
Writing a Driver

Step 6: Writing Entry Point Routines

driver_read() and driver_write() call.

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

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

You can implement a driver_read() routine in two ways:

• Call physio() with the appropriate parameters, allowing the driver_strategy() routine to complete the request. If you use physio(), you also need to write a driver_strategy() routine.

• The driver strategy routine is passed as a parameter to physio().

• Use uiomove() to buffer the data and then to complete the request. If you use uiomove(), the driver_read() routine does the following:

  1. Initializes data structures.
  2. Sets a flag indicating that I/O is in progress.
  3. Requests an I/O operation from the device.
  4. Waits or sleeps while the device completes the I/O operation.
  5. Calls uiomove() to transfer the data from the kernel's buffer to the user's buffer.
  6. Returns a value to the read() or readv() call.

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

• Use physio() and a driver_strategy() routine.

• Use uiomove(), in which case the driver_write() routine does the following:

  1. Initializes data structures.
  2. Calls uiomove() to copy the data into kernel space.
  3. Sets a flag indicating that I/O is in progress.
Writing a Driver
Step 6: Writing Entry Point Routines

4. Requests that the device start the I/O operation.
5. Waits or sleeps while the device completes the I/O operation.
6. Returns a value.

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

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

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

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

- `uio` A pointer to a `uio` structure. See “System-Defined Header Files” on page 77. The `uio` structure contains information about the data being read or written.

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

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

Using physio()

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

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

```c
int physio(int(*strat)(), struct buf *bp, dev_t dev, int flag, void(*mincnt)(), struct uio *uiop);
```
The \texttt{driver\_strategy()} routine, which sets up an I/O request.

\*\texttt{bp}  
A pointer to a \texttt{buf} structure. If the pointer is \texttt{NULL}, \texttt{physio()} allocates a \texttt{buf} structure from the file system's buffer cache.

Alternatively, the driver can allocate its own \texttt{buf} structure and pass a pointer to it.

\texttt{dev}  
The device number.

\texttt{flag}  
A read-write flag. Set the value to \texttt{B\_READ} for a read request; set it to \texttt{B\_WRITE} for a write request.

\texttt{mincnt}  
The routine that divides a data transfer that is larger than the system's maximum size for a single request (determined by the \texttt{block\_size} system parameter) into several requests to a driver, each request no larger than the system's maximum size. You can use the kernel's \texttt{minphys()} routine, which most drivers use, or you can write your own. The \texttt{physio()} routine sends the requests to the routine specified in \texttt{strat}.

\texttt{uio}  
Pointer to the \texttt{uio} structure.

The \texttt{physio()} routine handles the I/O transfer for the \texttt{driver\_read()} and \texttt{driver\_write()} routines as described in \texttt{physio(KER2)} in the HP-UX Driver Development Reference.

\texttt{physio()} locks the user's data area so that it cannot be swapped out during the transfer. Then for each \texttt{mincnt} size chunk to be transferred, \texttt{physio()} calls the \texttt{driver\_strategy()} routine with a \texttt{buf} structure *\texttt{bp}.

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

\texttt{physio()} continues to call \texttt{driver\_strategy()} for each \texttt{mincnt} size chunk, updating the \texttt{uio} structure each time until the transfer is done or an error is returned.\texttt{physio()} then unlocks the user's data area, saves the residual count (from \texttt{bp->b\_resid}) in the \texttt{uio} structure, interprets errors returned in \texttt{b\_error}, if any, and returns to the \texttt{driver\_read()} or
Writing a Driver
Step 6: Writing Entry Point Routines

driver_write() routine that called it.

Sample driver_read Routine Using physio()

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

Using uiomove()

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

The uiomove() call has the following declaration:

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

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

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

• Set flag to UIO_READ to have uiomove() copy n bytes, starting at address cp, into user space starting at address uiop.

  Set flag to UIO_WRITE to have uiomove() copy n bytes from uiop into kernel space starting at cp.

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

• geteblk() allocates a buffer from the file system’s buffer cache and returns it to the driver; the file system temporarily loses access to this
buffer. The buf structure and its associated buffer belong exclusively to the routine that called it.

- When the driver completes the request, it should release the buf structure and the buffer it obtained from geteblk() by using the kernel's brelse() routine.
- When a device driver gets a buffer using geteblk(), it is borrowing a buffer that would otherwise be used by the file system to cache data. This means a device driver that allocates buffers indiscriminately using geteblk() can affect the system's performance.

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

Writing a driver_ioctl() Routine

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

Although the user specifies a file descriptor, a command, and an argument to ioctl(), the kernel translates those arguments into the four parameters shown in DECLARATION.

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

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

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

You can use ioctl() to:

- Modify a driver's behavior.
Writing a Driver

Step 6: Writing Entry Point Routines

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

Section 7 of 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);
```

The parameters are:

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

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

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

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

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

Use one of the following statements to define each command word:
#define command _IO('t', n)
#define command _IOR('t', n, object)
#define command _IOW('t', n, object)
#define command _IOWR('t', n, object)

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

_IOR
Read data from the driver. That is, the driver writes into the kernel buffer pointed to by arg. Before returning to the user, the system copies the kernel buffer to the user specified buffer.

_IOW
Write data to the driver. That is, the driver reads from the kernel buffer pointed to by arg. Before calling the driver, the system copies the user specified buffer to the kernel buffer.

_IOWR
Both _IOR and _IOW.

_IO
Pass a pointer to the value arg. The value passed to the system call ioctl() is copied by the kernel and a pointer to that copy is passed to the driver. The pointer is of type ulong_t *.

The parameters are defined as follows:

command
The identifier you assign to your command.

t
An arbitrary character of your choice, used to associate the ioctl() call with your driver. Use the command grep 'define.*_IO' /usr/include/sys/* | more to ensure that you are not choosing a value (after macro expansion) that can conflict with another driver.

n
A number (0 to 127) that identifies a driver-specific command for the driver.

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

driver_ioctl() is expected to return the following values:

0  Successful completion.

<0  Error. See ioctl(2), ioctl(5) and errno(2) for a list of
Writing a Driver

Step 6: Writing Entry Point Routines

standard error values. ioctl() returns the error value to the user process in errno.

LP64 Considerations

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

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

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

```c
/* Public Header File */
#define SOME_IOCTL _IOR('X', 1, long)
/* Private Header File */
/* IOCTL for 32 Bit Applications */
#ifdef __LP64__
#define _SOME_IOCTL_32 _IOR('X', 1, int)
/* from 32-bit app */
#endif __LP64__
/* Driver ioctl code snippet */
switch (cmd) {
    case SOME_IOCTL:
        <do SOME_IOCTL processing>
    #ifdef __LP64__
        case _SOME_IOCTL_32:
            <do _SOME_IOCTL_32 processing>
    #endif /* __LP64__ */
}
```

Consider an ioctl cmd that 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__
#defineOPY_IOCTL_32 _IOWR('X', 2, struct buf_copy_32)
```
struct buf_copy_32 {
    ptr32_t buf_ptr32;
    int32_t buf_size;
}
#ifdef /* __LP64__ */
/* Driver ioctl code snippet */
switch (cmd) {
    case COPY_IOCTL:
<do COPY_IOCTL processing>
    break;
/* On 64 bit kernels these ioctls will process ioctls made by
32 bit applications */
#endif __LP64__
    case COPY_IOCTL_32:
<do COPY_IOCTL_32 processing>
    break;
#ifdef /* __LP64__ */
}
To copy the 32-bit application buffer, the driver calls copyin(). For example,
copyin((caddr_t)(struct _buf_copy_32 *)data->buf_ptr32),mybuf,
    sizeof(mybuf));
The copyin() and copyout() functions in the 64-bit kernel check to
determine whether the calling thread is a 32-bit application and, if so, do
the necessary 32-to-64-bit address conversion.
If there is a need with _IO to determine whether the calling thread is a
32-bit or 64-bit application, define two cmd values: one for ILP32 and
another for LP64. The following example demonstrates this.
/* Public Header File */
#endif __LP64__
#define LONG_IOCTL _IO('A',1)
#else
#define LONG_IOCTL _IO('a',1)
#endif /* __LP64__ */
/* Private Header File */
#endif __LP64__
#define _LONG_IOCTL_32('a',1)
/* long data from 32 bit app */
#endif /* __LP64__ */
Writing a Driver

Step 6: Writing Entry Point Routines

```c
/* Driver ioctl code snippet */
switch (cmd) {
    case LONG_IOCTL:
        <do LONG_IOCTL processing>
        break;
#ifdef __LP64__
    case _LONG_IOCTL_32:
        <do _LONG_IOCTL_32 processing for 32-bit app */
        break;
#endif /* __LP64__ */
```

Example

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

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

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

struct mydevice_ioctl_arg {
    char reg_value;
    caddr_t location;
};
#define CLEAR 0
#define SET 1
#define CARD_RESET _IO ( 'X', 0 );
#define CARD_STATUS _IOR ( 'X', 1, struct mydevice_ioctl_arg);
#define CARD_CONTROL _IOW ( 'X', 2, struct mydevice_ioctl_arg);
#define CARD_BUFADR _IOWR ( 'X', 3, struct mydevice_ioctl_arg);
```

This example defines four commands that the driver performs:

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

**CARD_CONTROL** Sets or clears the bits in the device's control register. In the `mydevice_ioctl_arg` structure, the `reg_value` field specifies the bits to be affected, and field
Writing a Driver

Step 6: Writing Entry Point Routines

location contains SET or CLEAR to indicate the action the driver is to take.

CARD_RESET Resets the device to its default state.
CARD_STATUS Returns the contents of the device’s status register to the user in the reg_value field of the mydevice_ioctl_arg structure.

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

#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 = MEMDEFAULT;
if (ioctl(fd, CARD_CONTROL, &ioctl_arg) < 0)
    printf("ioctl call failed, error number = %d\n", errno);

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

ifdef__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

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

/* header files this code segment needs */

Chapter 5 137
Writing a Driver

Step 6: Writing Entry Point Routines

```c
#include <sys/errno.h>
#include <sys/type.h>
#include <sys/mydevice.h>

struct my_device_registers *dev_rp;
mydevice_ioctl( dev_t dev,
    int cmd,
    struct mydevice_ioctl_arg *arg,
    int flag )
{

    ifdef__LP64__
        struct mydevice_ioctl_arg32 *arg32 =
            (struct mydevice_ioctl_arg32 *) arg;
    #endif

    switch(cmd)
    {
    case CARD_RESET:
        dev_rp->reset = 0;
        return(0);
    case CARD_STATUS:
        arg->reg_value = dev_rp->status;
        return(0);
    case CARD_CONTROL:
        switch(arg->location)
        {
        case SET:
            dev_rp->control |=arg->reg_value;
            return(0);
        case CLEAR:
            dev_rp->control &= ~arg->reg_value;
        default:
            return(EINVAL);
        } /* switch */
    case CARD_BUFADR:
        arg->location = set_buf_addr(arg->location);
        return(0);
    #ifdef__LP64__
        case CARD_STATUS_32:
            arg32->reg_value = dev_rp->status;
    #endif
    }
}
```
Writing a Driver

Step 6: Writing Entry Point Routines

```c
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;
            return(0);
        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_minphys()` Routine

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

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

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

void driver_minphys(struct buf *bp);

bp Pointer to a buf structure.
```

`minphys()` limits a single transfer to the size in bytes that your driver requires.
Writing a Driver

Step 6: Writing Entry Point Routines

driver_minphys() is passed as the mincnt parameter to physio(). In that case, physio() calls driver_minphys() and physio() keeps track of partial transfers that may occur due to the request size limit imposed by driver_minphys().
Example

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

static int
mydriver_write(dev_t dev, struct uio * uiop)
{
    return physio(mydriver_strategy, NULL, dev, B_WRITE,
                  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 is used to test I/O completion on a device. This routine is provided by the driver writer. It can have any unique name. You pass the name to WSIO Services by specifying it in the d_select field of the drv_ops structure. Commonly, driver is replaced by your driver’s name.

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

Performing select() on device files can have different interpretations depending on the device.

Use select() to poll a device for status.

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

You do this by specifying the global function seltrue in the d_select field of the drv_ops structure.

The driver_select() routine has no access to the readfds, writefds,
Writing a Driver

Step 6: Writing Entry Point Routines

and exceptfds values that the user passed to the select() system call. (See select(2) in the HP-UX Reference.) The driver_select() routine is passed only the device number and flag.

The driver must check for collisions in which two or more processes try to use select() on the same file for the same condition. To do this, the driver saves the pointer to the calling process’s user thread entry, and uses that pointer as an argument for the selwakeup() kernel call to revive the sleeping process when the condition becomes true.

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

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

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

*dev* The device number.

*flag* The type of readiness to test, according to the following values:

- FREAD Read
- FWRITE Write
- 0 Exception conditions

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

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

- 0 False. The device or driver is not ready for read or write or no exception condition was found. select() puts the calling process to sleep until the condition becomes true. The driver must inform the system when this condition becomes true.

The select() system call handles collisions if a true value is passed in the selwakeup() call’s second parameter. A true return (nonzero)
indicates the select succeeded. When false is returned the `select(2)`
system call sleeps on the global variable `selwait`.

```c
#include <sys/types.h>
#include <sys/param.h> /* for user.h */
#include <sys/user.h>   /* for u def */
#include <sys/kthread_iface.h>
/* kernel development fileset */
#include <sys/fileh>    /* 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 rw) {
    struct kthread t;
    spinlock(my_lock);
    switch(rw) {
    case FREAD:
        if (available data) {
            spinunlock(my_lock);
            return 1;
        } else if ((t=my_sel_struct->read_waiter) &&
            (kt_wchan(t) == (caddr_t)&selwait))
            mysel_struct->state |= READ_COLLISION;
        else
            mysel_struct->read_waiter = u.u_kthreadp;
        break;
    case FWRITE:
        if (ready for more data) {
            spinunlock(my_lock);
            return 1;
        } else if ((t=my_sel_struct->write_waiter) &&
            (kt_wchan(t) == (caddr_t)&selwait))
            mysel_struct->state |= WRITE_COLLISION;
        else
            mysel_struct->write_waiter = u.u_kthreadp;
        break;
    }
}
```
Writing a Driver

Step 6: Writing Entry Point Routines

```c
else
    mysel_struct->write_waiter = u.u_kthreadp;
    break;
} /* end switch */
spinunlock(my_lock);
return 0;
}
```

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

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

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

```c
/*
 * wake up any readers when driver has more input
 * available
 */
static void
```
Writing a driver_strategy() Routine for a Block Device

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

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

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

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

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

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

- Initializing data structures, such as DMA `buf` headers.
- Adding the I/O request to a queue, if necessary.
- Setting a flag that indicates I/O is in progress.
- Returning 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 may be executing on the interrupt stack. The process that invokes `driver_strategy()` determines whether to wait for the I/O to be completed.

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

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

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

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

**Using `driver_strategy()` for Write**

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

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

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

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

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

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

**Using `driver_strategy()` for Read**

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

For read requests, the `driver_strategy()` routine schedules the data to be copied from the device to the kernel buffer. The `driver_strategy()` routine should then return to the routine that invoked it. For read requests on block device files, the kernel always waits for the I/O to complete before returning to the user. The `read()` system call invokes `biowait()`, and waits for the I/O to complete.
When the I/O completes, the lower half of the driver should set `bp->b_resid` to the amount of data remaining to be transferred, set `B_ERROR` and set `bp->b_error` to an `errno` value if an error occurred, and call `biodone()`. The `read()` system call copies the data in the kernel buffer into the user's data area. The requested data is now available to the user process. The kernel releases the `buf` structure by clearing the `B_BUSY` flag, and calls `wakeup()` to wake up any processes sleeping on the buffer.

**Examples**

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

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

struct skelregs *skel; /* board registers */
struct buf *skelbuf; /* io buffer */
char r_int_enable_reg; /* software reg copy */

static void
skel_strategy(struct buf * bp)
{
    extern lock_t *my_lock;
    register caddr_t addr;
    register short cnt;
    struct isc_table_type *isc;
    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;
    }
    else
    
        skel_start(bp);
    }
    spinunlock( my_lock );
}

skel_start( bp )
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);
Writing a Driver

Step 7: Writing Other Driver Routines

This section describes primary driver routines that are not defined entry point routines; that is, they are not called through entry points defined in fields of your `drv_ops_t` structure. Interface drivers typically have `driver_attach()` and `driver_isr()` routines. They also may have `driver_if_init()` and `driver_probe()` routines. When an interface is shared by multiple device drivers, some method of linkage is required between the two types of drivers. Typically this is done using some type of I/O “switch” structure, as described in “The I/O Switch Tables” on page 47, Chapter 3. Device drivers also may have `driver_dev_init()` routines. Some of this material has been covered in previous sections, so this section will cover the `driver_isr()` and `driver_psize()` routines. Interface management and device queue management are also briefly discussed.

Writing a `driver_isr()` Routine

The `driver_isr()` function is used to handle device interrupts in interrupt context.

The `driver_isr()` routine is provided by the driver writer. It can have any unique name. You pass the name to WSIO Services by specifying it as a parameter of the `isrlink()` function, executed in your `driver_attach()` or `driver_if_init()` routine. Commonly, `driver` is replaced by your driver’s name.

The WSIO service `isrlink()` has the following interface:

```c
int isrlink(struct isc_table_type *isc, int (*isr)(),
            int irq_line, long arg1, long arg2)
```

`isc` is a pointer to the `isc_table` entry that represents the device, and `isr` is a pointer to the driver’s interrupt service routine (ISR). The third parameter `irq_line` is the interrupt request line assigned to the device. Most drivers will pass in `WSIO_IRQ_LINE_AUTO` to indicate the platform code must determine the appropriate `irq_line` for the device. If the driver knows that its device is to be bound to a particular `irq_line`, it can pass that `irq_line` value. The last two arguments, `arg1` and `arg2`, are used to specify arguments to pass to the driver’s ISR. By convention, drivers usually pass in `isc` as the first argument.
In an interface driver or a monolithic driver, the ISR processes interrupts from an interface card. The ISR performs the following tasks:

- Stops the interface card from interrupting.
- Determines a reason for the interrupt (if appropriate).
- Take appropriate action, such as cleanup or retry.
- Calls \texttt{wakeup()} or \texttt{biodone()}, or initiates the next step in processing an I/O request.
- Devices may share interrupt resources. The ISR associated with each device's driver may be called for interrupts not originating from its device. The ISR should be able to handle this and return 0 to the caller. Otherwise, the ISR returns 1, indicating that the interrupt has been serviced.

An ISR executes in an interrupt context, not a kernel thread context. Therefore, an ISR must never call \texttt{sleep()} or a function that may block. An ISR has the following interface:

\begin{verbatim}
int driver_isr (long arg1 int arg2);
\end{verbatim}

- \texttt{arg1} Driver defined parameter passed in the call to \texttt{isrlink()} as \texttt{arg1}.
- \texttt{arg2} Driver defined parameter passed in the call to \texttt{isrlink()} as \texttt{arg2}.

\texttt{driver_isr()} is expected to return the following values:

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

**Examples**

The following are examples of a \texttt{driver_if_init} and \texttt{driver_isr} functions. The \texttt{driver_if_init} function calls \texttt{isrlink()} to register its \texttt{driver_isr} function.

\begin{verbatim}
static int CentIf_init (struct isc_table_type *isc)
{
    <<do any additional driver initialization >>>
    ......................
    .................
    (void) isrlink (isc, CentIf_isr, WSIO_IRQ_LINE_AUTO,
                   (long)isc, 0L);
\end{verbatim}
Writing a Driver

Step 7: Writing Other Driver Routines

return (WSIO_OK);
}

The code that follows is an 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;
        sw_trigger(&intloc, RealIntrHndlr, isc, 3, 0);
        return 1; /* interrupt has been serviced */
    } else {
        return 0; /* interrupt not from my device */
    }
}

Writing a driver_psize() Routine

The driver_psize() function is used to get the swap partition size of a device

int driver_psize (dev_t dev);

dev Contains encoded major and minor numbers;

The driver_psize() WSIO function is provided by the driver writer. It can have any unique name. You pass the name to WSIO Services by specifying it in the d_psize field of the drv_ops structure. Commonly, driver is replaced by your driver’s name.

The driver_psize() WSIO function should return the size of the swap partition on a block swapping device, It is called by the kernel. Consider writing this routine only if your device is used for swapping.

driver_psize() returns the following values:

>0 Successful completion. The value is the swap partition size.

-1 Error.
This SCSI example assumes that `driver_psize()` is never called when the device is closed, hence there is no need to do an explicit open and close in the routine. Note the use of the SCSI Services `m_scsi_lun()` function.

```c
static int
mydriver_psize(dev_t dev)
{
    struct scsi_lun *lp = m_scsi_lun(dev);
    struct mydriver_lun *llp = lp->dd_lun;
    int nblks, rshift;

    nblks = llp->nblks;
    rshift = llp->devb_lshift;

    return (rshift > 0 ? nblks >> rshift : nblks << -rshift);
}
```

### Interrupt-Driven Routines for Device Drivers

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

If you are writing an interrupt-driven driver that also requires management of device queues and select codes, you will also need to deal with problems related to the environment (registers, variable states, and so on).

#### Management of Device Queues

If more than one instance of your device driver can run simultaneously, you need to provide your driver with a device queue. This will prevent your driver from sending requests to the device faster than the device can complete them. A device queue allows requests to be queued, to await their turn for the device, and to have their I/O completed.

Routines that manage device queues use the `buf` data structure. You need to be familiar with this data structure, its contents, and its intended usage.

This section describes only the fields in the structure that are explicitly used in managing device queues.

**The `buf` Structure** The `buf` structure is defined in `<sys/buf.h>`. See `buf(KER4)` in the HP-UX Driver Development Reference. All I/O requests end up as a `buf` structure. The `buf` structure contains information on the
current and potential owners for devices, and contains all information necessary for a driver to complete the I/O request.

The following buf fields are of particular importance in the context of device queue management (refer to the code for a complete list):

- **av_forw** Points to the next buf structure in the queue. Its value is NULL if the current buf structure is the last one in the queue.
- **b_error, b_flags** Used to determine if an error occurred.

### Interface Management

Whenever more than one device can reside on an interface card, the interface must be managed. For example, an HP-IB bus requires such management, but a built-in parallel interface does not. Because all the devices on an interface card cannot do I/O at the same time, each interface card must have an owner that has the sole ability to do I/O on devices connected to the card.

Fields in the buf structure describe the owner of an interface. Because there can only be one owner of an interface card at a time, interface drivers usually use a queue to hold buf structures waiting for their chance to own the interface card. There is one queue for each interface.

### Data Structures for Managing Interface Cards

Routines that manage interface cards use two data structures, buf and isc_table_type. You need to be familiar with these data structures, their contents, and their intended usage.

The following sections describe the fields in these structures that are used to manage interface cards.

**The isc_table_type Structure** The isc_table_type structure is defined in <sys/io.h>. See isc_table_type(KER4) in the HP-UX Driver Development Reference. The isc_table_type structure contains all pertinent information about the interface driver space. This structure must be initialized by the interface driver for the interface at the specified select code.
6 Installing Your Driver
Installing Your Driver

This chapter describes how to build your driver into the kernel. For information about demand loadable module drivers see Chapter 10, “Developing Dynamically Loadable Kernel Modules,” For other drivers, do the following:

1. Compile your driver; if you are building a driver for both 32-bit and 64-bit kernels, you will need to build separate libraries for each kernel flavor.
2. Add your driver's name to the system file.
3. Create a master file for your driver in the /usr/conf/master.d directory. Do not update files shipped with the system.
4. Run the config command to build your driver into the kernel.
5. Reboot the system.
6. Use the ioscan command to associate your driver with your hardware.
7. Use the mknod command to create a device-special file for your driver.
8. Use the ioscan command to test the configuration.

The sections below give detailed descriptions of the steps above.
In these procedures, we assume that your driver's name is mydriver.
Step 1: Compile Your Driver

Before you can add your driver to the HP-UX kernel, you must compile your driver to generate a new object file. To do this, change to the directory that contains your source files. We suggest that you use a subdirectory of /usr/conf, such as /usr/conf/mydriver.

Next, compile your driver using the HP ANSI C compiler. The HP product numbers for the compiler are:

- B3901BA - HP 9000 servers
- B3899BA - HP 9000 workstations
- B6237AA - Documentation set

NOTE

You may use the bundled compiler /usr/ccs/bin/cc to build the kernel from /stand/build/config.mk. You should use the HP ANSI C compiler to build your driver into the kernel.

Compiling a Driver

The easiest method is to use the compiler options found in the makefile /stand/build/config.mk. Copy this file to Makefile under the driver subdirectory. An example of mydriver is:

```
cp /stand/build/config.mk /usr/conf/mydriver/Makefile
```

You have to make the following modifications to the Makefile to build your driver:

1. In the Makefile, change the target all to your driver object file. So for mydriver example, change the following line in the Makefile

   **from:**
   
   ```
   all: ${DIR}/${HPUX}
   ```

   **to:**
   
   ```
   all: mydriver.o
   ```

2. In the Makefile, change CONF with your driver name. So for mydriver example, change the Makefile line

   **from:**
   
   ```
   CONF=/stand/build/conf
   ```

   **to:**
   
   ```
   CONF=conf/mydriver
   ```
Installing Your Driver

Step 1: Compile Your Driver

3. You may probably need to change the C compiler location. Say, the ANSI C compiler is located at /opt/ansic/bin. For the mydriver example, change the Makefile line from: CC= $(CKRN)/cc to: CC= /opt/ansic/bin/cc

4. When the ANSI C compiler is used to build your driver, you also need to include additional flags -Ae to the compiler options list. Refer to the cc (1) man page for more details. For the mydriver example, change the Makefile line from: CFLAGS= -w ${COPTS} ${K_CCOPTS} to: CFLAGS= -Ae -w ${COPTS} ${K_CCOPTS}

5. If you are building a 32-bit driver, make sure that the destination architecture flag is correctly specified as +DA1.1. The Makefile should have the following line:

DAFLAG=+DA1.1

6. If you are building a 64-bit driver, be sure the destination architecture flag is correctly specified as +DA2.0W. The Makefile should have the following line:

DAFLAG=+DA2.0

Now you can build your driver by typing make at the command prompt. Alternatively, you may use the compiler options directly at the command line as shown below:

If you are using an ANSI C compiler and building a 32-bit driver:

```
/opt/ansic/bin/cc -I. -c -o mydriver.o -Ae -w \\
-U_hp9000s700 -D_HIGHC_ -D_STDC_EXT__ \\
-D_KERNEL_BUILD -D_XPG4_EXTENDED -D_HPUX_SOURCE \\
-D_UNSUPPORTED -D_hp9000s800 -D_KERNEL -DKERNEL \\
+ES1.Xindirect_calls -Wp,-H300000 +XixdU +Hx0 +R500 \\
-Wl,-a,archive +DA1.1 +DS2.0 +ESsfc +ESssf \\
mydriver.c
```

If you are using an ANSI C compiler and building a 64-bit driver:

```
/opt/ansic/bin/cc -I. -c -o mydriver.o -Ae -w \\
-U_hp9000s700 -D_HIGHC_ -D_STDC_EXT__ \\
-D_KERNEL_BUILD -D_XPG4_EXTENDED -D_HPUX_SOURCE \\
-D_UNSUPPORTED -D_hp9000s800 -D_KERNEL -DKERNEL \\
+ES1.Xindirect_calls -Wp,-H300000 +XixdU +Hx0 +R500 \\
-Wl,-a,archive +DA2.0 +DS2.0 +ESsfc +ESssf \\
```
mydriver.c

When the compilation succeeds, it produces a file named mydriver.o in the current directory.
Step 2: Create a Master File

Create a master file for your driver in the /usr/conf/master.d directory.

The master files in /usr/conf/master.d provide configuration information about drivers. The HP master files group drivers along functionality lines; there is a master file for core drivers and subsystems, a master file for basic networking, and so on. The format of these files is defined in master(4) in the HP-UX Reference. The /usr/sbin/config command generates a comprehensive master file from the individual master files in /usr/conf/master.d.

Creating a Master File for Your Driver

Create a new master file in /usr/conf/master.d that contains the information specific to your driver.

The file can have any name, but we recommend that you name your master file in a way that identifies it with your driver, so customers will recognize the connection. In our example, we call it mydriver. The master file's name must be unique within /usr/conf/master.d. Perhaps you can identify it with your company name to avoid conflicts with other master file names from other suppliers.

CAUTION

Do not add your driver's information to one of the existing master files supplied by Hewlett-Packard in /usr/conf/master.d. These files will be overwritten when you or your customers update the system with the next HP-UX release.

The name of the master files in /usr/conf/master.d must conform to the following conventions:

- File names must not contain the characters period (.), tilde (~), or pound sign (#).
- File names must not contain the word core.
- File names must not contain the word RMTBRANCH.
- Files should have names no longer than 14 characters to allow your
Installing Your Driver

Step 2: Create a Master File

driver to be used on short file name systems.

Editing the Master File

Make the following entries in your master file:

• Each driver needs an entry in the $DRIVER_INSTALL table in a master file.

Here is an example of a portion of a $DRIVER_INSTALL table:

$DRIVER_INSTALL

***************

* Driver   Block major  Char major  Required for minimal system
*
asio0      -1           1  
cpd        -1           -1          1
kepd       -1           227  1
dev_config -1           69  1
klog       -1           189  1
cn         -1           0  1
mm          -1           3  1
pty0        -1           16  1
...

$$$

The following paragraphs describe the fields in the $DRIVER_INSTALL table and what to put in them for your driver. The fields are separated by spaces.

— The first field, Driver, contains the driver's name, for example, mydriver.

— The second and third fields, Block major and Char major, specify the driver's block and character major numbers. Specify -1 in both fields so that the kernel will assign your driver a major number dynamically. (These fields are informational; the values in the drv_info_t structure take precedence.)

— The fourth field specifies whether the driver is needed to run a minimal system. Enter 1 if true; otherwise, enter 0 or leave it blank.

• If your driver has dependencies on other drivers, list these other drivers in the $DRIVER_DEPENDENCY table in the master file. The format is:
Installing Your Driver

Step 2: Create a Master File

driver_name otherdriver1 otherdriver2 ...

- Use the $DRIVER_LIBRARY table to specify libraries that contain functions your driver uses. This should be set only if the entries you list here are in /usr/conf/lib directory. Look at Step 4 for additional details on different options for building a kernel containing your driver. The format is:

  driver_name lib1 lib2 ...

Use $LIBRARY table to specify whether a library has to be always included in the kernel or not.

If you list your driver library under $DRIVER_LIBRARY, include your driver library entry under this. The format is:

  driver_library_name <0 | 1>

wherein, 0 - if the library is optional
1 - if the library should “always” be included

Your driver may require entries in other master file tables.

See master(4) in the HP-UX Reference for details on the format of master file tables. You may also find it useful to examine an existing master file, such as core-hpux in /usr/conf/master.d.

Example Master File

An example master file for mydriver is described in this section. The master file must be located under /usr/conf/master.d with a name of mydriver. Refer to the beginning of this section for file name restrictions.

* * @(#) $Revision: 1.1.106.2 $ *
* The following devices are those that can be specified in the *
* system description file. The name specified must agree with *
* the name shown, or with an alias. *
* *
* name handle type mask block char *
* *
$DEVICE
** Reserved for Third Party and User Drivers.
*
* *** --- --- ---  --- 38
* *** --- --- ---  --- 39
* *** --- --- ---  --- 40
* *** --- --- ---  --- 41
* *** --- --- ---  --- 42
Installing Your Driver

Step 2: Create a Master File

* The following entries form the cdio table.
* field 1: cdio name
* field 2: required, optional
*
***

$CDIO

* The following entries form the alias table.
* field 1: product # field 2: driver name
*
***

$ALIAS

* The following entries form the tunable parameter table. Any
* of these values can be overwritten in the dfile. However,
* before changing any value, know the ramifications of your
* change.
*
***

$TUNABLE

* Driver dependency table, if one driver is present, what
* other drivers should also be in the dfile.
*

---

$DRIVER_INSTALL

******************************************************
* Driver install table
*
* This table contains the name of drivers which have converged
* I/O header structures and install entry points. Drivers in
* this table should not be defined in the driver table above.
*
* Note : as of 10.30, lan0 and lan1 are obsolete
******************************************************
* Driver    Block major   Char major
mydriver     -1           -1

***

$DRIVER_DEPENDENCY

* Driver dependency table, if one driver is present, what
* other drivers should also be in the dfile.
*
Installing Your Driver

Step 2: Create a Master File

* Driver name <dependency> <dependency> ..... $$$

$DRIVER_LIBRARY
*
* The driver/library table. This table defines which libraries
* a given driver depends on. If the driver is included in the
* dfie, then the libraries that driver depends on will be
* included on the ld(1) command line. Only optional libraries
* need to be specified in this table, (but required ones can be
* included, as well).
*
* Driver handle <libraries>
*
* subsystems first
$$$ *

$LIBRARY
*
* The library table. Each element in the library table
* describes one unique library. The flag member is a boolean
* value, it is initialized to 1 if the library should *always*
* be included on the ld(1) command line, or 0 if the library is
* optional (i.e. it is only included when one or more drivers
* require it). The order of the library table determines the
* order of the libraries on the ld(1) command line, (i.e.
* defines an implicit load order). New libraries must be added
* to this table.
*
* Note: libhp-ux.a must be the last entry, do not place *
* anything after it.
*
* Library <required>
*
$$$ *

$SUBSYSTEMS_DEFINE
*
* Subsystem #define table. For any subsystem that is found in
* the dfie and appears below, a #define is generated in the
* conf.c file. This is used to include any space defined in
* space.h.
$$$ *

$DRIVER_PRODUCT
*
* Driver product table. The following generates a table of
* supported products for a given interface card. The input is a
* driver handle followed by a list of product/vendor id’s. What
* is produced is a table of these product/vendor id’s to be used by the interface driver for further qualification during booting and the setting of root device location.
*
$$$

Chapter 6

165
Step 3: Modify the System File

You must make an entry for your driver in the system file. /bin/config uses the system file to build the link/install table. The contents of that table are traversed at configuration time to indicate which drivers will be preinstalled into the kernel.

The system file (the default is /stand/system) is the description file for a particular system configuration. See config(1M) to get more information about the format of the system file.

1. Change directory to the build environment (/stand/build). Then execute the system-preparation script, system_prep, which extracts the system file from the current kernel:

   ```
   cd /stand/build
   /usr/lbin/sysadm/system_prep -s system
   ```

   The system_prep script writes a system file in your current directory. (It creates /stand/build/system.)

   Alternatively, you can just copy the current system file into your build directory.

   ```
   cp /stand/system ./system
   ```

2. Edit the /stand/build/system file to add the name of your driver mydriver to the first section Drivers/Subsystems of the file on a separate line. See config(1M) for details on the format of the system file.
Step 4: Build a Kernel Containing Your Driver

You have created a master file for your driver and have edited the system file to include your driver. Now, follow these steps to build your kernel and prepare it for testing. Your current directory is still /stand/build.

1. Use the following command to generate the files used to create a new kernel containing your driver.

   `/usr/sbin/config -s system`

   This command has two output files, both placed in /stand/build, `config.mk` and `conf.c` (i.e., both are placed in your current directory).

   • The `config.mk` makefile generates a kernel. The two macro definitions, `OFILES` and `XOPBJS=`, are in `config.mk`.
   • The `conf.c` file links the drivers you specified in your system file with the kernel.

   (See `config(1M)` in the HP-UX Reference.)

2. Include your object file in the new kernel. There are four methods. The first and second are suitable for testing; the third and fourth are preferable for installing the production version.

   • Alternative 1: Use the `XOBJS` variable.
     Build your new kernel with the following command:

     `make -f config.mk XOBJS=/usr/conf/mydriver/mydriver.o`
     Wait a moment while `config.mk` links your driver with the rest of HP-UX, and builds a file named `vmunix` in your current directory.

   • Alternative 2: Use the `OFILES` variable.
     Modify `config.mk` so that the definition of `OFILES` includes your object file. Look in `config.mk` for lines that look something like:

     `$(ROOT)/libcdfs.a $(ROOT)/libhp-ux.a`  
     `OFILES=/usr/conf/mydriver/mydriver.o`

     Add a reference to your driver by adding it to the `OFILES` variable as follows:

     `$(ROOT)/libcdfs.a $(ROOT)/libhp-ux.a`
Installing Your Driver

Step 4: Build a Kernel Containing Your Driver

OFILES = mydriver.o

Build your new kernel with the following command:

make -f config.mk

Wait a moment while config.mk links your driver with the rest of HP-UX and builds a file named vmunix_test in your current directory.

• Alternative 3: Place Your Object File in a Library

Add your object file to the /usr/conf/lib/libusrdrv.a library. If the library exists, the LIBUSRDRV variable automatically links your kernel with the members of the library. You must create the library, if necessary, and add your driver to it.

Use this command:

ar -r /usr/conf/lib/libusrdrv.a mydriver.o

Build your new kernel with the following command:

make -f config.mk

Wait a moment while config.mk links your driver with the rest of HP-UX and builds a file named vmunix in your current directory.

• Alternative 4: Place Your Driver Library Under /usr/conf/lib

For example, in the case of mydriver:

Wait a moment while config.mk links your driver with the rest of HP-UX and builds a file named vmunix in your current directory.

— cd to /usr/conf/mydriver/
— Build the driver library with $ar -r libmydriver.a mydriver.o
— Copy the driver library to /usr/conf/lib/
  cp libmydriver.a /usr/conf/lib
— cd back to /stand/build.

You have to modify the driver master file to include the driver library. From the example master file shown, you will have to add your driver library entry under $DRIVER_LIBRARY_ and $LIBRARY tables. Since mydriver is optional in the kernel, we use a value of 0 for libmydriver.a under the #LIBRARY section. The modified parts of the master file look like this:
$DRIVER_LIBRARY
* The driver/library table. This table defines which
* libraries a given driver depends on. If the driver is
* included in the dfilie, the libraries that driver
* depends on will be included on the ld(1) command
* line. Only optional libraries *need* to be specified in
* this table, (but required ones can be included, as
* well).
*
* Driver handle <libraries>
*
* subsystems first
mydriver libmydriver.a

$LIBRARY
* The library table. Each element in the library table
* describes one unique library. The flag member is a
* boolean value, it is initialized to 1 if the library
* should *always* be included on the ld(1) command line,
* or 0 if the library is optional (i.e. it is only
* included when one or more drivers require it). The
* order of the library table determines the order of the
* libraries on the ld(1) command line, (i.e. defines an
* implicit load order). New libraries must be added to
* this table.
*
* Note: libhp-ux.a must be the last entry, do not place
* anything after it.
*
* Library <required>
*
libmydriver.a 0

After the master file is changed, use the following commands to
generate the files used to create a new kernel containing your
driver.

/usr/sbin/config -s system

This command has two output files, config.mk and conf.c, both
placed in /stand/build directory (which is your current directory).
Installing Your Driver

Step 4: Build a Kernel Containing Your Driver

Run the following command to build the kernel containing your driver.

```
make -f config.mk
```

3. Save the current system file and kernel by renaming them. If anything goes wrong, you still have a bootable kernel. For example,

```
mv /stand/system /stand/system.prev
mv /stand/vmunix /stand/vmunix.prev
```

4. Move the new system file and new kernel into place, ready to be used when you reboot the system.

```
mv /stand/build/system /stand/system
mv /stand/build/vmunix_test /stand/vmunix
```

5. Reboot your system with the new kernel. Enter the following command:

```
exec reboot
```

For more information about building and installing your kernel, see the HP-UX System Administration Tasks.
Step 5: Create a Device-Special File

Once the system has rebooted successfully, you need to create a device-special file for your device in the /dev directory. To do this, you must determine the device's major and minor numbers and use the mknod command.

1. Use the lsdev command to identify the major number assigned to the device driver. lsdev lists all device drivers configured into the kernel, and their block- and character-major numbers.

   By adapting an example from the lsdev(1M) manpage, you can extract the numbers from the display:

   ```
   lsdev -h -d mydriver | awk '{print $1}'
   character major number
   lsdev -h -d mydriver | awk '{print $2}'
   block major number
   ```

2. Invoke the /usr/sbin/ioscan command with the -f or -k option, and note the hardware path for which ioscan reports an unknown class of device.

3. Construct a minor number for the device by using the bit assignments for the driver. See “Major and Minor Numbers” on page 39, Chapter 3, for more information about bit assignments and dev_t.

4. Use the mknod command to create the device-special files for the device. See mknod(2) in the HP-UX Reference. For information on file-naming conventions, see Configuring HP-UX for Peripherals.

   In the example below, mydriver was (dynamically) assigned the block- and character-major numbers 65 and 234, respectively. Its minor number, 0x026000, is constructed like that of instr0 (bits 8 through 15 encode 2 as the instance of the interface card, and bits 16 through 19 encode 6 as the device's address).

   ```
   /usr/sbin/mknod /dev/mydriver b 65 0x026000
   /usr/sbin/mknod /dev/mydriver c 234 0x026000
   ```

5. Verify the configuration by invoking ioscan with the -fun or -fkn options. If the device-special files are created properly, ioscan displays them beneath the configured device.

For information on packaging third-party drivers for distribution with
Installing Your Driver

Step 5: Create a Device-Special File

HP-UX, see Managing HP-UX Software with SD-UX.
Chapter 7

Creating Networking Device Drivers
Creating Networking Device Drivers

This chapter provides information about designing and writing PCI networking device drivers.

The major difference in the network device driver model from HP-UX 10.20 for third party developers is that third party developers are expected to implement their own DLPI layer and not depend on or use the HP DLPI. Also, HP supports “pure” STREAMS model drivers; BSD style drivers are not supported anymore.

The first section contains an overview of the structure of networking drivers. You may use the steps outlined in this section as a general guide to HP-UX driver design. The second section introduces the HP-UX networking interface architecture for the PCI platform. Be sure to review this section before beginning development of your networking driver. The remaining sections of this chapter contain network device driver topics and sample code for each part. Refer to this information, as well as the enet sample driver provided in your driver development kit, to create your PCI networking device driver.

The information in this chapter is intended for developers with extensive experience in designing and writing networking device drivers for non-HP UNIX target systems.

Basic STREAMS module/driver development and general networking concepts and RFCs are not included in this documentation.
Overview of Networking Driver Structure

The flowchart in Figure 7-1, “Steps to Develop a Networking Driver,” shows a sequence of high level steps that you can use as a development guide when writing networking drivers on HP-UX systems. The sequence of information in the flowchart also closely maps to the sequence in which the information is presented in this chapter. Refer to the specific module descriptions below for pointers to the subject areas shown in the flowchart.

Prior to beginning the development of your driver, HP recommends that you carefully study the material on HP-UX Networking Interface Architecture in the following section.

STREAMS model device drivers

IHVs and ISVs are expected to write their own DLPI layer along with the network interface device driver in HP-UX 11.0. The DLPI layer is implemented as a STREAMS driver and the PCI network interface driver as a WSIO driver. The interface between the DLPI layer and the network interface device driver is not defined by HP. Third parties are free to define it according to their design needs. This document provides a framework which includes a STREAMS DLPI driver, dlpi_enet, a PCI network interface driver, enet, and the interface between the two, as part of the development kit.

NOTE

The names DLPI STREAMS driver, native DLPI, DLPI driver and DLPI layer are used interchangeably. Also, network interface driver, WSIO network driver, and PCI network interface driver mean the same.

Development Steps

a. HP-UX Network Interface Architecture  Overview of the STREAMS environment in HP-UX.

b. Data Structures  Describes the data structures in the networking interface layer: hw_ift_t, hw_dlpi_t and device driver data structure framework. Refer to “Data Structures
Creating Networking Device Drivers

Overview of Networking Driver Structure

and Interfaces” on page 184 for detailed information about these data structures.

c. Protection and Synchronization   Describes the OSF/Encore spinlock protection model. Refer to “Protection and Synchronization for Driver and Transport” on page 189 for more detailed information about supporting the spinlock scheme in HP-UX.
Figure 7-1 Steps to Develop a Networking Driver

d. Driver Initialization  Describes the install and initialization routines for WSIO networking drivers and STREAMS drivers.
Creating Networking Device Drivers

Overview of Networking Driver Structure

The attach routine is discussed for the WSIO driver. Refer to “Initializing Networking Device Drivers” on page 191 for detailed information about these routines.

e. Protocol Configuration, Binding, and Demultiplexing Describes configuration of the INET stack for the STREAMS model drivers. Also, the routines for the driver to bind and demultiplex upper-layer protocols to a device are explained. Refer to “Protocol Binding and Demultiplexing” on page 201 for detailed information about these routines.

f. mblk and queue macros Lists macros commonly used by networking and STREAMS drivers. Refer to “mblk and queue macros” for detailed information about these routines.

g. DLPI sequence Describes the general operation of DLPI STREAMS and network device drivers.

h. STREAMS Driver Provides the overview of STREAMS DLPI driver and describes the major functions in the driver.

i. WSIO networking driver Provides the overview of a networking device driver and describes the major functions in the driver.

j. DLKM Support? If not, go to l.

k. DLKM Wrappers Describes how to convert the STREAMS and network driver to DLKM modules.

l. Auxiliary code? HP customers expect to have network management and tracing and logging support in their networking products. HP recommends adding these routines to your network driver.

m. Network Management Support? If not, go to o.

n. Network Management Code Describes the routines to support network management requests. Refer to “Network Management Support” on page 240 for more detailed information.

o. Logging and Tracing Support? If not, go to q.

p. Logging and Tracing Code Describes the routines to support Logging and Tracing. Refer to “Formatting Networking Trace/Log Messages” on page 271 for more detailed information.
Creating Networking Device Drivers

Overview of Networking Driver Structure

q. Auxiliary Files? HP customers expect to have automated configuration through SAM, and to be able to display link and encapsulation statistics and tracing and logging messages. If not, go to Driver Complete.

r. SAM Support? If not, go to s.

s. SAM Files Provides configuration files to support the HP menu-driven utility to configure a networking driver. Refer to “Configuring a Networking Driver Through SAM” on page 245 for more detailed information.

t. driveradmin Support? If not, go to u.

u. driveradmin Files Describes the shared library for the driveradmin command. This is used to display link statistics and perform various administrative tasks. Refer to “Network Monitoring Commands: driveradmin, driverlinkloop and lanscan” on page 246 for more detailed information.

v. lanscan Support? If not, go to w.

w. lanscan Files Describes the shared library for the lanscan command. This is used to display link encapsulation information. Refer to “Network Monitoring Commands: driveradmin, driverlinkloop and lanscan” on page 246 for more detailed information.

x. Logging and Tracing? If not, the driver is completed.

y. Logging and Tracing Files Describes the files needed to support logging and tracing requests. Refer to Appendix C, “How to Design a Networking Trace/Log Subformatter and a Sample Subformatter,” for more detailed information.

HP-UX Networking Interface Architecture

The information in this section describes the HP-UX networking interface architecture for the PCI bus. This interface supports OSI protocols, Internet protocols, and DLPI protocols on HP-UX platforms.

The HP-UX networking subsystem comprises three logical layers, as shown in Figure 7-2, “Three-layered HP-UX Interface to the PCI Bus,” and is briefly described in the following three subsections:
Creating Networking Device Drivers

Overview of Networking Driver Structure

“Data Link Interface Layer” on page 182

“Network Protocol Layer” on page 182

“Protocol Interface Layer, Device File, and STREAMS Head” on page 182
Figure 7-2  Three-layered HP-UX Interface to the PCI Bus

- Application Layer
  - TCP/IP Networking Commands & Applications
  - Driver Networking Commands & Utilities
- User Space
- Kernel Space
- Protocol Interface Layer
- Streams Head & Device File
- Network Protocol Layer
- STREAMS-based Protocol
- Data Link Provider
- Network Device Drivers
- Ethernet Card
- Token Ring Card
- X.25 LAPB Card
- FDDI Card
Data Link Interface Layer

Data Link layer has DLPI STREAMS drivers and the network interface drivers. A DLPI driver interacts with STREAMS modules in the system. A Network interface driver is responsible for manipulating its hardware devices (e.g., Ethernet cards) and for encapsulating/decapsulating link-level protocol (e.g., SNAP) headers that are required to deliver messages to a destination. The data link layer:

- Directly connects to the network interface hardware (network interface, physical layer).
- Consists of the hardware interfaces and their respective device drivers.
- Implements DLPI Version 2.0 to interact with STREAM/UX.

Network Protocol Layer

The network protocol layer, above the data-link interface layer, encompasses four protocol families:

- Internet: TCP/IP, UDP/IP
- OSI
- X.25
- ARP

Each network protocol family belongs to a domain and uses the address scheme in that domain. For example, the Internet (INET) family of protocols form the Internet domain.

The network protocols of other domains, such as the OSI stack, may be functionally equivalent to the Internet stack, but are generally not compatible with Internet domain protocols.

Protocol Interface Layer, Device File, and STREAMS Head

This interface layer directly supports applications. The main functions of this layer are to:

- Identify different applications on the same host (for example, a socket interface or a device file interface).
Creating Networking Device Drivers
Overview of Networking Driver Structure

• Provide services from transport layer protocols and above to applications.

The interface for this layer provides the following abstract objects that applications can create, open, connect, or use for sending or receiving data:

• Sockets
• Streams
• Device files

Streams Environment Layer

The kernel modules for the HP-UX transports (e.g. TCP/IP, UDP, OSI) are now STREAMS modules. Drivers that interface to the transport stacks must now work with this environment.

Driver writers should refer to the following documents for information concerning STREAMS modules and device drivers. Notable attention should be paid to the DLPI references. This document will only briefly discuss the STREAMS mechanisms and will concentrate on specific HP variants.

The following documents are recommended sources:

Hewlett-Packard Company:

• HP 9000 Networking DLPI Programmer’s Guide, HP Part No. 98194-90059

Other References:

• Data Link Provider Interface Specifications, Unix International

Streams Model Network Driver

As mentioned earlier, from HP-UX 11.0 onwards, third party network drivers are expected to have their own DLPI layer. DLPI STREAMS driver interface with STREAMS modules/drivers (IP and ARP modules) and other transport stacks. The interface between a native (third party) DLPI STREAMS driver and the PCI network interface driver is not controlled by HP. Third parties are free to design this interface.
Data Structures and Interfaces

The following data structures are used by the network interface layer:

- `hw_ift_t` (defined in `sio/lan_dlpikrn.h`)
- `hw_dlpi_t` (contained in `hw_ift_t`; defined in `sio/lan_dlpikrn.h`)

Each device driver may maintain its `hw_ift_t` and `hw_dlpi_t` structure as part of a larger structure, the driver control block, `enet_if_t` (shown in Figure 7-3, “Networking Driver Control Block and Structures.”). The driver control block provides information used in driving and controlling the interface hardware.

**hw_ift_t Structure Description and Initialization**

The `hw_ift_t` structure provides a consistent method for the network system utilities, `lanscan` (see `lanscan(1M)`), `driveradmin` and `driverlinkloop`, to display detailed information about all network devices. The `hw_ift_t` structure is described below.

```c
typedef struct hw_ift {
    hw_dlpi_t hp_dlpi;
    uint32_t mac_type;
    uint32_t llc_flags;
    uint32_t maj_num;
    uint32_t nm_id;
    uint32_t instance_num;
    uint32_t instance_num;
    uint32_t mtu;
    char *name;
    uint8_t hw_path[MAX_HDW_PATH_LEN];
    uint32_t hdw_state;
    uint32_t mac_addr_len;
    uint8_t mac_addr[MAX_MAC_ADDR_LEN];
    uint32_t features;
    uint8_t *arpmod_name;
    uint32_t ppa;
    uint32_t watch_timer;
    uint32_t reserved2;
    lock_t *hwift_lock;
    struct hw_ift *next;
} hw_ift_t;
```
The following fields must be properly initialized by the device driver during system initialization to support the HP-UX system utilities.

- **hp_dlpi**: It must be initialized to all zeros.
- **mac_type**: Device type. See “MAC Types and Protocol Types” on page 502, Appendix D, for network media types.
- **llc_flags**: Link Level Control (LLC) encapsulation method.

---

**Figure 7-3**  Networking Driver Control Block and Structures
### Creating Networking Device Drivers

#### Data Structures and Interfaces

Defined flag values and the encapsulation method are listed in “MAC Types and Protocol Types” on page 502, Appendix D.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mjr_num</td>
<td>Major number of the device file associated with the device. The major number could be statically assigned or, if it is set to -1, the major number is allocated dynamically during driver initialization.</td>
</tr>
<tr>
<td>nm_id</td>
<td>Network management ID. nm_id should be initialized via a call to the <code>get_nmid()</code> routine.</td>
</tr>
<tr>
<td>instance_num</td>
<td>Device instance number. This is the value returned by calling the <code>wsio_isc_to_instance()</code> routine.</td>
</tr>
<tr>
<td>mtu</td>
<td>Maximum transmission unit (number of bytes) for a particular type of link or encapsulation. See “MTU Values” on page 505, Appendix D, for a list of predefined MTU values.</td>
</tr>
<tr>
<td>name</td>
<td>Driver device name used for naming shared libraries for <code>lanscan</code> and <code>driveradmin</code>.</td>
</tr>
</tbody>
</table>

**NOTE** The driver names `lan` and `fddi` are reserved for HP devices.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hdw_path</td>
<td>Hardware path obtained by calling <code>io_node_to_hw_path()</code> followed by <code>io_hw_path_to_str()</code>.</td>
</tr>
<tr>
<td>hdw_state</td>
<td>Hardware state of the device. 0, if the device is OK. If the device is not available, hdw_state must be set to LAN_DEAD.</td>
</tr>
<tr>
<td>mac_addr_len</td>
<td>Number of bytes of mac_addr[] for MAC address.</td>
</tr>
<tr>
<td>mac_addr</td>
<td>MAC address of the device. For Ethernet/IEEE 802.3 and FDDI, the address is in canonical form. For IEEE 802.5, the address is in wire form.</td>
</tr>
<tr>
<td>features</td>
<td>Features supported by device. Two flags are supported:</td>
</tr>
<tr>
<td>DRV_MP</td>
<td>Set this flag and make sure the device driver is MP-scalable or MP-safe, that is, uses <code>spinlock()</code>/<code>spinunlock()</code> to avoid...</td>
</tr>
</tbody>
</table>
race conditions. See “Protection and Synchronization for Driver and Transport” on page 189 in this chapter for more information. When this flag is set, the driver cannot use any spl* calls.

DRV_MBLK This flag must be set since the third party network driver is purely based on STREAMS model.

DRV_IP_MULTICAST This flag must be set if a driver supports the IP multicast feature.

DRV_LANC_PROMISC_SUPPORT This flag must be set if a driver supports promiscuous listening.

DRV_NO_FAST_PATH This flag must be set if a driver does not support fast path as described in “Transmission of Message Blocks”.

DRV_CKO This flag must be set if a driver supports TCP or UDP checksum calculations in hardware.

arpmod_name The name of ARP streams helper module. This helper module complements generic ARP module to resolve addresses in networks like Token Ring and Fibre Channel.

ppa PPA number for the interface. The driver should initialize this field with hw_ift->instance_num.

watch_timer For Hewlett-Packard internal use only. This field must be set to 0 for non-Hewlett-Packard devices.

reserved2 For Hewlett-Packard internal use only. This field must be set to 0 for non-Hewlett-Packard devices.

hwift_lock Pointer to a hwift_lock spinlock structure to protect the hw_ift structure. This field is initialized in hw_ift_attach().

next Pointer to next hw_ift structure in list. This field is set by calling the hw_ift_attach() routine during device driver initialization. See “Initializing Networking
Device Drivers" on page 191 in this chapter for detailed information.

The following example shows the initialization of the hw_ift structure. Initialization is generally done in the driver init routine:

```c
struct enet_ift_t *enetift_ptr;
hw_ift_t *hw_ift_ptr;
char mac_addr[6];
struct isc_table_type *isc_ptr; /* pointer to an isc_table structure */

hw_path_t hw_path;

hw_ift_ptr = &(enetift_ptr->hwift);

hw_ift_ptr->mac_type = DEV_ETHER;
hw_ift_ptr->llc_flags = IEEE | SNAP;
hw_ift_ptr->mjr_num = enet_drv_info.drv_info->c_major;
hw_ift_ptr->nm_id = get_nmid();
hw_ift_ptr->instance_num = wsio_iscto_instance(isc_ptr, NULL);
hw_ift_ptr->mtu = ETHER_MTU;
hw_ift_ptr->name = "enet";

io_node_to_path(isc_ptr->card_node, NULL, &hw_path);

io_hw_path_to_str(hw_ift_ptr->hdw_path, NULL, &hw_path);

hw_ift_ptr->state = LAN_DEAD;

bcopy((caddr_t)(mac_addr),
      (caddr_t)(hw_ift_ptr->mac_addr), 6);

hw_ift_ptr->features = DRV_MP | DRV_MBLK;

hw_ift_ptr->arpmod_name = (u_char *)""

hw_ift_ptr->watch_timer = 0;

hw_ift_ptr->ppa = hw_ift_ptr->instance_num;

hw_ift_ptr->reserved2 = 0;
```

### hw_dlpI Structure Description and Initialization

This structure provides support for HP-UX DLPI connections. This structure should be initialized to zero. Further discussion of structure fields is not provided.
Protection and Synchronization for Driver and Transport

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 MP systems. Driver data structures also need protection against interrupts. HP-UX transport networking adopted the OSF/Encore spinlock protection model in order to gain parallelism and provide scalable network performance. The spinlock scheme provides finer granularity locks, protecting data structures at finer levels, as opposed to grabbing a global network lock. More information on spinlocks is available in “Multiprocessing”, Chapter 4 of this manual, and spinlock(KER2) in the HP-UX Device Driver Reference manual.

NOTE

Each spinlock causes a busy-wait. Device-driver developers should be aware of the impact on system performance caused by the frequency of acquiring a spinlock and the duration of holding a spinlock.

As discussed in previous sections, the data structure in the network interface layer that link device drivers to the protocol layer is the hw_ift. The drivers have their own data structures—driver control block, send and receive management structures. These data structures are protected by using spinlocks.

hw_ift Structure Protection

One spinlock, the hwift_lock field, in the hw_ift structure is defined to protect the access to the structure fields.

The macros to acquire or release the hwift_lock spinlock to protect hw_ift structure fields are defined below.

HW_IFT_LOCK (hw_ift_ptr)  Acquire a spinlock on hwift_lock.

    hw_ift_ptr: Pointer to an hw_ift structure.

HW_IFT_UNLOCK(hw_ift_ptr)  Release previously acquired hwift_lock spinlock.
Creating Networking Device Drivers

Protection and Synchronization for Driver and Transport

hw_ift_ptr: Pointer to a hw_ift structure.

NOTE

The hwift_lock spinlock is allocated and initialized by the
hw_ift_attach() routine. As a result, the HW_IFT_LOCK() and
HW_IFT_UNLOCK macros are not available until returning from the
hw_ift_attach() routine.

Driver Structure Protection

DLPI and network drivers use spinlocks to protect their internal data
structures. HP-UX pre-defines the order (major order) for spinlocks for
LAN and STREAMS drivers to avoid deadlock conditions when
non-direct code paths are executed due to interruptions, such as faults,
traps and interrupts.

Drivers can increase concurrency by choosing to have finer granularity
locks. Since the major lock order is pre-defined by HP-UX, drivers could
use different minor order spinlocks to protect access to data structures.
For example, a WSIO driver could use one lock for transmit path and one
for receive path data structures. This would allow the driver to receive
and transmit concurrently.

A list of the relative pre-defined lock orders for spinlocks used by HP-UX
LAN products is shown below.

LAN_LANX_LOCK_ORDER Lock order for a spinlock used by HP-UX
LAN device drivers, such as btlan3 and lan2, to protect
local data structures. This lock order should be used by
all third-party networking device drivers during
initialization of a spinlock used to protect device-driver
structures.

LAN_HWIFT_LOCK_ORDER Lock order for spinlock hwift_lock,
defined in sio/lan_dlpikrn.h, and the lock order
protecting the embedded MIB structure.

STREAMS_USR1_LOCK_ORDER Lock order for spinlock used by
STREAMS drivers to protect their data structures.
Initializing Networking Device Drivers

In HP-UX 11.0, developing a network interface driver involves developing a STREAMS DLPI driver and a WSIO network interface driver. A DLPI driver is part of STREAMS/UX and is used by the file system for device open and close. For this reason the DLPI driver is both a STREAMS and CDIO driver. The network interface driver is a WSIO driver. The initialization of one driver does not depend on the other. Initialization for each driver is described below.

STREAMS DLPI Driver Initialization

For a detailed description of generic STREAMS driver development, refer to the STREAMS/UX for HP900 Reference Manual. This section briefly explains the initialization process with the help of excerpts from a sample driver dlpi_enet.

The install routine of a STREAMS DLPI driver, driver_install(), should call CDIO install install_driver() (CDIO3) and STREAMS/UX install str_install() functions.

The CDIO system requires the following data structures to be defined and initialized before calling install_driver() in driver_install().

drv_ops_t dlpi_enet_drv_ops;
drv_info_t dlpi_enet_drv_info = {
    "dlpi_enet",    /* driver name */
    "pseudo",      /* driver class */
    DRV_CHAR | DRV_PSEUDO | DRV_MP_SAFE,
    /* type */
    -1,           /* block major number */
    -1,           /* character major number */
    NULL, NULL, NULL /* always NULL */
};

STREAMS/UX requires that streams_info_t be initialized as shown in the following code sample. This structure is passed in the call str_install().

static struct module_info enet_rminfo = {
    5050, "enetlan", 0, 65536, 65536, 1
};
static struct module_info enet_wminfo = {
    5050, "enetlan", 0, 65536, 1, 1
};
Creating Networking Device Drivers

Initializing Networking Device Drivers

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 dlpi_enet_str_info = {
    "dlpi_enet",          /* name */
    -1,                   /* dynamic maj # */
    { &enet_rinit, &enet_winit, NULL,NULL},
    /* streamtab */
    STR_IS_DEVICE|MGR_IS_MP|STR_SYSV4_OPEN,
    /* stream flags */
    SQLVL_QUEUE,          /* sync level */
    ""
};

The install routine for the sample driver is shown below.
Please note that the install routine should be prefixed with the driver name.

int
dlpi_enet_install(void)
{
    int rv;
    bzero((caddr_t)&dlpi_enet_drv_ops, sizeof(drv_ops_t));
    msg_printf("dlpi_enet: install\n");
    if (!(rv = install_driver(&dlpi_enet_drv_info,
                              &dlpi_enet_drv_ops))) {
        if(rv = str_install(&dlpi_enet_str_info)) {
            uninstall_driver(&dlpi_enet_drv_info);
            msg_printf("dlpi_enet: install failed\n");
        }
    }
    return rv;
}

WSIO Network Driver Initialization

Each HP-UX PCI networking device driver must have a driver_install (WSIO_DRV) routine and a driver_attach (WSIO_DRV) routine. If the networking device driver controls and interacts with a hardware device, the driver is also required to have an interrupt service routine to handle the device interrupts.

The following brief descriptions of the required install, attach, and
initialization routines are provided as an introduction to networking device driver initialization.

int

**driver_install()**  An entry of `driver_install()` is called during the I/O system configuration process. When the `driver_install()` routine is called, it hooks the `driver_attach()` entry on the top of a linked list of attach routines for all of the interface drivers in the system.
Creating Networking Device Drivers

Initializing Networking Device Drivers

```c
#include <sysconf.h> #ifdef __LP64__
int driver_attach(uint32_t product_id, struct isc_table_type *isc_ptr)
#else
driver_attach(PCI_ID product_id, struct isc_table_type *isc_ptr)
#endif

product_id Four bytes of PCI product ID.
isc_ptr Pointer to `isc_table_type` structure.

```c
void driver_init(struct isc_table_type *isc_ptr)
```
isc_ptr Pointer to `isc_table_type` structure.

```c
int driver_isr(struct isc_table_type *isc_ptr, caddr_t cb_ptr)
```
isc_ptr Pointer to `isc_table_type` structure.

```c
cb_ptr Pointer to driver control block. This `cb_ptr` is driver developer defined and passed as a pointer through the `isrlink()` routine during the `driver_attach()` or `driver_init()` routines.
```

The `driver_attach()` and `driver_install()` initialization procedures are common to all HP-UX device drivers. More detailed treatment of each step is presented in Chapter 5, “Writing a Driver.”

Networking-specific routines are discussed below.

**Calling driver_install()**

When the HP-UX system is configured through the `config` command, a table of `driver_install()` entry points is created from information in `/stand/system`.

When `driver_install()` is called by the I/O system configuration process through the `driver_install()` entry point configured in the system, the `driver_install()` routine places the `driver_attach()` entry in a table of drivers to be called at configuration time. `driver_install()` then calls the `wsio_install_driver()` routine to register the driver with the I/O subsystem and returns any error.

The following example shows a call to `driver_install()`.

```c
static drv_ops_t enet_drv_ops = {
    NULL, /* open */
    NULL, /* close */
```
Creating Networking Device Drivers

Initializing Networking Device Drivers

```c
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 drv_info_t enet_drv_info = {
  "enet", /* driver name */
  "lan", /* class name */
  DRV_CHAR | DRV_SAVE_CONF | DRV_MP_SAFE, /* driver flags */
  -1, /* block major number */
  -1, /* character major number */
  NULL, NULL, NULL, /* structures always set to NULL */
};
static wsio_drv_data_t enet_data = {
  "enet", /* for matching probes with drivers */
  T_INTERFACE, /* type of hardware, dev or IF */
  DRV_CONVERGED, /* driver flag */
  NULL, /* minor number build routine */
  NULL, /* minor number decode routine */
};
static wsio_drv_info_t enet_wsio_info = {
  &enet_drv_info, /* driver info */
  &enet_drv_ops, /* driver ops */
  &enet_data, /* driver data */
};
extern int (*pci_attach)(); /* to attach PCI driver to system */
int (*enet_saved_attach)();
```
Creating Networking Device Drivers

Initializing Networking Device Drivers

```c
int enet_install()
{
    enet_saved_attach = pci_attach;
    /* save the current top entry */
    pci_attach = enet_attach;
    /* link attach entry to list */

    return(wsio_install_driver(&enet_wsio_info));
}
```

/*
* enet_wsio_info is the driver header information that contains
* the information normally found in /usr/conf/master.d
*/
}

**Calling driver_attach()**

Use the `driver_attach()` routine to determine whether the product ID passed in matches the `driver_attach` device and vendor ID. If the IDs do not match, the `driver_attach()` routine calls the next attach routine in the chain by calling the `*driver_saved_attach()` routine.

NOTE

The `driver_attach()` routine may be called many times before a match is found. For the device in the first slot, the associated `driver_attach()` routine is called by the number of devices in the PCI backplane. For the device in the last slot of the PCI backplane, the associated `driver_attach()` routine is called only once.

When the `driver_attach()` routine recognizes the device ID, it takes actions such as allocating and initializing its driver control blocks and PCI I/O registers. The routine also calls the `pci_read_cfg_uint32_isc()` routine to get the system PCI configuration of the device (if needed), `driver_attach()` also sets up driver initialization call and finally calls the `isc_claim()` to claim the device. A sample `driver_attach()` routine is shown below:

```c
struct gfsw enet_gfsw;
...
...
int
#ifdef __LP64__
enet_attach( uint32_t id, struct isc_table_type *isc)
#else
enet_attach( PCI_ID id, struct isc_table_type *isc)
#endif
{
    msg_printf("enet attach id = %x\n",id);
    #ifdef __LP64__
        /* Support for PCI only */
        if (!(id.vendor_id==DEV_VENDORID &&
            id.device_id==DEVDEVICEID)) {
            return enet_saved_pci_attach(id, isc);
        }
    #else
        if (!(id == DEV_ID)) {
            return enet_saved_pci_attach(id, isc);
        }
    #endif
    isc->gfsw = &enet_gfsw;
    CONNECT_INIT_ROUTINE(isc,enet_init);
    isc->gfsw->diag = (int (*)())NULL;
#endif
    #ifdef __LP64__
        isc->if_id = (int)(id & 0x0000ffffU);
    #else
        isc->if_id = (int)id.device_id;
    #endif
    isc_claim(isc, &enet_wsio_drv_info);
Creating Networking Device Drivers

Initializing Networking Device Drivers

```c
io_inform("PCI 100 Base TX Controller found", isc, 5);
return enet_saved_pci_attach(id, isc);
}
```

HP-UX initialization continues by calling a `driver_init()` routine to begin driver initialization. It allocates the driver control block and driver data structures, gets and sets PCI configuration information, links the driver ISR to the PCI interrupt, and initializes and resets the controller hardware. The following is the skeleton initialization function showing PCI configuration and linking of the driver ISR.

```c
int
enet_init(struct isc_table_type *isc)
{
    enet_ift_t *enet_iftp;
    ...
    ubit32 base_addrp,id, revid, latency_timer, int_reg;
    ubit32 sub_id, ssid, cfda, csr6;
    BUS_TRANS_DESC desc;
    ubit32 error;
    ...
    ...
    /*
    * Allocate driver control block - enet_iftp
    */
    ...
    /*
    * Obtain memory for Transmit and Receive Descriptor Rings and any additional driver data structures*/
    ...
    /*
    * Get/Set PCI configuration
    */
    pci_read_cfg_uint32_isc(isc,SSID,&ssid);
    enet_iftp->sub_id = (ubit16)(ssid >> 16) ;
    enet_iftp->sub_vendor_id = (ubit16)(ssid & 0x0000ffff) ;

    /* Read the Configuration ID information */
    pci_read_cfg_uint32_isc(isc,CFID,&id);

    /* Read the Configuration Revision information */
    pci_read_cfg_uint32_isc(isc,CFRV,&revid);

    /* Read the Configuration Interrupt information */
    pci_read_cfg_uint32_isc(isc,CFIT,&int_reg);
```
Creating Networking Device Drivers

Chapter 7

Networking Driver-Specific Initialization

If the initialization is successful, the `driver_init()` routine proceeds with the following steps:

```c
/* Read the Configuration Driver Area information */
pci_read_cfg_uint32_isc(isc, CFDA, &cfda);

pci_write_cfg_uint32_isc(isc, CFDA, cfda);

....

/* Turn on PCI memory access and bus master capability on host */
pci_write_cfg_uint8_isc(isc, CFCS, CFCS_MEMORY_SPACE_ACCESS
CFCS_MASTER_OPERATION
CFCS_PARITY_ERROR_RESPONSE
CFCS_SYSTEM_ERROR_ENABLE
CFCS_I_O_SPACE_ACCESS);

.../*
 * Init and reset the controller */

/*
 * Perform general enet_ift initialization */

.../*
 * Setup hwift structure */

.../*
 * Attach hwift to global list */

hw_ift_attach(&enet_iftp->lancift.hwift);

.../*
 * Setup interrupt handler. The -1 option will cause
 * isrlink to read the configuration interrupt
 * register to determine the irq for the PCI device */

isrlink(isc, enet_isr, -1, (long)isc, (long)enet_iftp);

...`
Creating Networking Device Drivers

Initializing Networking Device Drivers

- Initializes the MIB structure and the `hw_dlpi` and `hw_ift` structures (see the preceding sections, “hw_ift_t Structure Description and Initialization” and “hw_dlpi Structure Description and Initialization”, for details).

- Calls the `hw_ift_attach()` routine to link the `hw_ift` structure to a global list of `hw_ift` structures of active interfaces. The `hw_ift_attach()` routine is defined as:

  ```c
  hw_ift_attach(hw_ift_t *hw_ift_ptr)
  hw_ift_ptr       Pointer to the associated hw_ift_t structure
  ```
Protocol Binding and Demultiplexing

Protocol binding and demultiplexing is the mechanism a networking driver uses to associate (bind) an upper-layer protocol to a device. This binding ensures that the driver correctly demultiplexes and delivers inbound packets to the corresponding upper-layer protocol, based on the upper-layer protocol's bind request.

To correctly demultiplex inbound packets, a networking driver must:

- Obtain protocol-specific information during protocol binding.
- Obtain packet-specific information.
- Process packets and information by the upper level protocols.

Table 7-1, “Packet Type, Protocol Kind, and Protocol Value,” summarizes the information a networking driver requires to demultiplex inbound packets for corresponding upper-layer protocols. More detailed information is provided in the section “DLPI Sequence” on page 205.

<table>
<thead>
<tr>
<th>Packet Type</th>
<th>Protocol Kind</th>
<th>Protocol Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ethernet type</td>
<td>LAN_TYPE</td>
<td>TYPE value</td>
</tr>
<tr>
<td>SNAP type</td>
<td>LAN_SNAP</td>
<td>OID + extended SNAP info</td>
</tr>
<tr>
<td>IEEE 802.2 LLC type</td>
<td>LAN_SAP</td>
<td>SAP value</td>
</tr>
</tbody>
</table>

Protocol kind is the type of protocol to bind. Interpretation of the protocol value field depends on the protocol kind value. “Protocol Kinds and Values” on page 510, Appendix D, lists the names of the available kinds of protocols to bind and some of the available protocol values.

When the networking driver binds a protocol with protocol kind and type values, the driver knows what kind of packets to handle for that bind. The networking driver processes inbound packets on the interrupt control stack (ICS) for all the protocol binds by calling an associated STREAMS queue. To do this, it calls `putnext()` (see the STREAMS/UX for the HP 9000 Reference Manual) in the device driver's interrupt service routine. The driver must use the protocol ID that was carried in the dl_proto_info field of the DL_HP_BIND to pass the packet to the
right Stream that is logged (see “DLPI Sequence” on page 205 for details).

Protocol Binding and Unbinding

Each upper-layer protocol issues a bind request to the networking driver to effect binding. The driver is responsible for keeping track of all upper layer protocols currently bound to it. The networking driver also must have a way to unbind a protocol upon request.

Protocol Demultiplexing

One of the main functions of the device driver’s interrupt service routine is to dispatch inbound packets to the appropriate upper layer protocol. To achieve that, the interrupt service routine in the driver must:

1. Distinguish packet protocol format and type:
   - Ethernet
   - IEEE 802.2 Link Level Control (LLC) (non-SNAP)
   - SNAP (IEEE 802.2 LLC extended)
2. Locate the proper inbound packet service routine or queue for each valid incoming packet.

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-type packet:

- If the value of the TYPE field of an inbound packet is equal to or greater than 0x600, the packet is an Ethernet-type packet. The protocol kind of the packet is LAN_TYPE, and the protocol value is the TYPE field specified in the packet (see Appendix D, “Protocol Kinds and Values”).
- If the value of the TYPE field is less than 0x600, the packet could be an IEEE 802.2 LLC-type packet, SNAP-type or non-SNAP-type.

To determine whether the packet is a SNAP-type IEEE 802.2 LLC packet:
• The packet is considered to be a SNAP packet (defined in IEEE 802.1a) if both the DSAP and the SSAP values are 0xAA. The protocol kind of the packet is LAN_SNAP, the protocol value is 0xAA, and the protocol value extended is the 5-byte SNAP protocol data specified in the SNAP header (see “Protocol Kinds and Values” on page 510, Appendix D).

• Otherwise, it is an IEEE 802.2 LLC non SNAP-type packet. The protocol kind is LAN_SAP and the protocol value is the DSAP field that is specified in the packet (see “Protocol Kinds and Values” on page 510, Appendix D).

The relationships of protocol kind, protocol value, and protocol processing for different types of packets are shown in Table 7-1, on page 201.

After the device driver has found the protocol kind and protocol value in an inbound packet, the driver must locate the protocol input queue that corresponds with the bind request previously received from an upper layer protocol. This queue information must be stored by the driver during binding.

If the upper layer requires header stripping, the device driver strips off the link-level control (LLC) header before passing the inbound packet to the upstream queue.
Message block and queue_t functions and macros

The message block and queue functions and macros are defined by STREAMS/UX. The reader is referred to STREAMS/UX for the HP 9000 Reference Manual for further information.

The list of commonly used message block functions.

allocb() Allocate a message block
freemsg() Free a message block
pullupmsg() Concatenate and align the data stored in complex message
adjmsg() Adjust the length of the message
dupmsg() Duplicate a simple or complex message

The following is the list of queue functions commonly used in a STREAMS driver.

putq() queue message to be processed by queue service procedure
putnext() call queue's put procedure
canput() test whether queue can receive message
qreply() send the message back upstream
OTHERQ() other queue in the queue pair
streams_put(), streams_put_release() allow non-STREAMS/UX (e.g driver ICS) to 'put' in a queue
DLPI Sequence

The Data Link Provider Interface (DLPI) specifies a STREAMS-based kernel implementation of the ISO Data Link Service Definition (ISO 8886) and Logical Link Control (ISO 8802/2 LLC). DLPI allows a data link service user to access and use a variety of conforming data link services without special knowledge of the provider’s protocol. The interface specifies access to data link service providers and does not define a specific protocol implementation. HPUX transports (e.g. TCP/IP, UDP, OSI) are now STREAMS modules. Third parties are expected to develop STREAMS DLPI driver conforming to DLPI version 2.0 to support their network interface drivers and can not depend on the HP DLPI implementation. This section provides information about how third party drivers can integrate into a STREAMS/UX framework in HP-UX.

Two styles of DLPI provider are defined by the DLPI document, distinguished by the way they enable a DLPI user to choose a particular physical point of attachment (PPA). The style 1 provider assigns a PPA based on the major/minor device the DLPI user opened. The style 2 provider requires a DLPI user to explicitly identify the desired PPA by using a special attach service primitive. This document illustrates the development of Style 2 DLPI driver.

Device file, Interface name and PPA number

DLPI users can access DLPI providers through generic DLPI device files (i.e a device file corresponding to a DLPI STREAMS driver). A DLPI device file can be created by mknod(2) or insf(1M) by using device driver information from lsdev(1M). The following example shows the devices enet (sample WSIO PCI driver) and dlpi_enet (sample DLPI STREAMS driver). The device files created for the STREAMS DLPI driver are also shown.

```
# lsdev
.............................................
.............................................
239          -1         enet            lan
240          -1         dlpi_enet       pseudo

# ll /dev/enet*
lrwxrwxrwx  1 root/sys  11 Apr 12 18:47 /dev/enet ->
```

Chapter 7 205
Creating Networking Device Drivers

DLPI Sequence

/dev/dlpi_enet

```
# ll /dev/dlpi_enet*
crw-rw-rw-   1 root sys        72 0x0000f0 Apr 12 18:46 /dev/dlpi_enet
crw-rw-rw-   1 root sys      240 0x000000 Apr 12 18:46 /dev/dlpi_enet0
```

lanscan(1M) lists all the LAN interfaces in the system from the list of hw_ift_ts (every network interface driver should do hw_ift_attach() during initialization). From this list we can identify the interface name and PPA numbers. Please refer to section, “Initializing Networking Device Drivers” on page 191, for details about hw_ift_attach().

The following output from lanscan illustrates the interface name and PPA numbers for the sample WSIO network driver. As can be seen the sample driver has ‘attached’ to LAN interfaces at hardware paths 8/0/1/0 and 8/0/2/0.

**Table 7-2**  
**lanscan Output**

<table>
<thead>
<tr>
<th>Hdw Path</th>
<th>Station Address</th>
<th>Card Im#</th>
<th>Hdw State</th>
<th>Net-Interface Name</th>
<th>PPA</th>
<th>NM ID</th>
<th>MAC Type</th>
<th>HP-DL PT Support</th>
<th>DLPI Majr#</th>
</tr>
</thead>
<tbody>
<tr>
<td>8/16/6</td>
<td>0x0060 B07ED BF0</td>
<td>0</td>
<td>UP</td>
<td>lan0 snap0</td>
<td>1</td>
<td>ETHE R</td>
<td>Yes</td>
<td>119</td>
<td></td>
</tr>
<tr>
<td>8/0/1/0</td>
<td>0x0060 B07A2 21E</td>
<td>1</td>
<td>UP</td>
<td>enet1</td>
<td>2</td>
<td>ETHE R</td>
<td>No</td>
<td>240</td>
<td></td>
</tr>
<tr>
<td>8/0/2/0</td>
<td>0x0060 B0B2D 850</td>
<td>2</td>
<td>UP</td>
<td>enet2</td>
<td>3</td>
<td>ETHE R</td>
<td>No</td>
<td>240</td>
<td></td>
</tr>
</tbody>
</table>

**IP and ARP Configuration**

Once the interface name and the PPA number are known, ifconfig(1M) is used to configure IP and ARP. When ifconfig is done for enet1 listed by
lanscan above, the IP and ARP streams are setup as listed in the steps below.

1. `ifconfig` opens device file `/dev/enet` and understands that PPA configured is 1
2. `ifconfig` issues an ioctl to push IP module on top of `dlpi_enet` driver
3. `ifconfig` issues another ioctl to issue attach and bind request for PPA 1
4. `ifconfig` opens device file `/dev/enet` and issues ioctl to push ARP on top of `dlpi_enet` driver
5. `ifconfig` again performs step 3 for ARP/dlpi_enet stream
6. `ifconfig` opens `/dev/ip` and uses it as dummy multiplexer and IP/dlpi_enet and ARP/dlpi_enet streams are linked under dummy multiplexer.

**STEAMS DLPI and network driver overview**

Figure 7-4, “DLPI Sequence,” shows the basic structure of STEAMS DLPI driver implementation in HP-UX. Two main data structures, `enet_hw_ift_t` and `enet_hw_dlpi_t` establish the linkage between the STEAMS driver and the network device driver; it is not an exported interface by HP-UX. This is an example implementation. Third party developers may define their own interface to address their design needs. Please refer to “Initializing Networking Device Drivers” on page 191 for initialization of `enet_hw_ift_t` and `enet_hw_dlpi_t`. 
Creating Networking Device Drivers

DLPI Sequence

Figure 7-4 DLPI Sequence

Application Layer

putmsg()

DLS user

getmsg()

User Space

Kernel Space

Streams Head
/dev/dlpi and /dev/dlpiX

Network Protocol Layer

Data Link Layer

STREAMS DLPI driver

driver_wput()

driver_wsrv()

_fast_in()

_unitdata_out()

_INTR()

PROC_IOCTL()

INFO, ATTACH,
BIND, PPA_REQ,

UNITDATA_OUT()

_PCI bus

PCI bus

WSIO network driver

driver_install()
driver_attach()
driver_link()
driver_init()

(*dlpi_ioctl())

(*dlpi_output())

(*dlpi_build_hdr())

(*dlpi_output())

(driver_open())

_driver_wput()

_times()
Creating Networking Device Drivers

DLPI Sequence

Chapter 7

The general STREAMS/DLPI buffer/message processing is done in the STREAMS DLPI driver. The network driver implements the device initialization, input, output, and control functions. Modularizing functionality this way makes it possible to support different types of LAN interfaces with one STREAMS DLPI driver. The high level functions in the drivers and their interaction is shown in Figure 7-4 on page 208. The next two sections briefly describe the STREAMS DLPI and network interface drivers.

Synchronization Between Device Driver and DLPI driver

For a non-STREAMS character I/O mechanism, synchronization between device driver and device can be accomplished by having the device driver sleep, via the sleep() kernel call, on a unique number, typically, an object address, while waiting for the request to complete.

Upon receiving the request completion information from the device, the device driver then resumes the process via the wakeup() kernel call. For STREAMS, however, this kind of sleep-wakeup synchronization mechanism is not permitted because STREAMS may run on either the ICS or the STREAMS scheduler context stack. Synchronization between DLPI and the device driver is not defined in the DLPI 2.0 documentation.

The sample DLPI driver has an enet_dlpi_wakeup() routine to support the necessary synchronization between DLPI and networking device drivers.

This enet_dlpi_wakeup() routine, which simulates the STREAMS environment wakeup() kernel call, is defined in sio/libdlpi.a as:

```
void enet_dlpi_wakeup(caddr_t addr_ptr)
```

Parameter:

addr_ptr Address of an object to wakeup. It should correspond to the negative value returned by the (*dlpi_ioctl)() routine.

Certain actions are required of the network device driver when device control requests passed through the (*dlpi_ioctl)() routine have a negative returned value.

The following rules summarize actions each networking device driver must take in dealing with such dlpi_ioctl requests:

1. The control request does one of the following:
Creating Networking Device Drivers

DLPI Sequence

- If the control request completes immediately with no error, the (*dlpi_ioctl()) routine should immediately return 0 to DLPI.
- If the control request completes immediately with an error, the error should be returned as a positive value (from errno.h).
- If the control request cannot complete immediately (that is, the driver must make a request to the hardware), the device driver must hold the enet_hw_ift_lock and return a globally unique negative value to DLPI.

2. Some time later, an interrupt or timeout occurs, and the device driver interrupt service routine figures out if the interrupt is for a previously blocked and waiting request.

3. The device driver completes the previous (*dlpi_ioctl()) by putting the results in the appropriate place for that ioctl.

4. The device driver calls the enet_dlpi_wakeup() routine with the address of the sleep object that the (*dlpi_ioctl()) routine previously returned to DLPI.

STREAMS synchronization

HP-UX STREAMS supports MP scalable drivers and modules. STREAMS/UX provides five levels of parallelism, called queue, queue pair, module, elsewhere, and global. The queue synchronization level provides the most concurrency. Refer to STREAMS/UX for HP 9000 Reference Manual for detailed information. One can configure the amount of parallelism for modules and drivers by specifying the synchronization level in streams_info_t during str_install(). The sample DLPI STREAMS driver uses queue synchronization level.

Entering STREAMS from ICS

In order to enter the STREAMS framework correctly from non-STREAMS/UX code, the STREAMS/UX provides streams_put utilities. The driver ICS function can call streams_put(), 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 safety manipulate the queue and access the same data structures as the queue's put routine. streams_put_release() executes the streams_put functionality on a specified processor. Refer to STREAMS/UX for HP 9000 Reference Manual for further information.
STREAMS DLPI Driver

This section discusses the STREAMS DLPI driver briefly. The objective here is to present the code flow of DLPI sample driver, `dlpi_enet`, as background to the sample driver code. Refer to the sample driver code for details. This section contains the following topics:

- DLPI driver data structures
- Open and close routines
- Control functions that describe processing of DLPI primitives like attach/detach, bind/unbind, enable/disable multicast, enable/disable, and promiscuous
- The main I/O path
- DLPI primitives supported in the sample driver

Major Data Structures

NOTE  
These data structures are part of Sample driver. They do not constitute any interface defined by HP-UX.

**enet_hw_if_t**

The information in this data structure relates to the network device driver that associates itself with STREAMS DLPI, `dlpi_enet`. There is one such data structure for each network device under `dlpi_enet`. The structure itself is shown below.

```c
typedef struct enet_hw_if {  
    hw_dlpi_t hp_dlpi;
    uint32_t mac_type;
    uint32_t llc_flags;
    uint32_t mjr_num;
    uint32_t nm_id;
}
```
Creating Networking Device Drivers
STREAMS DLPI Driver

```c
uint32_t     instance_num;
uint32_t     mtu;
char         *name;
uint8_t      hdw_path[MAX_HDW_PATH_LEN];
uint32_t     hdw_state;
uint32_t     mac_addr_len;
uint8_t      mac_addr[MAX_MAC_ADDR_LEN];
uint32_t     features;
uint32_t     ppa;
uint32_t     watch_timer;
lock_t       *hwift_lock;
struct enet_ift    *next;
}enet_hw_ift_t;
```

The table below lists all the fields along with their purpose.
## Table 7-3  enet_hw_ift_t Data Fields

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>hp_dlpi</td>
<td>structure with driver 'entry' points (see below)</td>
</tr>
<tr>
<td>mac_type</td>
<td>MAC type - Ethernet, FDDI, ATM see lan_dlpikern.h</td>
</tr>
<tr>
<td>llc_flags</td>
<td>supported LLC encapsulation methods</td>
</tr>
<tr>
<td>mjr_num</td>
<td>major number of the device</td>
</tr>
<tr>
<td>nm_id</td>
<td>get network management ID</td>
</tr>
<tr>
<td>instance_num</td>
<td>return from wsio_isc_to_instance() - WSIO instance number</td>
</tr>
<tr>
<td>mtu</td>
<td>MTU of the link</td>
</tr>
<tr>
<td>name</td>
<td>name of the network driver - sample driver is enet</td>
</tr>
<tr>
<td>hdw_path</td>
<td>hardware path of the network device</td>
</tr>
<tr>
<td>hdw_state</td>
<td>state of the hardware</td>
</tr>
<tr>
<td>mac_addr_len</td>
<td>length of MAC address in octets</td>
</tr>
<tr>
<td>mac_addr</td>
<td>MAC address</td>
</tr>
<tr>
<td>features</td>
<td>features of driver - DRV_* in lan_dlpikrn.h</td>
</tr>
<tr>
<td>ppa</td>
<td>PPA number of the interface -&gt; set to instance number</td>
</tr>
<tr>
<td>watch_timer</td>
<td>watch dog timer support</td>
</tr>
<tr>
<td>hw_ift_lock</td>
<td>MP protection</td>
</tr>
<tr>
<td>next</td>
<td>link to next enet_hw_ift_t</td>
</tr>
</tbody>
</table>

### enet_hw_dlpi_t

This structure is part of enet_hw_ift_t and contains driver service functions exported to DLPI driver. It also contains MIB for the network interface.

```c
enet_hw_dlpi_t
{
```
Creating Networking Device Drivers

STREAMS DLPI Driver

caddr_t drv_data;
intptr_t(*dlpi_output)();
int(*dlpi_build_hdr)();
intptr_t(*dlpi_ioctl)();
caddr_tmib_ptr;

int mib_len;
}

Table 7-4

<table>
<thead>
<tr>
<th>field name</th>
<th>purpose of the field</th>
</tr>
</thead>
<tbody>
<tr>
<td>drv_data</td>
<td>Driver dependent data structure to be passed while calling driver 'entry' points</td>
</tr>
<tr>
<td>dlpi_output</td>
<td>Output routine for packet</td>
</tr>
<tr>
<td>dlpi_build_hdr</td>
<td>Routine provided for building header</td>
</tr>
<tr>
<td>dlpi_ioctl</td>
<td>Routine provided by driver to process device dependent part of DLPI primitives</td>
</tr>
<tr>
<td>mib_ptr</td>
<td>Pointer to mib stats</td>
</tr>
<tr>
<td>mib_len</td>
<td>Byte count of mib data pointed by mib_ptr</td>
</tr>
</tbody>
</table>

enet_dlpi_data_t

This data structure contains STREAMS DLPI driver information for a Stream that is open currently with the driver.

typedef struct _enet_dlpi {
    enet_hw_ift_t*hwiftp;
    cred_t*cred;
    queue_t*queue_ptr;
    dev_tenet_dev;
    uint32_t dlsap_addr_length;
    uint8_t dlsap_addr[MAX_DLSAP_LEN];
    uint16_t service_mode;
} enet_dlpi_data_t;
The following table lists the fields along with explanation.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>hwiftp</td>
<td>The interface that associated with this open stream</td>
</tr>
<tr>
<td>cred</td>
<td>credential structure of the user who opened this stream</td>
</tr>
<tr>
<td>queue_ptr</td>
<td>queue pointer to the read queue of the stream</td>
</tr>
<tr>
<td>enet_dev</td>
<td>dlpi_enet device number</td>
</tr>
</tbody>
</table>

Table 7-5 enet_dlpi_data_t Data Fields
<table>
<thead>
<tr>
<th>Field Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>dlsap_addr_length</td>
<td>length of DLSAP address</td>
</tr>
<tr>
<td>dlsap_addr[]</td>
<td>MAC addr + SAP</td>
</tr>
<tr>
<td>service_mode</td>
<td>Only DL_CLDLS supported in the sample driver</td>
</tr>
<tr>
<td>curr_state</td>
<td>DLPI state</td>
</tr>
<tr>
<td>xidtest_flag</td>
<td>dl_xidtest_flg from DL_BIND_REQ; indicates to the driver that XID and/or TEST responses for this stream are to be generated by DLPI driver</td>
</tr>
<tr>
<td>mac_type</td>
<td>interface MAC type</td>
</tr>
<tr>
<td>mac_mtu</td>
<td>interface MTU</td>
</tr>
<tr>
<td>dlsap_ptr</td>
<td>dlsap_t structure list of logged SAPs</td>
</tr>
<tr>
<td>ssap</td>
<td>first SAP logged on stream</td>
</tr>
<tr>
<td>sxsap</td>
<td>first extended SAP logged on stream</td>
</tr>
<tr>
<td>enet_mcast_list</td>
<td>list of multicast addresses on this stream</td>
</tr>
<tr>
<td>promiscuous_flag</td>
<td>this flag is set to the promiscuous level specified in the DL_PROMISCON_REQ primitive</td>
</tr>
<tr>
<td>promisc_filter</td>
<td>this field is set to 1 if the stream has been bound with any SAP</td>
</tr>
<tr>
<td>noloopback_flag</td>
<td>This is set when application wants to handle loopback. This flag is set when DLPI_SET_NOLOOPBACK ioctl is issued. DLPI turns on the MSGNOLOOP flag in mblk message on every outbound message so that driver won't loop back the packet</td>
</tr>
<tr>
<td>no_src_routing</td>
<td>Set when DLPI_NO_SRC_ROUTING is issued</td>
</tr>
<tr>
<td>arp_stream</td>
<td>set if this is ARP stream</td>
</tr>
<tr>
<td>ip_stream</td>
<td>set if this is IP stream</td>
</tr>
<tr>
<td>fast_path</td>
<td>set if application requests to set up fast path</td>
</tr>
</tbody>
</table>
The DLPI driver can be accessed via either a regular device or a clone of the original device. The major number of the device file for a cloneable driver must be the clone driver’s major number, 72. (Refer to STREAMS/UX for HP 9000 Reference Manual for mode details of clone driver). The minor number is set to the real major number of the device. The clone open is useful because the application does not need to keep track of which minor number is available and does not need to deal with multiple device files.

As can be seen from the following example, /dev/dlpi_enet is a clone device of dlpi_enet driver. While /dev/dlpi_enet0 is a regular device file with a device major number, 240, and minor number, 0.

```bash
# lsdev
.............................................
.............................................
239          -1 enet lan
240          -1 dlpi_enet pseudo
#ll /dev/enet*
lrwxrwxrwx   1 root       sys 11 Apr 12 18:47 /dev/enet -> /dev/dlpi_enet
#ll /dev/dlpi_enet*
```

**Opening and Closing a Driver**

Table 7-5 enet_dlpi_data_t Data Fields

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>fast_path_pkt_type</td>
<td>is the fast path packet type</td>
</tr>
<tr>
<td>fast_path_llc_length</td>
<td>is the LLC header length used in the fast path</td>
</tr>
<tr>
<td>pre_state</td>
<td>keeps the state before a pending ioctl or control request with the driver, so that when the request is complete the streams can be set to correct state</td>
</tr>
</tbody>
</table>

This array holds enet_dlpi_data_t pointers to keep track of the open streams.
Creating Networking Device Drivers

STREAMS DLPI Driver

When a clone device is opened, the clone driver invokes the DLPI driver's open routine with the CLONEOPEN flag set. The open function `dlpi_enet_open()` allocates the `enet_dlpi_data_t` for the stream being opened and initializes it. The minor number of a normal device file open is used as the index into `enet_dlpi_data_ptr_arr[]` to store and access `enet_dlpi_data_t` for the stream. The indexes 1 to 99 are reserved for regular open in sample driver. For clone opens, an unused minor number starting from 100 is allocated. The `enet_dlpi_data_t` for the stream is stored in the `enet_dlpi_data_ptr_arr` indexed by new minor number.

Control Functions

The function `dlpi_enet_wput()`, the streams put procedure, calls various control functions to service DLPI M_PROTO and M_PCPROTO messages with local management primitives (information reporting, attach, bind, and others like multicast, promiscuous). This function consists of a switch table that calls the service function based on message `dl_primitive`. The following is a list of service functions and their function:
Table 7-6  Message Service Functions

<table>
<thead>
<tr>
<th>Function Name [prefixed by enet_dlpi]</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>_info()</td>
<td>Service function for DL_INFO_REQ. The information is returned upstream in structure dl_info_ack_t. If the PPA is not attached yet, then mac type and mtu is set to DL_CSMACD and IEEE8023_MTU.</td>
</tr>
<tr>
<td>_multicast_list()</td>
<td>This function is called to service the DL_HP_MULTICAST_LIST_REQ primitive. In turn this function calls driver dlpi_ioctl() to get the list by passing the command DL_HP_GET_MIB_STATS.</td>
</tr>
<tr>
<td>_ppa_req()</td>
<td>Receipt of DL_HP_PPA_REQ results in calling this function. hw_if_t list is searched for this PPA and the information from hw_if_t is returned.</td>
</tr>
<tr>
<td>_attach()</td>
<td>The information about PPA to be attached is found from hw_if_t list; dlpi_ioctl() is issued to the driver with primitive DL_HP_HDW_INIT. The enet_dlpi_data_t for this stream is updated with network interface information and the stream DLPI state.</td>
</tr>
<tr>
<td>_detach()</td>
<td>Disable all multicasts that were enabled through this stream by issuing dlpi_ioctl()s to the network driver. If promiscuous mode was enabled by this stream then disable it. clean_str_spu_sw_q() is called to clean up any requests in the STREAMS/UX. Finally update the state in enet_dlpi_data to DL_UNATTACHED.</td>
</tr>
<tr>
<td>_bind()</td>
<td>DL_BIND_REQ primitive request indicates to bind a DLSAP to the stream. Protocol kind (LAN_TYPE, LAN_SNAP or LAN_SAP) is determined by SAP value in the request. Driver ioctl request is setup and DL_HP_BIND command is issued to the driver. Once driver bind is successful, dlsap_t is allocated and initialized with protocol type and value of SAP. enet_dlpi_data_t for this stream is updated with this bind details.</td>
</tr>
</tbody>
</table>
STREAMS DLPI Driver

Table 7-6 Message Service Functions

<table>
<thead>
<tr>
<th>Function Name [prefixed by enet_dlpi]</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>_unbind()</td>
<td>Driver ioctl request is setup for unbind and DL_HP_UNBIND is issued. dlsap_t is deallocated and the information in enet_dlpi_data_t about the bind is set to default value.</td>
</tr>
<tr>
<td>_subs_bind()</td>
<td>Upon receipt of DL_SUBS_BIND_REQ, this function is called. Driver ioctl request is setup and DL_HP_BIND command is issued to the driver. If the dl_subs_bind_class is DL_PEER_BIND, then a new dlsap_t is allocated and initialized with protocol type and value of SAP. With DL_HEIRARCHICAL_BIND the dlsap_addr information in enet_dlpi_data_t is updated with bind details.</td>
</tr>
<tr>
<td>_subs_unbind()</td>
<td>For each dlsap_t bound, compare the unbind request SAP, and if there is, match the issue driver DL_HP_UNBIND.</td>
</tr>
<tr>
<td>_xidtest_out()</td>
<td>This function services DL_TEST_REQ, DL_TEST_RES, DL_XID_REQ and DL_XID_RES. It builds the test/xid packet and sends it to the driver using dlpi_output().</td>
</tr>
<tr>
<td>_control()</td>
<td>The primitives serviced by this function are - DL_ENABMULTI_REQ, DL_DISABMULTI_REQ, DL_SET_PHYS_ADDR_REQ, DL_PROMISCON_REQ, DL_PROMISCOFF_REQ and DL_HP_HW_RESET_REQ. The respective ioctl commands are issued to driver via dlpi_ioctl(). If the request didn't complete immediately then this routine sleeps on address of the sleep object of the dlpi_ioctl().</td>
</tr>
<tr>
<td>_status()</td>
<td>Send hw_if-&gt;hdw_state upstream in response to DL_HP_HW_STATUS_REQ request.</td>
</tr>
</tbody>
</table>
Table 7-6  Message Service Functions

<table>
<thead>
<tr>
<th>Function Name [prefixed by enet_dlpi]</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>_getphyaddr()</td>
<td>Requests the network driver ioctl command DL_HP_GET_PHYS_ADDR, which gets the permanent ROM physical address of the network interface, to service DL_PHYS_ADDR_REQ.</td>
</tr>
<tr>
<td>_get_mibstats()</td>
<td>Request the network driver, through the driver ioctl command, to get standard MIBs; this is to service the DLPI primitive DL_GET_STATISTICS_REQ.</td>
</tr>
<tr>
<td>_get_mib_req()</td>
<td>Services MC_GET_MIB_REQ (sys/mci.h). The driver ioctl DL_HP_GET_STATISTICS is issued to get current MIB statistics.</td>
</tr>
<tr>
<td>_set_mib_req()</td>
<td>Services MC_SET_MIB_REQ. The driver ioctl DL_HP_RESET_STATS is issued to reset the MIB statistics.</td>
</tr>
</tbody>
</table>

IOCTL Processing

STREAMS/UX provides the capability for user processes to perform control functions by using ioctl calls on device drivers in a stream. These commands cause the stream head to create an M_IOCTL message that includes the ioctl arguments and to send the message downstream to be received and processed by a device driver. The streams put function call enet_dlpi_process_ioctl() to service M_IOCTL message types. This function consists of a switch block that services various M_IOCTL messages. The IOCTL commands are defined in sys/dlpi_ext.h.

The sample driver implements DLPI_IOC_HDR_INFO, DLPI_IOC_DRIVER_OPTIONS, and DLPI_SET_NOLOOPBACK.

The application sends DLPI an M_IOCTL message with the ioctl command DLPI_IOC_HDR_INFO. The M_IOCTL message block is linked with the M_PROTO message block with DL_UNITDATA_REQ primitive. The LLC header format is built for the specific interface in a new M_DATA message block and is linked to M_PROTO, and the whole complex message is sent back to the application.
Creating Networking Device Drivers

STREAMS DLPI Driver

The ioctl DLPI_IOC_DRIVER_OPTIONS is processed by sending the hw_ift_t information for the request stream.

DLPI_SET_NOLOOPBACK ioctl causes the enet_dlpi_data->nollopback_flg to be set to the value specified in the ioctl parameter.

Transmission of Message blocks

The message block transmission has two paths in the sample implementation. The regular data path uses the DL_UNITDATA_REQ primitive and the fast path. The regular path is defined in the DLPI standards. The fast path uses DLPI_IOC_HDR_INFO ioctl to set up the path and is an HP extension to the DLPI standard.

Regular Data Path

The regular data path message transmission works as follows. The streams put function dlpi_enet_wput() receives the DL_UNITDATA_REQ primitive request from the application to send a message to a destination specified in the unitdata message. dlpi_enet_wput() calls the enet_dlpi_unitdata_out() function to service the request. enet_dlpi_unitdata_out() applies sanity checks for the stream's DLPI state and the request parameters and builds the LLC header. The LLC header message block is linked with the first M_DATA (with DL_UNITDATA_REQ) and proceeds with calling the driver's output routine dlpi_output() in enet_hw_dlpi structure.

Fast Path

For better performance, fast path is used to transmit and receive data. The basic idea of fast path is that the DLPI user sends DLPI ioctl DLPI_IOC_HDR_INFO to set up the fast path on the stream. The DLPI builds an LLC header template and sends it back to the user. Then, for an outbound packet, the user prepends the link header to the data, based on the template of the link header, and sends M_DATA messages to DLPI. DLPI passes this packet to the network driver without going through the process of building the link header. Similarly, for an inbound packet on the fastpath stream, DLPI strips off the LLC header and passes it to the user without building and prepending the DL_UNITDATA_IND primitive to the data.
Reception of Message blocks

The message is received by the `enet_dllpi_intr()` function that was passed to the driver along with the stream queue pointer. The following sanity checks are applied:

- Drop multicast packets for which there is no enabled multicast address.
- If `DL_PROMISC_MULTI` is at promiscuous level and `streams` is in state `DL_UNBOUND`, discard unicast packets.
- If `DL_PROMISC_SAP` then discard packets that are not destined for stream’s network interface.

This function proceeds calling `enet_dllpi_unitdata_in()` or `enet_dllpi_fast_in()`, based on whether fast path is set or not.

`enet_dllpi_unitdata_in()` allocates an MPROTO message block and builds a DL_UNITDATA_IND primitive from the LLC header in the M_DATA message received from the driver. The LLC header is stripped off the M_DATA message, and this block is linked to unitdata message and sent to the application.

The function implemented in `enet_dllpi_fast_in()` was discussed in the “Fast Path” section above.

Summary of DLPI Primitives and IOCTLs

The following table summarizes the DLPI primitives and IOCTLs that have been dealt with in the sample drivers, along with appropriate comments. Note that the processing of most DLPI primitives and IOCTLs involves driver interaction. This is discussed in the next section.

<table>
<thead>
<tr>
<th>DLPI Primitive or IOCTL</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL_INFO_REQ</td>
<td>Information reporting</td>
</tr>
<tr>
<td>DL_INFO_ACK</td>
<td></td>
</tr>
<tr>
<td>DL_ERROR_ACK</td>
<td></td>
</tr>
</tbody>
</table>

Table 7-7 DLPI Primitives and IOCTLs
## DLPI Primitives and IOCTLs

<table>
<thead>
<tr>
<th>DLPI Primitive or IOCTL</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL_ATTACH_REQ</td>
<td>Attach</td>
</tr>
<tr>
<td>DL_DETACH_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_OK_ACK</td>
<td></td>
</tr>
<tr>
<td>DL_ERROR_ACK</td>
<td></td>
</tr>
<tr>
<td>DL_BIND_REQ</td>
<td>Bind</td>
</tr>
<tr>
<td>DL_BIND_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_UNBIND_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_SUBS_UNBIND_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_OK_ACK</td>
<td></td>
</tr>
<tr>
<td>DL_ERROR_ACK</td>
<td></td>
</tr>
<tr>
<td>DL_ENABMULTI_REQ</td>
<td>Other</td>
</tr>
<tr>
<td>DL_DISABMULTI_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_PROMISCON_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_PROMISCOFF_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_PHYS_ADDR_REQ</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_ERROR_ACK</td>
<td></td>
</tr>
<tr>
<td>DL_UNITDATA_REQ</td>
<td>DLPI Ver 2.0 Connection less Data transfer</td>
</tr>
<tr>
<td>DL_UNITDATA_IND</td>
<td></td>
</tr>
<tr>
<td>HP EXTENDED DLPI PRIMITIVES</td>
<td></td>
</tr>
</tbody>
</table>
### Table 7-7 DLPI Primitives and IOCTLs

<table>
<thead>
<tr>
<th>DLPI Primitive or IOCTL</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL_HP_MULTICAST_LIST_REQ</td>
<td>These are HP extensions to DLPI 2.0 and may change. They are defined in sys/dlpi_ext.h</td>
</tr>
<tr>
<td>DL_HP_PPA_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_HP_OK_TO_LOG_SNAP</td>
<td></td>
</tr>
<tr>
<td>DL_HP_HW_RESET_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_HP_HW_STATUS_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_HP_RESET_STATS_REQ</td>
<td></td>
</tr>
<tr>
<td>MC_GET_MIB_REQ</td>
<td>These HP specific IOCTLs and may change. They are defined in sys/mci.h</td>
</tr>
<tr>
<td>MC_SET_MIB_REQ</td>
<td></td>
</tr>
<tr>
<td>HP IOCTLS</td>
<td></td>
</tr>
<tr>
<td>DLPI_SET_NOLOOPBACK</td>
<td>These HP specific IOCTLs and may change. They are defined in sys/dlpi_ext.h</td>
</tr>
<tr>
<td>DLPI_IOC_HDR_INFO</td>
<td></td>
</tr>
<tr>
<td>DLPI_IOC_DRIVER_OPTIONS</td>
<td></td>
</tr>
</tbody>
</table>
WSIO network driver

This section briefly explains the code flow of a network driver. Please refer to Figure 7-4 on page 208 for network driver major functions and their interaction with the DLPI driver. Refer to the sample code for the details.

Data Structures

enet_ift

This structure holds network interface PCI information, register addresses, transmit and receive buffers and descriptors, driver state, and MIB statistics. This structure also embeds an enlan_ift structure that holds generic LAN information pertaining to this interface. The following shows the structure organization.

typedef struct enet_ift {
    enlan_ift  lancift;
    /*****************************************************
    * PCI Configuration information - PCI CONF
    *******************************************************/
    ...  
    ... 
    /*****************************************************
    * PCI Control and Status registers. Each field contains the
    * HPA + offset for the network contlr. registers - DEV REG
    *******************************************************/
    ...  
    ... 
    /*****************************************************
    * Device Specific Section - DEV SPEC
    *******************************************************/
    struct isc_table_type*isc;

enet_srom_t *srom; /* Serial ROM layout*/
ubit32 drv_state; /* Driver state info*/
ubit32 reset_state; /* Driver reset state*/
...
...
/****************************
Transmit Section - TX SECT
***********************************/
et_t *tbr; /* Transmit buffer Ring */
et_td_t *tdr; /* Transmit Descriptor Ring */
...
...
/****************************
Receive Section - RX SECT
***********************************/
et_rd_t *rdr; /* Receive Descriptor Ring */
et_rb_t *rbr; /* Receive buffer Ring */
...
...
/****************************
Full Duplex, speed and Transmit Threshold setting - SETTINGS
*******************************************************************************/
...
...
/****************************
Local Driver Receive Stats - STATS
*******************************************************************************/
rcv_stats_t rstats; /* Receive Statistics*/
/****************************
Local Driver Transmit Stats - STATS
Creating Networking Device Drivers

WSIO network driver

trx_stats_ttstats;/* Transmit Statistics*/
***************************************************************/

Mib Specific Section
***************************************************************/
mib_xEntrymib_xstats;
mib_Dot3StatsEntrydot3_ext_stats;
mib_Dot3CollEntrydot3_ext_coll;
/***************************************************************/

* Misc
***************************************************************/

Table 7-8 enet_ift Data Fields

<table>
<thead>
<tr>
<th>Field Name/Generic Description</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>lancift</td>
<td>Contains generic LAN information</td>
</tr>
<tr>
<td>PCI INFO</td>
<td>Has PCI configuration information</td>
</tr>
<tr>
<td>DEV REG</td>
<td>Fields have Control and Status Register addresses</td>
</tr>
<tr>
<td>DEV SPEC</td>
<td>Device specific information, such as ISC structure, Serial ROM data, driver states, and cable state</td>
</tr>
</tbody>
</table>
enlan_ift

enlan_ift holds generic LAN information for the network interface. The structure is shown below and table explains about fields.

typedef struct{
    enet_hw_ift_t hwift;
    lan_timer_lantimer;
    int ptr_t (*hw_req());
    int (*dma_time());
    /* Status and statistics data area - STATUS & STAT*/
    uint32_t BAD_CONTROL;
    uint32_t UNKNOWN_PROTO;
    uint32_t RXD_XID;
    uint32_t RXD_TEST;
}

<table>
<thead>
<tr>
<th>Field Name/Generic Description</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>TX SECT : tbr, tdr..</td>
<td>This set of fields contain transmit buffers, transmit descriptors, and counters</td>
</tr>
<tr>
<td>RX SECT : rbr,rdr..</td>
<td>This set of fields contain receive buffers, receive descriptors, and counters</td>
</tr>
<tr>
<td>SETTINGS</td>
<td>Full duplex, link speed, selected connection type, and transmit threshold settings</td>
</tr>
<tr>
<td>STATS</td>
<td>Driver local Receiver and Transmitter statistics</td>
</tr>
<tr>
<td>mib_xstats</td>
<td>MIB objects (RFC 1066/1156) and additional counters</td>
</tr>
<tr>
<td>dot3_ext_stats</td>
<td>Extended MIB statistics</td>
</tr>
<tr>
<td>dot3_ext_coll</td>
<td>Extended MIB collisions</td>
</tr>
<tr>
<td>enet_r_lock</td>
<td>Lock for accessing enet_ift</td>
</tr>
</tbody>
</table>
uint32_t RXD_SPECIAL_DROPPED;
short int is_scaninterval;
/* Configuration info */
int num_multicast_addr;
int broadcast_filter;
int multicast_filter;
enlanc_promisc_type_t promiscuous_filter;
int hdw_initialized;
uint8_t mcast[96];
uint32_t mcast_ref_cnt[16];
mib_xEntry *mib_xstats_ptr;
lock_t* enlanc_lock;
}
enlan_ift;

Table 7-9 enlan_ift Data Fields

<table>
<thead>
<tr>
<th>Field Name/Generic Description</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>hwift</td>
<td>Generic Hardware information</td>
</tr>
<tr>
<td>lantimer</td>
<td>DMA/Control timer to track if a DMA or control operation is taking too long</td>
</tr>
<tr>
<td>hw_req()</td>
<td>h/w interface request function pointer</td>
</tr>
<tr>
<td>dma_time()</td>
<td>DMA timeout error handling</td>
</tr>
<tr>
<td>STATUS &amp; STAT</td>
<td>More statistics</td>
</tr>
<tr>
<td>num_multicast_addr</td>
<td>Number of multicast addresses active</td>
</tr>
<tr>
<td>broadcast_filter</td>
<td>Read packet filters</td>
</tr>
<tr>
<td>multicast_filter</td>
<td></td>
</tr>
<tr>
<td>promiscuous_filter</td>
<td></td>
</tr>
<tr>
<td>mcast, mcast_ref_cnt</td>
<td>Multicast addresses and their reference count</td>
</tr>
</tbody>
</table>
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.

```c
struct logged_info{
    int protocol_val[5];
    void (*rint)();
    caddr_t data_ptr;
    uintptr_t lu_protocol_info;
    int flags;
};
```

### Table 7-9 enlan_ift Data Fields

<table>
<thead>
<tr>
<th>Field Name/Generic Description</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>mib_xstats_ptr</td>
<td>MIB object</td>
</tr>
<tr>
<td>enlanc_lock</td>
<td>Lock to access enlanc_ift</td>
</tr>
</tbody>
</table>

### logged_info, logged_link

The following structure is used to link the logged_infos.

```c
struct logged_link{
    ...;
};
```

### Table 7-10 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>rint()</td>
<td>pointer to DLPI input routine</td>
</tr>
<tr>
<td>data_ptr</td>
<td>Driver control block (enet_if)</td>
</tr>
<tr>
<td>lu_protocol_info</td>
<td>queue pointer of the stream which 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 logged_infos.
Creating Networking Device Drivers
WSIO network driver

struct logged_link *next;
struct logged_info log;
}

IOCTLs

The DLPI layer uses the driver ioctl function available in enet_hw_dlpi to communicate the device-dependent DLPI primitives to the device driver for processing. Refer to “Writing a driver_ioctl() Routine” on page 131, Chapter 5, for details about the dlpi_ioctl() function. Essentially the DLPI driver calls this function with a driver control block enet_if, the ioctl command, and the message block with request data. The function pointer dlpi_ioctl is initialized to enet_ioctl() during initialization by the network driver.

The table below summarizes the driver IOCTL commands and the processing by the network driver.

<table>
<thead>
<tr>
<th>IOCTL</th>
<th>Processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL_HP_BIND</td>
<td>This command binds a DLSAP to the stream. Driver logs the bind specifics in logged_info structure and calls enlanc_log_protocol() to add to the list of bound SAPs.</td>
</tr>
<tr>
<td>DL_HP_UNBIND</td>
<td>Driver removes the previously bound DLSAP from the list of logged_info structures using enlanc_remove_protocol() function.</td>
</tr>
<tr>
<td>DL_HP_LOOKUP_PROTO</td>
<td>Driver checks that a DLSAP is bound to the interface. This is done by using function enlanc_lookup().</td>
</tr>
<tr>
<td>IOCTL</td>
<td>Processing</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>DL_HP_ENABLMULTI</td>
<td>Call enlan_media_control() process this request; this function checks for validity of multicast address. Otherwise ADD_MCAST is called to add a multicast address in the list, and this results in calling a chain of enet_hw_req() and enet_ctl_req() to enable the new multicast address in the hardware.</td>
</tr>
<tr>
<td>DL_HP_DISABMULTI</td>
<td>Same processing as above, except that the multicast address is removed.</td>
</tr>
<tr>
<td>DL_HP_PROMISCON</td>
<td>Driver calls enlan_media_control() to enet_hw_req() to enet_ctl_req() to enable promiscuous to 'update' the current promiscuous filter and set the hardware.</td>
</tr>
<tr>
<td>DL_HP_PROMISCOFF</td>
<td>Same as above, except we are updating/removing the promiscuous filter.</td>
</tr>
<tr>
<td>DL_HP_GET_PHYS_ADDR</td>
<td>The functions are called in the following order - enet_hw_req(), enet_ctl_req(), enet_get_stats(). Physical address is returned.</td>
</tr>
<tr>
<td>DL_HP_SET_PHYS_ADDR</td>
<td>Same as above. Local address is copied with new address.</td>
</tr>
<tr>
<td>DL_HP_RESET_STATS</td>
<td>The function called are - enlan_media_control(), enet_hw_req(), enet_ctl_req(), and enet_ext_clearmib() to clear MIB.</td>
</tr>
<tr>
<td>DL_HP_GET_STATISTICS</td>
<td>The functions called are - enet_hw_req(), enet_ctl_req(), and enet_ext_mibstats() to get MIB statistics.</td>
</tr>
</tbody>
</table>
Creating Networking Device Drivers

**WSIO network driver**

### Table 7-11

<table>
<thead>
<tr>
<th>IOCTL</th>
<th>Processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL_HP_HW_RESET</td>
<td>The following functions are called in order - enlanc_media_control(), enet_hw_req(), enet_ctl_req(), and enet_reset() to perform hardware reset.</td>
</tr>
</tbody>
</table>

**Outbound Path**

The enet driver write path starts with `enet_output()` called from the DLPI layer through the `enet_hw_dlpi->dlpi_output` function pointer.

**enet_output()**

This function calls `enet_hw_req()` to handle the write request.

**enet_hw_req()**

All write requests `LAN_REQ_WRITE` and loopback write requests `LAN_REQ_WRITE_L` are processed when the driver state is `ENET_ONLINE`. Otherwise, only loopback write requests are processed, and other write requests are discarded.

Non-loopback unicast packets are transmitted in the fast path by calling `ENET_TRANSMIT_FRAME`. Multicast, Broadcast, self addressed frames, frames < 14 bytes, and frames with buffers > `ENET_MAX_BUF_PER_FRAME` are handled in the slow path by calling `enet_slow_hw_req()`.

Non-write requests are passed on to `enet_ctl_req()`.

**enet_slow_hw_req()**

Non unicast frames are handled in `enet_transmit_complete()`. If the number of buffers is > `ENET_MAX_BUF_PER_FRAME`, an attempt is made to copy all the buffers into one, to use only one transmit descriptor and use 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
Creating Networking Device Drivers
WSIO network driver

frame; if not, the frame is dropped. If transmit descriptors are unavailable, the buffer is queued up for later transmission. Pending transmits are handled in the transmit complete interrupt.

enet_transmit_complete() is called to process transmit complete interrupts. Otherwise, the transmit descriptors are setup 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. enet_transmit_complete() is called to handle the interrupt.

enet_transmit_complete()

enet_process_complete() processes transmit complete interrupts. Call enet_transmit_slow_complete() to process non-unicast frames or setup frames. Transmit error handling is done by calling enet_transmit_error(). If there are frames queued for transmission, call enet_transmit_pending_frames() to restart transmission.

enet_transmit_pending_frames()

While there are frames pending transmission, map the frames, setup the transmit descriptors, and issue transmit poll to the device to transmit the frames.

Inbound Path

The enet read path is on the ICS. enet_isr() is called when the network interface's PCI interrupt is received and enet_receive_frame() is invoked to process received frames.

enet_receive_pkts()

This function is called from the receive interrupt handler. Some sanity checking is done on the received frames to determine if they are good. The message block chain is constructed from receive descriptor. If the driver state is ENET_ONLINE, call enet_process_packet() to process the frame. Otherwise, call enet_process_looper() to process the frame. Replenishing the receive descriptor ring with buffers is done while doing frame receive processing.

enet_process_packet()

This function determines that the frame header is Ethernet or IEEE 802.2 and enlanc_ether_ics() or enlanc_802_2_ics() is called
Creating Networking Device Drivers

WSIO network driver

accordingly.

**enet_process_looper()**

This function processes the loopback packet. The current driver substate determines the action taken. The packet buffer is validated but not used and discarded.

**enlanc_802_2_ics()**

The packet type (802.2 or 802.2 SNAP), protocol kind (LAN_TYPE, LAN_SNAP or LAN_SAP), and protocol value are extracted from the received packet. If the interface supports promiscuous mode and it is set, route the packet to all streams qualified for the set promiscuous level using `enlanc_route_promisc()`. The lookup for logged DLSAPs is `enlanc_lookup()`, and if there is a match, this routine sends the packet to the logged stream (by calling the function registered during the bind). XID and TEST packets are processed in `enet_802_2_test_ctl()`.

**enlanc_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, route the packet to all streams qualified for the set promiscuous level, using `enlanc_route_promisc()`. The lookup for logged DLSAPs is `enlanc_lookup()`, and if there is a match, this routine sends the packet to the logged stream by calling the function registered during the bind.
DLKM drivers

DLKM driver support may be provided in future.
Platform Specifics

Interface drivers are supposed to take care of platform dependencies so that one object can run on any HP platform. The CDIO in the I/O subsystem provides a consistent view of HP-UX platforms to drivers by hiding the platform dependencies as much as possible. Some newer platforms, such as the V Class, have dependencies that require special coding.

V Class

The following brief overview of the V class PCI I/O architecture provides a good background for driver writers porting a driver to V Class Systems.

EPIC is the bridge between the PCI bus and Processors, Memory, and interconnections. Two types of host memory are accessible by an I/O card DMA transaction—non-coherent shared memory on the EPIC bridge and channel-based access to coherent system memory. Multiple channels are available to PCI slots or card functions. Driver instances related to different slots will not share a DMA channel or steal resources from each other. The I/O card cannot access any non-coherent address space beyond EPIC.

The driver model for EPIC expects that all control structures are small and stored in EPIC shared memory. All application data is assumed to be in buffers in coherent system memory. These buffers are read or written as part of DMA stream. Outbound prefetch is initiated when a buffer is mapped for an I/O card’s DMA access. (Since it is in coherent memory, it can be prefetched).

WSIO mapping calls work the same way on V Class platforms as on other platforms. EPIC CDIO (accessed via WSIO) will not reassign an IOVA range until all mappings within the channel have been released. So one must be careful with long term mappings.

The following points are useful while writing PCI network drivers for V class.

- Allocate transmit and receive descriptor memory in shared memory \texttt{wsio\_allocate\_shared\_memory()} so they can be accessed faster.
- Shared memory does not need \texttt{wsio\_map()} since it is already both virtually and physically contiguous.
The code examples below illustrate use of the function `wsio_allocate_shared_memory()`. V class specific enhancements are with `#ifdef V_CLASS` macro and if (is_SPP) {} code blocks.

/* This code illustrates the use of shared memory to allocate a transmitter buffer ring for a V CLASS system network controller. Refer to the sample driver enet.c for more details. Look for `#ifdef V_CLASS` or if(is_SPP()) statements.
*/

```
enet_init (struct isc_table_type *isc) {
    enet_ift_t *enet_iftp;
    size_t size;
    u_long phys_base;
    ...
    ...
    /* size: initialized to the size of enet_iftp->tdr (transmitter descriptor ring) */
    ...
    err = wsio_allocate_shared_memory(isc, size,
        (caddr_t *)&phys_base,
        (caddr_t *)&enet_iftp->tdr,
        0);
    VASSERT(err == SHMEM_OK);
    ...
    ...
}
```
Network Management Support

Hewlett-Packard's implementation of MIBs and the access methods to MIB information from HP-UX 10.00 and previous releases has been monolithic in nature, such that all MIB support was directly done in kernel. This approach forced Hewlett-Packard to constantly change the kernel to incorporate new MIB instrumentation when new links or drivers, either supplied by Hewlett-Packard or a third party, were added.

Hewlett-Packard moved from a single monolithic agent to a variable number of agents, called subagents. Whenever a new driver is added to a system, a user-space subagent specific to this driver is also supplied. This subagent provides the MIB instrumentation needed to access the MIB objects associated with the driver. Figure 7-5 shows the master agent/subagents relationship and partitioning of the subagents. The assumption now is that whoever supplies the new driver will also supply the subagent for that driver.

An SNMP manager only communicates with the master agent, and the master agent sends requests to the appropriate subagent(s). The subagent(s) reply to the master agent, which sends the reply to the SNMP manager.

The new Network Management interface will be user-based, contained completely within a user-space library (libnm.a), and in general will have a one-to-one mapping to the calls provided by the old Network Management interface.
In replacing the `/dev/netman`, the following ioctls will not be available.

- NMIOGET
- NMIOSET
- NMIODEL
- NMIOCRE
- NMPEEK
- NMPOKE
Creating Networking Device Drivers

Network Management Support

Network Management services are to be used by STREAMS-based networking interfaces that provide an ifEntry in the MIB-II ifTable object (see below for ifEntry struct in `sys/mib.h`). In the 4.3 Berkeley based networking stack, the ifTable was directly tied to the global ifnet structure list. When a networking interface registered an ifnet structure via if_attach, an ifIndex value was returned which was to be used in the MIB-II ifEntry object managed by that networking interface. The ifTable was known by the TCP/IP protocol stack and all interfaces to retrieve the ifTable and a specific ifEntry was through the TCP/IP protocol stack. With the movement to a STREAMS-based TCP/IP protocol stack, the global ifnet structure list no longer exists and, therefore, the global ifTable management no longer exists.

Even though in the STREAMS based networking environment the ifTable is not globally managed, each ifEntry in the ifTable must have a unique ifIndex value so the ifTable can be created. Therefore, the ifIndex values must be globally managed. Along with managing the ifIndex values, the MIB-II ifNumber object must also be managed. The Network Management services described next are for retrieving and returning a unique ifIndex value.

```c
u_int32 get_nmid
```

Allocates a system-unique ifIndex value for use in the MIB-II ifEntry object. Any kernel entity that required an entry in the ifTable should use this service for retrieving the value of the ifIndex field.

```
u_int32 get_nmid() > 0 indicates that the call succeeded and the value returned is the ifIndex value.
<=0 indicates that the request failed to allocate an ifIndex value.
```

Example code in enet driver:
```
enet_iftp->lanclft.hwift.nm_id = get_nmid();
```

```c
u_int32 return_nmid
```

Return a previously assigned ifIndex to the pool of available ifIndex values. This network management service should be called by all kernel entities that own an ifIndex value before it is unloaded from the system.

```
ifIndex: ifIndex value to be returned to the pool of available ifIndex values.
```
<0 Indicates that the ifIndex value being returned was not the previously assigned ifIndex value.

>=0 Indicates the ifIndex was successfully returned to the pool.

In sys/mib.h, mib_ifEntry is defined as follow:

```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 ifnOctets;
    counter ifnUcastPkts;
    counter ifnNUcastPkts;
    counter ifnDiscards;
    counter ifnErrors;
    counter ifnUnknownProtos;
    counter ifOutOctests;
    counter ifOutNUcastPkts;
    counter ifOutDiscards;
    counter ifOutErrors;
    gauge ifOutQlen;
    int ifSpecific;
} mib_ifEntry;
```

The device driver's job is to fill out the fields in struct mib_ifEntry above in the appropriate place so any application can retrieve information for
Creating Networking Device Drivers

Network Management Support

using the Network Management Support services interface.
Configuring a Networking Driver Through SAM

NOTE

SAM does not support native DLPI at this point. This option will be considered in the future.
Creating Networking Device Drivers

Network Monitoring Commands: driveradmin, driverlinkloop and lanscan

The lanscan command that comes bundled with HP-UX works with third-party device drivers. However, the default lanadmin and linkloop commands are not supported on third-party drivers. It is the responsibility of the driver developer to provide these utilities. The following sections document sample utilities, enetlinkloop and enetadmin. A framework for these utilities is presented in Appendix F for the benefit of driver developers.

It is recommended that the utilities be named driveradmin and driverlinkloop to avoid conflict with the default HP-supplied commands. Also it is recommended that these third party utilities be located in the /opt directory on the target machine.

The driveradmin and lanscan commands use shared libraries to display certain interface-specific network data. Developers are responsible for writing these shared libraries if they want driveradmin and lanscan to work with their new device drivers.

Examples of shared libraries that allow driveradmin and lanscan to work with the enet driver and skeleton code for driveradmin and driverlinkloop are available in Appendix D, “Shared Library Examples for the driveradmin and lanscan Commands.”

driveradmin command

The driveradmin command allows the user to perform various administrative tasks on a specified LAN interface. To perform most network administrative tasks, the driveradmin command executes the same basic program regardless of the interface. The task of displaying the interface statistics, which can vary from interface to interface, has been put into shared libraries. One shared library is available for each networking driver to work with driveradmin.

Invoking a Shared Library to Display Statistics

The shared library is invoked by driveradmin when the user selects the 3rd party option to input a third party's special device filename. Then the user selects the Display command in the LAN Interface Test Mode menu. The shared library invoked by the Display command is determined by...
the Physical Point of Attachment (PPA) that has been selected either implicitly by default or explicitly by the user.

The `driveradmin` routine then completes the steps below:

1. `driveradmin` implicitly selects a PPA by getting the first element of the PPA list. The first element becomes the default PPA.

2. After getting the PPA, `driveradmin` attaches to it. By attaching to the PPA, `driveradmin` can get the driver name that was returned in the attach routine.

   The name of the driver returned in the attach originates from the string that is stored by the driver at initialization time in the name element of the `hw_ift` structure. See “hw_ift_t Structure Description and Initialization” on page 184 for a description of this structure.

3. `driveradmin` uses this driver name to determine the name of the shared library file to access by doing a `shl_load()` of a file that has been named, in the following form:

   `/usr/lib/lanadmin/libdsdriver_name.sl`

   where `driver_name` is the string stored in the name element of the `hw_ift` structure. Every other part of the full path name above is hardcoded in `driveradmin`.

4. After loading the shared library, `driveradmin` again uses the driver name to determine the name of the shared library function to use, by doing a `shl_findsym()` of a function name with the following form:

   `dsdriver_name`

   where `driver_name` is as described above. The `ds` stands for “display statistics.”

5. `driveradmin` then uses the handle returned by `shl_findsym()` to invoke the shared library.

Right after the user selects the Display command of the LAN Interface Test Mode menu and just before invoking the shared library function, `driveradmin` displays the status display title, the date and time, and the first line of the statistics, which is always the PPA. This output resembles the following:

```
LAN INTERFACE STATUS DISPLAY
Tue, Jun 1, 1999  10:47:37
PPA Number = 1
```
Creating Networking Device Drivers

Network Monitoring Commands: driveradmin, driverlinkloop and lanscan

Arguments Passed to Shared Library Functions

driveradmin passes the following arguments into the shared library:

- **int fd**  
  File descriptor of third party’s special device filename used by the shared library function to get the statistics.

- **int cur_ppa**  
  PPA of interface whose statistics are to be displayed.

- **int termlines**  
  Number of terminal lines in the current screen/window; typically used by a shared library function to determine whether the number of statistics being displayed is greater than the screen length.

Writing a driveradmin Shared Library

Two requirements must be met for any existing or new shared library function written specifically to display the interface statistics:

- The shared library function must be named dsdriver_name (with driver_name as the string stored in name in the hw_if structure)

There are no restrictions on what the shared library function can be written to do. For ease of use and consistency with other Hewlett-Packard networking data outputs, each shared library should be written to make the statistics display emulate the statistics displays of existing Hewlett-Packard shared libraries. To promote such consistency for all systems, driveradmin always displays the PPA as the first line of the statistics display.

Defining the Statistics

The statistics that the Hewlett-Packard LAN drivers maintain and that the Hewlett-Packard shared libraries display are the MIB-II statistics defined in RFC 1213. These statistics are common to all Hewlett-Packard LAN links, Ethernet, Token Ring, FDDI, and Fibre Channel. In addition, most Hewlett-Packard shared libraries and most Hewlett-Packard LAN drivers maintain the link-specific MIB statistics. For example, Hewlett-Packard Ethernet/802.3 drivers maintain the Ethernet-like MIB statistics defined in RFC 1398.

Localizing Output Messages

All outputs from shared libraries should be localized. The shared libraries should use the Hewlett-Packard Native Language Support
Creating Networking Device Drivers

Network Monitoring Commands: driveradmin, driverlinkloop and lanscan


Example for Writing a Shared Library

The dsenet.c file provides an example of how to write a shared library. The actual getting and displaying of interface statistics is described in the sample shared library, dsenet(), which is the actual source code for libdsenet.sl, shown in Appendix D, “Shared Library Examples for the driveradmin and lanscan Commands.”

Shared Library Message Catalog

The shared library function, dsenet(), first opens its message catalog. This catalog file is accessed only by its shared library and could be named anything and put any place. To avoid confusion, however, you should conform to Hewlett-Packard conventions for the naming and placement of message catalog files. Each Hewlett-Packard shared library has its own message catalog file in:

/usr/lib/nls/C/dsdriver_name.cat

Getting the Interface Statistics

The dsenet() function uses the DL_GET_STATISTICS_REQ primitive to request interface statistics from the enet driver. The function expects to receive a DL_GET_STATISTICS_ACK primitive that contains the requested statistics. You can, alternatively, use the source of the dsenet() function as an example.

Displaying the Interface Statistics

If the driver maintains RFC 1213 MIB II statistics, the shared library can use the code in the dsenet() function that displays these statistics. If the driver also maintains interface specific statistics, the shared library should display a “Continue” message after displaying the RFC 1213 statistics and wait for a key to be pressed before displaying them. dsenet() functions in this manner.

lanscan Command

The lanscan command displays information about each of the LAN links on the system. lanscan can get access programmatically to all information to be displayed except the encapsulation method. To
determine the encapsulation method, lanscan must make a request to the shared library. There is one shared library for each networking driver that is to work with lanscan.

Displaying Encapsulation Methods

A lanscan shared library can display the encapsulation methods supported by an interface. The shared libraries are invoked by lanscan when the user selects the “-v” (verbose) option on the command line. Since lanscan displays information about all LAN interfaces on the system, a different shared library is invoked for each interface. lanscan traverses the hw_ift linked list to find out what LAN interfaces are configured and what information is to be displayed. See “hw_ift_t Structure Description and Initialization” on page 184 for more information on the hw_ift structure.

When “-v” is selected:

1. lanscan gets the driver name out of the name element of the hw_ift structure to find out the name of the shared library file to access.
2. lanscan does a shl_load() of the file with the following form:
   /
   /usr/lib/lanscan/libpe
   /driver_name
   /sl
   where driver_name is the string stored in the name element of the hw_ift structure. Every other part of the full pathname above is hardcoded in lanscan.
3. After loading in the shared library, lanscan again uses the driver name to find the name of the shared library function to use by executing shl_findsym() of a function name with the following form:
   pe
   driver_name
   where driver_name is as just described with pe standing for “print encapsulation.”
4. lanscan then uses the handle returned by shl_findsym() to invoke the shared library.

Argument Passed to the Shared Library

lanscan passes the following argument into the shared library:

hw_ift_t *hwift;   Pointer to the hw_ift structure for the interface whose information is being displayed by lanscan.
Recommendations for the lanscan

Shared Library Function

The shared library function should start displaying the encapsulation methods at the point where the cursor currently is located. It should not output any spaces, tabs, or line feeds. The shared library function has columns 43 (column count starting from 0) through 80 with which to display all the supported encapsulation methods. The shared library function should not output any spaces, tabs, or line feeds after displaying the encapsulation methods.

Shared library outputs should always be localized. That is, the Native Language Support (NLS) message catalogs should be used. Refer to the HP Native Language Support: User’s Guide for further information.

Shared Library Message Catalog

The peenet() shared library function first opens its message catalog. Each shared library has its own message catalog file in:
/usr/lib/nls/C/pe<driver_name>.cat

NOTE

Use this path and file name coding to avoid confusion and to conform to Hewlett-Packard shared libraries and other conventions.

Encapsulation Methods Support

To discover the checking and displaying of supported encapsulation methods, refer to the sample shared library peenet(), which is source code for libpeenet.sl.

The llc_flags element of the hw_ift structure for a given interface tells which encapsulation methods are supported by the driver.

The following example from /usr/include/sio/lan_dlpikrn.h shows the presentation of bit definitions:

/* LLC Encapsulation Types */
#define IEEE0x01/* IEEE 8022*/
#define HP_EXT_IEEE0x02/* HP Extended IEEE 8022*/
#define SNAP0x04/* IEEE SNAP*/
#define ETHERTYPE0x08/* Ethernet*/
#define NOVELL0x10/* Ethernet */
Creating Networking Device Drivers

Network Monitoring Commands: driveradmin, driverlinkloop and lanscan

**driverlinkloop Command**

The `driverlinkloop` command uses IEEE 802.2 link-level test frames (TEST path) to check connectivity within a local area network. `driverlinkloop` explicitly gets a PPA from the input command line or implicitly selects a PPA by getting the first element of the PPA list and attaches to this PPA via HP-DLPI as default. Unless users specify the third party's option in the command line to input a third party DLPI stream driver's filename, `driverlinkloop` will attach to the picked PPA via this third party's DLPI stream driver. After attachment, the `driverlinkloop` routine will use DL_TEST_REQ primitive to “ping” peer data link providers to test the data-transfer path.

Example:

```
enetlinkloop -i 1 -3 /dev/dlpi_enet 0x0060B07EAAFD
```
Network Tracing and Logging Support for Troubleshooting

This section describes and explains the use of HP-UX network tracing and logging facilities.

To aid in troubleshooting network problems, support for network troubleshooting must address several trends:

- The complexity of network systems is increasing.
- The number of protocols and standards is large and continues to grow.
- The number of possible combinations of services and applications created and used on a network is increasing.
- The troubleshooter is usually far removed from those who understand the network, products, and systems best.

Troubleshooters need knowledgeable support tools to address this complexity and difficulty. Support tools must provide as much information as possible about when and where problems occur. The network code must provide the troubleshooter with failure occurrence, cause, and suggested repair information.

HP-UX network tracing and logging facilities are tools for capturing network events and packets in a log for analysis to support troubleshooting. (Sometimes special diagnostic and test tools must also be used; for example, network traffic analyzers, interpretability tests, and other such aids.)

HP-UX network tracing and logging facilities permit subsystems to record events in a central location for subsequent processing. That information can then be provided to customers and support personnel to audit network activity and troubleshoot network problems.

Introductory Overview of HP-UX Tracing and Logging

HP-UX network tracing and logging facilities provide the following general features:

- A mechanism for recording log events and trace data
Creating Networking Device Drivers

Network Tracing and Logging Support for Troubleshooting

- A facility for determining what information to capture
- A mechanism for selecting and formatting the recorded information
- A set of user-interface commands that:
  - Configure, start, and stop the trace and log services.
  - Format captured messages.

These commands and the other HP-UX network tracing and logging facilities (files, subroutines, etc.) discussed in the following sections, provide a programmatic interface that allows user routines and kernel routines to access the services.

Figure 7-6, “Network Tracing and Logging Elements and Data Flow I,” shows the data flow among the following elements of the HP-UX trace and log system:

- `nettlconf` command
  - `nettlgen.conf` subsystem configuration database (for the following three commands)
  - `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
These elements are explained in the following sections of this chapter.
Creating Networking Device Drivers

Network Tracing and Logging Support for Troubleshooting

**nettlgen.conf(4)**

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.

---

**NOTE**

You must obtain this subsystem ID from Hewlett-Packard (see “Assign Subsystem ID” on page 258.

---

**nettlconf(1M)**

The `nettlconf` command creates and updates the database file `/etc/nettlgen.conf`, the file used to configure each subsystem. This database file controls the behavior of the `nettl` and `netfmt` commands for tracing, logging, and formatting (trace/log) messages. See `nettlconf(1M)`, `nettl(1M)`, and `netfmt(1M)`.

Information such as the subsystem name, library name, and subformatter function are given to the `nettlconf` command, which stores them in the `/etc/nettlgen.conf` configuration file. This command is used in the configure script of the subsystem module during a system install/update time to integrate the subsystem into the trace and log tool. Subsystems use the `nettlconf` command to store a description of themselves in the `nettlgen.conf` database file—typically performed only once, at product installation time.

**nettl(1M)**

This command uses the subsystem information to create subsystem management tables in shared memory and in the kernel, and 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. `nettl` initializes the ktl driver, also called `netdiag1`, and `nettl` starts up the `nktl_daemon` and `ntl_reader` daemons. See the manpage for more detailed information.

**netfmt(1M)**

This command formats binary trace and log data into readable ASCII
Creating Networking Device Drivers

Network Tracing and Logging Support for Troubleshooting

text as described below. Post-filtering of the data is controlled through this command.

1. `netfmt` uses subsystem configuration information to identify shared libraries, provided by subsystems, that contain functions to parse subsystem filters and format subsystem data.

2. `netfmt` dynamically loads all shared libraries and finds the functions each time it is executed.

3. `netfmt` calls the functions of subsystems for which it has data.

4. `netfmt` parses the filter file if it is present. The file is sorted according to the first field, the subsystem name, in the filter file.

5. For each subsystem referenced in the file, the `subsys_N_get_options()` function for that subsystem is called with the filter data. The `subsys_N_get_options()` function is responsible for interpreting and storing the filter data.

6. `netfmt` reads the input file. For each record found it calls the corresponding `subsys_N_format()` function to format the record.

The subsystem will not format the record if the values in the record match the values specified in the filters. The subsystem should format the record according to the format options specified, for example, nice, terse, and raw. See the `netfmt(1M)` manpage for more detailed information.

Using HP-UX Logging and Tracing for Troubleshooting Support

The following guidelines may help developers remain customer-oriented in designing tracing and logging facilities for solving the problems of their troubleshooter clients:

- Log only what is needed to solve problems.
- Record all information to diagnose the problem in the log.
- Provide a hex dump to the troubleshooter only as the last resort.
- Make each product do as much self-diagnosis and repair as possible, and do it quietly. Notify the end user only when intervention is required or requested.
- Give the customer what is needed to solve the customer's problems,
Creating Networking Device Drivers

Network Tracing and Logging Support for Troubleshooting

not the developer’s problems.

The following information will help you set up tracing and logging to support troubleshooting:

• Assign Subsystem ID
• Classify Trace Data
• Format Trace Data
• Classify Log Data
• What and When to Log

Assign Subsystem ID

Each networking product requires its own unique subsystem ID number, which must be assigned by the Hewlett-Packard OpenConnect Team.

To do so, you must email a request for a unique subsystem ID for your product to Hewlett-Packard at nettl_support@india.hp.com. In the message, you should identify a suggested interface subsystem name for your product. You will be assigned this name if it is not already planned for or in use by Hewlett-Packard products. Check /usr/include/sys/subsys_id.h in your system prior to selecting a name. Do not request names such as lan, lo, ni, X25_, and others that are already assigned.

Your 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

You must use the file you receive from Hewlett-Packard as /usr/include/sys/subsys_id.h in your HP-UX device driver development system when you compile your networking device driver.

Classify Trace Data

Tracing can capture or make snapshots of loopback or header information, as well as inbound and outbound packets going through the network. The main purpose of tracing is to analyze networking problems
discovered in either a log error message or in the failure of a networking operation to complete successfully. Tracing follows or records normal events and abnormal events alike and is typically used on events that occur frequently, such as connections opening and closing, or retransmitted data.

Trace kinds are defined as:

- **PDU** Inbound and outbound Protocol Data Units (including header and data).
- **Header** Inbound and outbound protocol headers.
- **Loopback** Trace of packets emanating and returning to the same system.
- **Procedure** Trace of entry and exit from all procedures.
- **Error** Invalid state transitions, invalid protocol data units, bad headers, resource errors, system call errors, and protocol violations. Distinguishing when to use an error trace or an error log can be difficult. In some cases, you may want both. The tracing and logging utility goes to different files, and locating and synchronizing the entries between the two files may be too difficult. Having both an error log and error trace helps to synchronize the two files. Sometimes other log messages are also recorded in the trace file when tracing is enabled.
- **State** Protocol states or connection states, not limited to entry and exit from a layer or procedure. Use this trace kind when recording information about normal state transitions.
- **Connection** Information about connections as they are made and destroyed.
- **Logging Trace** Special kind of trace that contains a log message. This trace kind will help the troubleshooter locate and synchronize logging and tracing output.

**Format Trace Data**

Troubleshooters should trace both incoming and outgoing data through the stack. The trace records from different processes should be threaded together to form a complete record of the path the PDU takes going from...
the user application out the wire, and vice versa.

Refer to the following guidelines when implementing your tracing routines.

- Each process should trace incoming and outgoing data from both top and bottom. Alternatively, each protocol could trace only its incoming and outgoing headers.
- A subformatter for a process's trace information must be provided by the implementer of the process.
- The subformatter formats only the data for which that process is responsible. For example, if the X.25 driver sends a trace record, it decodes only the X.25 portion of the PDU, leaving the rest for the process above it to decode. Likewise, OTS decodes only the Network, Transport, and Session layer portions, leaving the upper layers to the application processes.

Classify Log Data

Logging is a way of capturing and recording specific network activities and infrequent significant network events, such as state changes, errors, and connection establishment. The main purpose of logging is to inform the system operator about these significant events and to make a permanent record for later interrogation. Typical log messages are about errors (catastrophic, recoverable and non-recoverable), warnings (major and minor), or system-wide information (such as changes to configuration or operation).

Logged events are considered in the following classes:

Disaster: Signals that the software detected a severe and irrecoverable error condition that typically affects multiple user applications or connections and may jeopardize system integrity. For example, the condition may cause a system crash or corrupt a system table. Another example is when a condition implies that an action generated by one process may damage other processes.

Error: Signals an event or condition that, while not affecting the overall subsystem or network operation, causes an application program to fail or complete in an error condition. Indicates that the system is not performing as it should, but the underlying networking subsystem...
was able to recover. For example, an error class condition occurs when a process must abort its operation or take extra steps to recover a certain state.

**Warning**
Indicates an abnormal event, but not necessarily a networking problem; event possibly caused by a subsystem problem. Examples include possible pointer alignment problems or data being accessed that has not been initialized.

**Informative**
Describes infrequent operations and current system activities, such as protocol module initiation and termination sufficiently important to post.

### What and When to Log

The most important part of logged messages is the ASCII string describing the event, which is the first item a system operator might see on the system console following an event in the network operation. Deciding what to log and when it should be logged often involves trade-offs in terms of usability, performance, schedule constraints, and management and peer pressure. Besides the items outlined in the preceding tracing or logging sections, some general guidelines include:

- If an event results or causes the product or system to be unusable by all users, it should be logged as a Disaster class log message.
- If an event affects a single application, it should be logged as an Error class log message.
- If an event may cause an error or disaster in the future, or cause performance degradations, it should be logged as a Warning class log message.
- If an event occurs infrequently, and is something the user may want to know about but will not cause future problems, it should be logged as an Informative class log message.
- If an event occurs frequently, or with regularity, it is probably not appropriate to log it, but to trace it instead. Don't use Informative log messages in place of tracing.
- Don't log “Me Too!” messages in Error or Disaster class. These are events which occur in response to an error or disaster event in another place, but aren't themselves a disaster or error. “Me Too!” messages are characterized as providing no additional information to
solve the problem at hand.

- Do not acquire a new log instance if one is already available for the particular event thread you are on. (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 troubleshooter should be able to know what happened, what caused it, and how to proceed to fix the problem on the basis of your log message alone.

- State the exact commands to use to perform the recommended actions.

- If the explanation is too long to include in the log message, refer troubleshooters to the appropriate manual to take further steps or gain more knowledge about the problem.

- Encapsulate logging calls in functions or macros.

- Adhere to the logging error classes (Disaster, Error, Warning, and Informative) to promote uniformity in the troubleshooting process you recommend and to facilitate communication with HP support groups.

- Restrict logged information to only a few, well-defined types; for example, event number, a bounded array, or a string.

- Identify error recovery procedures for Disaster- and Error-class events.

- Devote most of your effort to understanding and documenting the procedures listed above. Only after completing error recovery procedures for these events should you focus on Informative and Warning class events, and then only if they would actually be useful.

### Passing Data to HP-UX Tracing/Logging

Kernel subsystems that use the trace and log services must include the following in their source files and makefiles.

```c
#include <net_diag.h>   Contains macro calls to check that tracing and logging is enabled for the subsystem.
#include <subsys_id.h> Contains subsystem information and definitions for log classes and trace kinds.
```

The function calls for kernel subsystems are provided for capturing trace
Creating Networking Device Drivers

Network Tracing and Logging Support for Troubleshooting

and log data.

KTRC_CK()

This routine is used to trace on an all interface-device basis.

This macro allows the calling process to verify if tracing is enabled for the current subsystem. The returned value is 1 (one) if tracing is enabled. It is defined as:

KTRC_CK(subsys_id, trace_kind)

subsys_id   Unique subsystem ID of the calling subsystem. The number is assigned by Hewlett-Packard; see “Assign Subsystem ID” on page 258.

trace_kind   Defines the kind of trace; available kinds are defined in the subsys_id.h header file and are described in more detail in “Classify Trace Data” on page 258, as follows:

<table>
<thead>
<tr>
<th>Trace Kind</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDR_IN_BIT</td>
<td>Inbound header tracing mask</td>
</tr>
<tr>
<td>HDR_OUT_BIT</td>
<td>Outbound header tracing mask</td>
</tr>
<tr>
<td>PDU_IN_BIT</td>
<td>Inbound PDU tracing mask</td>
</tr>
<tr>
<td>PDU_OUT_BIT</td>
<td>Outbound PDU tracing mask</td>
</tr>
<tr>
<td>PROCEDURE_TRACE_BIT</td>
<td>Procedure entry/exit trace</td>
</tr>
<tr>
<td>ERROR_TRACE_BIT</td>
<td>Error tracing mask</td>
</tr>
<tr>
<td>LOGGING_TRACE_BIT</td>
<td>Log call tracing mask</td>
</tr>
<tr>
<td>LOOP_BACK_BIT</td>
<td>For loopback</td>
</tr>
<tr>
<td>PTOP_BIT</td>
<td>For point to point</td>
</tr>
</tbody>
</table>

NOTE

There are some alias or redefine the trace_kind functions above in the net_diag.h header file:

```c
#define TR_LINK_LOOP LOOP_BACK_BIT
#define TR_LINK_INBOUND PDU_IN_BIT
#define TR_LINK_OUTBOUND PDU_OUT_BIT
```

For example, a hypothetical driver named enet.c might use this macro.
Creating Networking Device Drivers
Network Tracing and Logging Support for Troubleshooting

as follows:
if (KTRC_CK(ENET_ID, TR_LINK_INBOUND))
{
    ktrc_write(...);
}

**ktrc_write()**

This routine is used to send trace messages to the kernel trace and log facility.

Prefiltering is done at the time of the trace call, and unwanted messages are dropped. This routine always returns a success indicator of 0 and is defined as:

```c
ktrc_write ( int subsys_id,
            int kind,
            int path_id,
            int device_id,
            caddr_ttl_packet,
            int ttl_packet_cnt)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>subsys_id</td>
<td>Unique subsystem ID of the calling subsystem (number assigned by Hewlett-Packard; see the “Assign Subsystem ID” on page 258 section).</td>
</tr>
<tr>
<td>trace_kind</td>
<td>Defines the kind of trace. All kinds are defined in the header file subsys_id.h. The following is the defined trace kind values (see “Classify Trace Data” on page 258). They can be OR’ed to produce the combination of trace kinds.</td>
</tr>
<tr>
<td>HDR_IN_BIT</td>
<td>Inbound header tracing mask</td>
</tr>
<tr>
<td>HDR_OUT_BIT</td>
<td>Outbound header tracing mask</td>
</tr>
<tr>
<td>PDU_IN_BIT</td>
<td>Inbound PDU tracing mask</td>
</tr>
<tr>
<td>PDU_OUT_BIT</td>
<td>Outbound PDU tracing mask</td>
</tr>
<tr>
<td>PROCEDURE_TRACE_BIT</td>
<td>Procedure entry/exit trace</td>
</tr>
<tr>
<td>STATE_TRACE_BIT</td>
<td>State machine tracing mask</td>
</tr>
<tr>
<td>ERROR_TRACE_BIT</td>
<td>Error tracing mask</td>
</tr>
</tbody>
</table>
LOGGING_TRACE_BIT
  Log call tracing mask

LOOP_BACK_BIT
  For loopback

PTOP_BIT   For point to point

path_id   Connection path on the host. If this is a nonapplicable
  parameter, pass in –1.

device_id   Device ID number (for example, if_unit) of the calling
  subsystem message. If this is a nonapplicable
  parameter, pass in –1.

tl_packet   Either a pointer to an mbuf chain or a pointer to a set of
  iovec structures as determined by tl_packet_cnt.
  The calling routine will pass a pointer (cast to caddr_t)
  to an mbuf chain or an iovec structure. This structure
  is immediately copied into an mbuf chain owned by
  tracing and logging facilities. Therefore, it is not
  necessary for the calling routine to copy the data and
  then pass a pointer to it.

tl_packet_cnt   If –1, then tl_packet points to an mbuf chain. If
  greater than 0, this is the number of the iovec
  structure to which tl_packet points.

As with logging, developers should encapsulate tracing calls in functions
or macros. The code scenario in the following section shows a typical use
of tracing calls.

**Tracing Code Sample**  The following example shows a trace of an
outbound packet whose various parts are located in distinct memory
locations. The trace uses the vectored data capability of the
ktrc_write() call. The same could be accomplished using an mbuf chain
as well.

```c
1  #include "../h/netdiag1.h"
2  #include "../h/net_diag.h"
3  #include "../h/subsys_id.h"
4  #include "../h/uio.h"
5  ...
6
7  #define MAX_BUF 3  /* any number of vectors are allowed */
8  #define TRACE  0
```
Creating Networking Device Drivers

Network Tracing and Logging Support for Troubleshooting

```c
#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
         * 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,
```

Chapter 7
Creating Networking Device Drivers
Network Tracing and Logging Support for Troubleshooting

51    kind,
52    path_id,
53    device_id,
54    &tl_buf,
55    tl_buf_cnt);
56  }
57
58 return(0);
59 }

KLOG_CK()

This macro allows the calling process to find out whether logging is enabled for the current subsystem.
The returned value is 1 (one) if logging is enabled. It is defined as:

KLOG_CK(subsys_id, log_class)

subsys_id    Unique ID number (assigned by Hewlett-Packard) of the calling subsystem.

log_class    Defines the classification of event. All classes are defined in the header file subsys_id.h (see also “Classify Trace Data” on page 258). Four classes are defined for logging messages:

Informative   Normal messages only
Warning       Warning messages
Error         Error condition messages
Disaster      Critical error messages

kget_log_instance()

This call accepts no parameters but returns a unique log instance value. The log instance helps thread log messages together so that the user can easily identify the messages that result from the same event. A change in the log instance means that a new event is being logged.

The log instance value should be passed between subsystems through their interface parameter list so each module can have access to it. If a module encounters a unique event, it will obtain a log instance value. Otherwise, the module should use the current log instance value it was passed without calling kget_log_instance(). (See also klogg_write() for information on log instance values.)
Creating Networking Device Drivers

Network Tracing and Logging Support for Troubleshooting

**klogg_write()**

This routine is used to send log messages to the kernel trace and log facility. Prefiltering is done at the time of the log call, and unwanted messages are dropped. This routine always returns a success of 0 and is defined as:

```c
klogg_write(int subsys_id,
            int class,
            int device_id,
            int log_instance,
            caddr_t tl_packet,
            int tl_packet_cnt)
```

- **subsys_id**
  Unique ID (number assigned by Hewlett-Packard) of the calling subsystem.

- **class**
  Defines the classification of event. All classes are defined in the header file `subsys_id.h` (see also “Classify Trace Data” on page 258). Four classes are defined for logging messages:
  - Informative: Normal messages only
  - Warning: Warning messages
  - Error: Error condition messages
  - Disaster: Critical error messages

- **device_id**
  Device ID number (for example, if_unit) of the calling subsystem message. If this is a nonapplicable parameter, pass in -1.

- **log_instance**
  Unique static number used to identify the thread of events attending an interface. If this is a nonapplicable parameter, pass in -1.

- **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, then `tl_packet` points to an `mbuf` chain. If the

---

Chapter 7
value is greater than 0, it is the number of iovec structure (as defined in uio.h) that tl_packet points to.

Logging Code Sample

The following scenarios may help you to better understand how to use the intrinsic calls of HP-UX logging facilities. These are typical fragments of code that a subsystem might include to perform logging calls.

```c
#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 for call to KLOG_CK()
     */
```
subsys_id = MY_SUBSYS_ID_NUMBER; /* assigned by HP */
class = DISASTER;
device_id = -1; /* -1 means not applicable */

if (KLOG_CK(subsys_id, class))
{
    /*
     * Logging is enabled for this subsystem
     * and class combination.
     */

    if (log_instance == 0)
    {
        /*
         * There was no previous log instance
         * associated with this event. This is
         * the first module to encounter the
         * problem, so it gets the log instance.
         * Log instance should be available to
         * all modules in the subsystem and to
         * other subsystems.
         */

        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. This
     * data structure is entirely up to the
     * local developer to design. The
     * subformatter for this subsystem must
     * be able to decode the data structure,
     * but other than that there are no
     * restrictions on what gets passed. The
     * local developer may choose to use a
     * single mbuf chain to hold all the
     * event information, or pass a vectored
     * buffer to the klogg_write() call to
     */
Creating Networking Device Drivers

Network Tracing and Logging Support for Troubleshooting

The following sections detail facilities and network device driver developer responsibilities for formatting trace/log data output. Some sections provide code and output examples. Appendix C, “How to Design a Networking Trace/Log Subformatter and a Sample Subformatter,” shows a generic style subformatter that can handle the preceding logging and tracing examples using the basic calls of netfmt.

Formatting Networking Trace/Log Messages

The netfmt formatter is the facility that presents trace and log information in human-readable form. It comprises two distinct pieces:

- **Subformatter**: a function provided by the subsystem to interpret the messages and produce human-readable form output.
- **Formatter core**: responsible for file handling, global filtering, and dispatching messages to the appropriate subsystem subformatter.
This `netfmt` formatter is a filter that transforms the binary trace or log output data file into human-readable form. The party that puts the trace and log calls into the code must also provide a means of formatting/interpreting the data passed in those calls. Similarly essential is ensuring the loading of all potentially useful subformatter libraries.

`netfmt` uses the subsystems configuration information to identify the shared libraries (provided by the subsystems) which contain functions to parse subsystem filters and format subsystem data. `netfmt` dynamically loads all shared libraries and finds the functions each time it is executed. `netfmt` calls the functions of a subsystem only when it has data belonging to the subsystem.

The formatter filter handles or discards records based on data in each message header; such filtering is described in the `netfmt(1M)` manpage. (See also “netfmt(1M)” in “Introductory Overview of HP-UX Tracing and Logging” on page 253.)

The formatter and subformatters determine filtering and formatting options by processing an auxiliary file referred to as the options file. This options file filtering feature is available to any subsystem.

During filtering, the formatter checks the message to make sure it contains good information. If the formatter finds a corrupted message header, an unknown subsystem, a message that is too long to handle, and so forth, it prints an informative message, formats the message header, and discards the remainder of the data. It then continues with the rest of the file.

The formatter provides 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. These functions are discussed in following sections, with recommendations for usage. Figure 7-6, “Network Tracing and Logging Elements and Data Flow I,” identifies important tracing/logging subroutines and shows their relationship to tracing/logging facilities shown in Figure 7-5, “Master Agent/Subagents Relationship.” You can add additional functions, if necessary.

Appendix E in this guide addresses what the formatted data could look like, and gives further information on this subject. To recap, the formatter provides the following features:

- Loads subformatter shared libraries
Creating Networking Device Drivers
Network Tracing and Logging Support for Troubleshooting

• Processes filtering and formatting options
• Handles binary input
• Handles global filtering
• Processes unknown or bogus data
• Dispatches data to correct subformatter
• Handles common subformatter tasks

How to design a Subformatter and related issues are included in Appendix E.
Alternative Means of Development Debugging

Besides HP-UX network tracing and logging, several alternative methods and facilities for troubleshooting support are available to a developer. These alternatives are mentioned briefly in this section.
The simplest troubleshooting tool is to open up a file and perform debugging writes into it. However, using this scheme does not allow troubleshooters to control the information recorded, and the file often remains unknown and unnoticed. Further, this scheme is difficult to implement for kernel drivers.

A more sophisticated scheme is to use the HP-UX logging facility, syslog (see syslog(3)), which can capture information from various processes. syslog is similar to the logging facility discussed in this section, but it has fewer features. The single routine call, syslog, creates and sends a log message to the syslog daemon, syslogd. A configuration file, /etc/syslog.conf, determines where the message is dispatched: to a file, to another node, or to a user’s session. For subsystems with a light amount of logging (that is, using simple printf()’s) and no tracing, syslog might be an adequate facility; see the manpage for more detail.

Similarly, STREAMS modules and drivers might use the strlog (see strlog(7)) interface provided by the STREAMS facility to capture information from multiple processes. This interface is similar to syslog, except that additional control over what is captured is provided. The strlog call creates and sends log messages that can be collected with either of the strace or strerr daemons (see strace(1M) and strerr(1M)). As with syslog, this interface does not allow localization of the logged information. See the manpages for more detail.
Creating Networking Device Drivers

Network Tracing and Logging Support for Troubleshooting
8 Writing PCI Device Drivers
Introduction

This chapter presents routines and conceptual material specifically for drivers of Peripheral Component Interconnect (PCI) devices. PCI is an industry-standard bus supported as a bus-nexus CDIO on HP-UX systems as of Release 10.20, as a means of providing expansion I/O. PCI Services are a supplement to the WSIO HP-UX driver environment, providing PCI-specific functionality to drivers that use a PCI bus either as a means to providing expansion slots or for core I/O functionality.

In conjunction with the WSIO Services driver environment, PCI Services form the complete environment necessary to write an HP-UX driver capable of handling a PCI card. The services are generic in nature and not tied to any particular PCI bus adapter.

This chapter corresponds to the PCI Local Bus Specification, Revision 2.1. It also specifies the features, possible limitations, and assumptions of the services that you may need to be aware of.

The HP-UX PCI Services routines are described in the HP-UX Driver Development Reference; they are also summarized in “PCI Services Summary”.

NOTE

The examples in this chapter follow the routine-naming conventions described in “Step 1: Choosing a Driver Name” on page 76, Chapter 5.
Terms and Definitions

800             A shorthand name for HP-UX servers and their operating software, regardless of current model designation.

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 one 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. Base address registers contain PCI addresses when set up.

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, although any card-initiated cycle (for example, a peer-to-peer transaction) is an example of bus mastering.

CDIO            Context-Dependent I/O. A feature of the HP-UX I/O subsystem that provides a consistent interface for I/O busses and device drivers.

DDG             Driver Development Guide. This document essentially describes how to write an HP-UX device or interface driver. It specifically covers routines that are generally implemented in a device or interface driver.

DDR             Driver Development Reference. This document serves as the technical reference for drivers writers. It is a companion to the DDG. It specifically covers HP-UX provided kernel routines and services that are used by device or interface drivers.

DMA             Direct Memory Access. When a card masters the bus in order to do reads or writes to system memory.

map a PCI device or 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 to virtual address**

Mapping a PCI memory space address to a Virtual Address is the act that allows a driver to access PCI space. Access to PCI memory space can be done directly (on workstations only), or by using READ_REG_UINTn_ISC() or WRITE_REG_UINTn_ISC() (for both servers and workstations), with that Virtual Address. The mapping is done through a call to map_mem_to_host().

**map to port handle**

Mapping a PCI I/O space address to a port handle is the act which allows a driver to access I/O space using pci_read_port_uintN_isc() and pci_write_port_uintN_isc(), passing in the port handle as a argument. The mapping is done through a call to pci_get_port_handle_isc().

**PA**

Precision Architecture. Generally used to refer to system features as opposed to card features.

**Virtual Address**

A PA memory virtual address is anything that can be used by the system (PA processor) for a load or store. If a range of Virtual Addresses is mapped to a range of PCI addresses corresponding to a card’s base address register, then loads and stores to these memory addresses will result in loads and stores to the PCI card’s registers. In PCI Services, these accesses are performed with the READ_REG_UINTn_ISC() and WRITE_REG_UINTn_ISC() macros. Workstations may also access these addresses directly with code such as “*dest++ = *source++”.

**PCI**

Peripheral Component Interconnect. An industry-standard bus used on HP-UX systems to
provide 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 device or function**  
The smallest configurable entity. Each function is completely independent from the other, and requires a separate driver instance.

**PCI I/O space**  
The “space” that is addressed by an I/O cycle on the PCI bus. This is a less preferable way to access card registers on cards that choose to respond to “PCI I/O” accesses. The current PA I/O architecture requires two instructions for each processor access of PCI I/O space (an access to set the address in I/O space followed by an access to access the I/O space). Most cards have registers that are in PCI memory space instead of I/O space (that is, they respond to PCI memory cycles, not PCI I/O cycles).

**PCI memory space**  
The “space” that is addressed by a memory cycle on the PCI bus. It is called memory space to indicate that it is memory-mapped input/output, as opposed to a special “I/O” style of input/output. The current PA Workstation I/O architecture allows the PA processor to directly access PCI memory space (i.e., a single instruction). Typical cards map their registers into PCI memory space, meaning they can only be accessed by PCI.
Writing PCI Device Drivers

Terms and Definitions

memory cycles.

**PCI Services**  The software routines and other code that support the PCI bus interface in HP-UX. This document describes the part that is driver visible.

**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_uintN_isc() and pci_write_port_uintN_isc().

**WSIO**  Workstation I/O. A well-defined environment provided for driver implementation on HP-UX workstations and servers.
PCI Overview

This section gives you a brief overview of PCI. It is not intended to be sufficient PCI information in itself; you should be familiar with the PCI Local Bus Specification, Revision 2.1 before trying to write a driver for a PCI card.

PCI Register Spaces

There are three register spaces in PCI:

- PCI Configuration Space
- PCI Memory Space
- PCI I/O Space.

Generic configuration registers are placed in configuration space. Registers for card-specific control and status and for on-card data buffers are generally located in PCI memory space or (less often) in PCI I/O space.

PCI Configuration Space

PCI configuration space holds specific registers having to do with initialization and configuration of PCI devices. Some or all of this register space is the same for all PCI devices, giving generic initialization software the ability to recognize and configure all PCI-compliant devices.

This space is accessed primarily at startup time, when initialization occurs, but there is no prohibition on accessing it at other times after startup.

The following PCI Services access registers in PCI configuration space.

- \texttt{pci\_read\_cfg\_uintN\_isc()}
- \texttt{pci\_write\_cfg\_uintN\_isc()}

These functions take a configuration space offset (0x00–0xff) as their address inputs. See “Defined Constants”. The registers at addresses 0x00–0x3f are defined in the PCI Local Bus Specification, Revision 2.1, but the remainder of the space can be used by the card maker for any card-specific registers it sees fit to put there. In most cases, however, card-specific registers are placed in PCI memory space or PCI I/O space.
instead.

**PCI Memory Space**

Most cards place their registers for control, data buffering, and status in PCI memory space. In HP-UX systems, accesses to PCI memory space have higher performance than access to PCI I/O space. Registers mapped in PCI memory space respond to memory cycles on the PCI bus.

The following PCI Services access registers in PCI memory space.

- `READ_REG_UINTn_ISC()`
- `WRITE_REG_UINTn_ISC()`

These macros take Virtual Addresses (which are mapped to PCI memory addresses) as their address inputs. They have different effects depending on whether or not `PCI_LITTLE_ENDIAN_ONLY` is defined by the driver prior to including `pci.h`. See “The PCI_LITTLE_ENDIAN_ONLY Flag” for more details.

Mapped PCI memory space, on workstations only, can also be accessed directly. In this case you will have to handle “endian” issues yourself.

**PCI I/O Space**

Some cards place their registers for control, data buffering, and status in PCI I/O space. Registers mapped in PCI I/O space respond ONLY to I/O cycles on the PCI bus.

The following PCI Services access registers in PCI I/O space.

- `pci_read_port_uintN_isc()`  
- `pci_write_port_uintN_isc()`

These functions take port handles and offsets as their address inputs.

**PCI Transaction Ordering**

This section covers the ordering of transactions to and from PCI space. These transactions include:

- Processor mastered reads and writes to PCI space
- PCI card mastered reads and writes to host memory
- Interleaved processor and PCI card mastered reads and writes of host
Host bus to PCI bridges used in HP-UX systems need to comply with the transaction ordering requirements of both busses. As a result, in certain cases the order of completion guaranteed under the Producer-Consumer model as defined in the PCI Local Bus Specification, Revision 2.1 is not met.

**Ordering of Processor Mastered Reads and Writes To PCI Space**

This section details transaction ordering for processor mastered PCI transactions. Typical examples of this type of transaction are reading and writing of registers on a PCI interface card.

**Blocking versus Nonblocking Transactions**  Processor mastered reads of PCI space are blocking transactions. This means that ordering is not a problem with reads, since only one read can occur at a time. A read holds the caller (processor) until it completes. The hardware implementation prevents a second processor reading from the same PCI space until the first processor’s read completes.

Writes to PCI registers, on the other hand, are nonblocking (“posted”) transactions. This means that, to get better performance, the writing process does not wait for a write to complete after calling for it (writes do not block). The write will complete on its own, and the writer can do other things, including other writes, in the meantime. Because multiple outstanding uncompleted writes are possible (and common) under this model, ordering must be established on the completion of the writes.

Processor mastered PCI write ordering is relatively simple. If a processor writes to registers A, B, and C in that order, the writes will complete such that they are only observable in the same order (for example, you could never observe that B had been written but A had not been yet). If two or more processors are writing to registers, their ordering with respect to each other is considered irrelevant, but the ordering of their individual writes is preserved as above. This is the order of completion guaranteed under the Producer-Consumer model as defined in the PCI Local Bus Specification, Revision 2.1.

**Write Side-Effects**  The side-effects of any write are not guaranteed to happen immediately. Writes are posted; they will complete eventually.

All posted writes must be flushed and completed before any read is allowed to complete. So, to assume a write’s effects have actually
occurred, a read must be performed to flush the writes posted in the queue. You must keep this in mind when coding register writes; most of the time, it is acceptable to not know when a register write completes, but in some cases, you have to be careful.

A good example of such a case is when a driver's interrupt service routine (ISR) is dealing with the interrupt request register (IRR) on a card. Clearing a bit in the register indicates that the interrupt has been serviced. This is done by posting a write to the register. If the driver posts this write and exits its ISR, it could conceivably get interrupted again immediately because the write hadn't yet reached the bit in the IRR to tell it to stop trying to interrupt. One solution to this potential problem is to make sure to read back the value in the IRR before exiting from the ISR. Most drivers do this anyway so they can handle multiple interrupts in the same ISR visit.

Ordering of PCI Card Mastered Reads and Writes to Host Memory

We use the terms DMA read for a PCI mastered read from host memory, and DMA write for a PCI mastered write to host memory. In current hardware implementations, transaction ordering of DMA reads and DMA writes are ONLY preserved when the target memory location is contained in the same processor cacheline. In other cases, DMA reads are allowed to pass DMA writes and driver writers need to take this behavior into account.

If your driver needs the exact PCI producer-consumer behavior, as seen from the PCI card, you must ensure that the element(s) residing in host memory, requiring strict ordering, are physically on the same cacheline. Current hardware implementations have cachelines that are multiples of 32 bytes in length. For safety you should make sure that you limit your flag or status elements to 32 bytes aligned on MAX_CACHELINE_SIZE boundaries (defined in /usr/include/sys/dma.h).

Ordering of Interleaved Processor and PCI Card Mastered Reads and Writes to Host Memory

If your driver expects PCI or PA ordering rules to apply in this situation, you need to ensure that your producer-consumer elements reside on the same cacheline. The following scenario does not meet the producer-consumer transaction ordering requirements.
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 it's next command from cacheline Y
processor writes command to cacheline Y “do more work”
processor checks status in cacheline X

If the processor's read of cacheline X (status) returns “working”, the processor assumes that the card has not checked its command yet. Therefore it has not gone to sleep and does not need to be waked up. If the status read returns “done”, the processor wakes up the card.

The crux of the problem here is that ordering is not enforced between the two cachelines and DMA reads can pass DMA writes. Thus, both the processor and the card's reads can return the original value. This would result in the card going to sleep and the processor not waking it up.

If you cannot place the status and commands on the same cacheline, you must use some other means to ensure correct behavior. One possible workaround would be to set a timeout to ensure that the above deadlock did not occur. In most cases, commands are written to the card's register, i.e., the command is not in host memory and the above scenario would not apply.

Ordering of Interleaved Processor Mastered Writes, PCI Card Mastered Reads of Host Memory, and Processor Mastered Writes of Host Memory

The following scenario does not meet the producer-consumer transaction ordering requirements.

Processor writes a command to the PCI card to stop processing a task list in host memory, because the processor is about to update or change the list.

Processor begins updating the task list in main memory.

Card does a DMA read of the next (possibly being updated) element of the task list in main memory as a part of normal processing.

Posted processor write to the card arrives at the card telling it to stop processing the list, which unfortunately it has just done.

The problem in this case is similar to the previous problem. DMA reads
Writing PCI Device Drivers

PCI Overview

by a PCI master are allowed to pass processor writes to PCI space. Since processor writes are posted, ordering is not guaranteed on the combination of the internal system bus and the PCI bus. This situation can be avoided by doing a processor read of PCI space immediately following the processor write, as shown next.

- Processor writes a command to the PCI card to stop processing the task list.
- Processor does a “dummy” read of the PCI card (perhaps a status register on the card) to make sure that the “posted” write to PCI space has completed. Perhaps a read of card status is required here in any case to ensure that the DMA engine has stopped fetching tasks.
- Processor updates the task list in main memory.
- Processor writes a command to the PCI card to resume task processing.

The above behavior can occur on all shipping PCI based systems as of the date of this document. Drivers written for workstations should probably always ensure that, where necessary, posted writes are followed by “dummy” reads to ensure ordering. This behavior will probably not occur in servers due to chipset implementation, and may not occur in future workstation products.

PCI Endian Issues

HP PA-RISC is a big-endian system; for a multbyte quantity, the most significant byte (MSB) has the lowest address, and the least significant byte (LSB) has the highest. Intel's i86 processors, on the other hand, are little-endian. Because PCI was derived from the PC world, it, too, is little-endian.

When multbyte words are transferred between the PCI bus and the system bus (HP PA-RISC), the bytes of the word are reversed or swapped by the hardware.

This insures that the receiving system can properly interpret and store the data, from most significant byte to least significant byte. This will not happen when the data is transferred byte-by-byte, but this method is inefficient.

Byte Swapping

So that each system gets data in the format it expects, the PCI hardware
uses a hardwired swapping mechanism at the interface between the two systems. The hardware swaps each byte of a 32-bit word so that all the bytes end up in the correct order on both sides of the interface. This means that large arrays of bytes, such as LAN packets and disk blocks, are in the correct order, even if they are transferred a multibyte word at a time.

This byte-ordering ensures that devices like disks, that are connected to the built-in SCSI on the internal system bus, can instead be connected to a SCSI card on the PCI bus.

When Pre-swapping is Required

Because of the byte swapping, the interpretation of multibyte integers is problematical. To see why this is so, assume that the transfer is occurring from the big-endian system to the little-endian system, and that swapping is being performed. If the byte array in question is a four-byte integer, it will be stored in big-endian format, MSB at the lowest address, on the little-endian side. If a device on the little-endian side of the interface decides to interpret these bytes as a four-byte integer; however, the “value” it will see will have all the bytes reversed. The same thing happens when transfers go in the opposite direction.

To correct the misinterpretation of multibyte integers on the opposite side of the bus, any multibyte quantity that is to be interpreted as an integer will have to be preswapped. This preswapping is then reversed by the hardwired swapping, making the value correct for integer interpretation on the other side of the interface. If the integer is stored in memory, however, it will end up reversed.

Several macros are provided in the file pci.h to assist in swapping data.
PCI Device Setup

This section is a collection of several pieces of information that you need to understand before attempting to set up a PCI device.

Mapping Base Address Registers into PCI Memory and I/O Space

When an HP-UX system boots, processor dependent code (PDC), IO dependent code (IODC), and HP-UX system code maps a PCI card’s memory space base address registers into PCI memory space and I/O space base address registers into PCI I/O space.

The system attempts to map in all memory and I/O regions described by every PCI device or function's memory and I/O base registers located in the PCI configuration space. If a suitable mapping is found, the system will write the base of the range back into the corresponding base address register. This address will be a PCI memory address if the base register identifies itself as a memory base, and a PCI I/O address if the base address identifies itself as an I/O base.

A driver's `driver_attach()` routine can then access the values loaded into the base registers in configuration space. It is important that a driver does not overwrite these addresses with different values, except as follows: As long as response to memory or I/O accesses via the command register has not yet been enabled, it is acceptable to store the register contents, write all ones to the register to determine the region size (as explained in the PCI Local Bus Specification, Revision 2.1) and, then restore the original contents.

Using the Base Address Registers

Before a driver can actually use these base addresses, another kind of "mapping" must take place. The problem is that the addresses placed in the base address registers by the system do not contain Virtual Addresses usable by the computer. Instead, they contain PCI addresses, used to talk on the bus. If a base address register is a memory base, it contains a PCI memory address. If it is an I/O base, it contains a PCI I/O address (See “PCI Register Spaces” for more information).

In either case, to use the PCI address in the base register, a mapping to a
PA resource must take place, in order to allow the system to access the
registers pointed to by the base.

It is very important that you do NOT arbitrarily mask bit 0 of a base
address register. This bit indicates whether or not this particular set of
registers responds to PCI memory cycles or PCI I/O cycles. During early
PDC/IODC configuration, the defined base address registers are written
in a manner prescribed by the PCI specification to determine size,
alignment, and access type. If bit 0 is a “1” then PDC/IODC probing has
determined that this particular register set ONLY responds to I/O cycles.
If the base address register responds to I/O cycles, you MUST use the
PCI services provided port I/O routines for access.

**Using PCI Memory Base Registers**

To use a PCI memory base address register, the range of PCI memory
space must be mapped to a range of PA memory space. This is
accomplished by calling `map_mem_to_host()`. The `map_mem_to_host()`
call takes the PCI memory address (obtained directly from the base
address register) and a size as inputs, and returns a Virtual Address that
can be used to access that PCI address range. The accessor macros,
`READ_REG_UINTn_ISC()` and `WRITE_REG_UINTn_ISC()`, take PA virtual
memory addresses as arguments, not PCI memory addresses.

---

**NOTE**

After reading a PCI memory base address register’s value out of PCI
configuration space, it is usually necessary to mask off the bottom four
bits prior to making services calls such as `map_mem_to_host()`, since
they have special values defined by the PCI Local Bus Specification,
Revision 2.1 (See “Sample driver_attach() Routine” for an example).

---

Once this virtual mapping is done, the machine uses PA memory-mapped
I/O to access the range. In other words, accesses to that range of PA
memory space will be transmitted through into the PCI memory space.
This just means that loads and stores to these PA memory addresses will
result in loads and stores to the registers you wish to access.

For WSIO drivers, the `if_reg_ptr` member of the ISC structure is a
Virtual Address corresponding to a base address register that has
already had this virtual mapping done to make it usable by the driver
and system. If `if_reg_ptr` is NULL, the driver needs to map the range
itself (see “Mapping the Memory Base Register” for information on this).
This is generally done in a PCI device’s `driver_attach()` routine.
Using PCI I/O Base Registers
To use a PCI I/O base address register, the corresponding range of PCI I/O space must be mapped to a resource managed by PCI Services called a port handle, defined by the PCI structure `PCI_PORT_HNDL`. The PCI I/O space accessor functions, `pci_read_port_uintN_isc()` and `pci_write_port_uintN_isc()`, take port handles as arguments.

To do this mapping from a PCI I/O address to a port handle, the driver must read the I/O base registers from configuration space and call `pci_get_port_handle_isc()`, which takes a PCI I/O space address and a size as input, and returns a `PCI_PORT_HNDL` as output. This port handle (with an offset) is then used to access the registers in PCI I/O space.

---

**NOTE**

When reading a PCI I/O base address register's value out of PCI configuration space, it is necessary to mask off the bottom two bits prior to making a call to services such as `pci_get_port_handle()`, since they have special values defined by the PCI Local Bus Specification, Revision 2.1 (See `pci_get_port_hndl_isc(PCI3)` for an example).

---

Automatic IRQ Determination
PCI drivers calling `isrlink()` and `isrunlink()` should always pass `-1` as the `irq_line` argument. This argument value causes the functions to read the needed IRQ information from the PCI device or function configuration space Interrupt Pin and/or Interrupt Line registers and use it to set up the ISR properly. If you need the IRQ information, you can read it from the Interrupt Line register.

Mapping the Memory Base Register
Many cards will have only a single range of registers (only a single memory base address register). For cards like these, the `if_reg_ptr` field in the ISC structure is useful.

PCI Services automatically maps one memory-space register into the `isc->if_reg_ptr` field in the following manner and with the following limitations:

- Only the first nonzero 32-bit memory base register found is mapped,
starting at 0x10 and searching up to 0x24 inclusive. These are the six
defined base address register locations in PCI configuration space. A
Virtual Address for accessing this register is stored in \texttt{if_reg_ptr}.

- However, if that base register’s size (the size of the register range) is
  in excess of 8 KB, it is NOT mapped and \texttt{if_reg_ptr} is set to NULL. In
  this case, the driver itself must map the base registers it wants, using
  the PCI bus-dependent configuration access routines in conjunction
  with \texttt{map_mem_to_host()}.

- If \texttt{if_reg_ptr} is NULL and the result of a \texttt{map_mem_to_host()} call is
  NULL, then for whatever reason, this particular address could NOT
  be mapped and you MUST NOT attempt to access it.

These limitations are necessary to define which of many possible base
registers will be mapped, as well as to prevent unnecessary use of
translation lookaside buffers (TLB). If PCI Services do not map in any
memory base register, or if there are more registers than the first one
found as above, the driver can read the base registers explicitly from the
PCI device or function’s configuration space and get a PA virtual
mapping with the \texttt{map_mem_to_host()} kernel routine. (See “Sample
driver_attach() Routine” for an example).

The limitations also prevent wasting of kernel resources on base
registers that we may not wish to map in the normal way (for example, a
graphics card frame buffer is an enormous range that should be treated
differently from a regular register range). PCI Services has arbitrarily
decreed that anything bigger than 8 KB should be dealt with by the
driver, not mapped automatically by WSIO services.

**PCI Configuration Space Restrictions**

The registers in the PCI configuration space of each device are described
in the PCI Local Bus Specification, Revision 2.1. Many of these registers
are writable, but not every writable register is appropriate for a driver to
modify. Some of the fields are set up on behalf of the driver and card by
the system, which has information that a driver or card could not know
about system parameters.

The basic guideline is that things that you do not understand or need not
have anything to do with should not be altered. The following are some
examples of configuration registers to leave alone:

- The Command Register (most parts of it)
The command register must be written by drivers in order to enable bus-mastering, memory space access, and I/O space access, among other things. Many bits in this register are irrelevant to a driver and some have already been set by the system. Bits in the command register that may have been previously set must not be overwritten. Therefore, when a driver wants to set a bit in the register, it must first read the current state of the register, use bitwise OR or AND to make any changes, then write the value back. This procedure preserves bits previously set by the system.

- The Latency Timer Register
  This is set by the system. It should not be tampered with by individual drivers, as incorrect settings can degrade overall system performance.

- The Cache Line Size Register
  This register is set by the system to match the machine's cache line. Drivers do not know the cache line size for the particular machine they are currently running on, so they should not change this register's contents.

- The Base Address Registers
  The system uses the information in these registers to map their ranges into PCI memory and I/O space. It then writes a value back into the register corresponding to the base of the range it allocated. These ranges should not be overwritten by drivers, with one exception. In some cases, it may be necessary for a driver to determine the size and alignment of the range a base address register is mapped to. The procedure for getting this information involves writing all ones to the register, reading the result back, and decoding it for the needed values, as described in the PCI Local Bus Specification, Revision 2.1. Doing this is permitted as long as the original value is read and stored first, then restored to the register after the size has been determined. This should only be done before memory or I/O transactions to the card have been enabled through the command register.

- The Interrupt Line Register
  System-specific interrupt routing information is stored in this register. Writing a new value to it will probably cause the card to stop working.
PCI Device Operation

The PCI_LITTLE_ENDIAN_ONLY Flag

We recommended that drivers define the PCI_LITTLE_ENDIAN_ONLY flag before they include pci.h. This will help them get better performance from their I/O accesses.

Most PCI drivers are written for cards whose primary method of accessing registers is through PCI memory space.

PCI drivers written for workstations only, currently all third party drivers, may use direct C code constructs to access registers in PCI memory space. For example:

```c
myClearRegs(regsToInit,size)
#define *regsToInit;  // Not necessary
int size;
{
  int i;
  for (i=0;i<size;i++)
    *regsToInit++ = 0;
}
```

These drivers may also use the READ_REG_UINTn_ISC() and WRITE_REG_UINTn_ISC() macros with the PCI_LITTLE_ENDIAN_ONLY flag defined in the pci.h header. The choice of whether to directly access a register or to use one of the macros essentially becomes whether or not you want to swap.

The READ_REG_UINTn_ISC() and WRITE_REG_UINTn_ISC() macros are the safest accessors of PCI memory space, but what they are actually defined to do depends on whether or not the PCI_LITTLE_ENDIAN_ONLY flag was defined by the driver before the driver source code included the pci.h header.

If the driver does NOT explicitly define PCI_LITTLE_ENDIAN_ONLY before including pci.h, then the macros expand into function calls that are guaranteed to byte-swap correctly and perform the memory access. This can be considered "extra safe" mode; it will always work on all bus adapters in the future. The function calls guarantee PCI-adapter-independence. However, extra function call overhead is added to the register access, reducing its performance.
Writing PCI Device Drivers

PCI Device Operation

If the driver DOES explicitly define PCI_LITTLE_ENDIAN_ONLY, the performance loss due to the function call is taken away. In this case, the macros are expanded by the preprocessor into a series of in-line instructions that byte-swap and perform the access without a function call, under the assumption that the PCI adapter under which the card is running has directly mapped the PCI memory space into driver-accessible PA I/O space. This assumption is valid for all current and planned PCI adapters, with the single exception of a few special PA internal system bus based server PCI card projects. All regular drivers (i.e., those that are not explicitly written to drive a specially-equipped PA internal system bus based card) will benefit from defining the PCI_LITTLE_ENDIAN_ONLY flag and should do so before including pci.h.

The following pseudocode (resembling and summarizing the actual code in pci.h) may help explain the flag's relation to the macros, and how and why to use it:

```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 (DMA)

A PCI device acting as a PCI bus master uses direct memory access (DMA) to generate read or write cycles that access locations in PA memory and card memory. DMA is a primary method of getting information to or from a card in large chunks, as opposed to doing many reads or writes to buffers of card registers.

PCI has no special routines to perform DMA. It uses the standard WSIO
Services calls for bus-independent DMA, including:

- `init_map_context()`
- `wsio_map()`, `wsio_fastmap()`, and `wsio_unmap()`
- `dma_setup()` and `dma_cleanup()`
- The `iovec` and `dma_parms` structures

In the HP-UX Driver Development Reference, see `dma_cleanup(WSIO3)`, `dma_parms(WSIO4)`, `dma_setup(WSIO3)`, `init_map_context(CDIO3)`, `iovec(KER4)`, `wsio_fastmap(WSIO3)`, `wsio_map(WSIO3)`, and `wsio_unmap(WSIO3)`.

Be aware that certain combinations of WSIO mapping service calls can interact with PCI masters to create an inconsistent view of memory. See “PCI Masters and Coherence”.

Many EISA drivers make calls to functions like `eisa_dma_setup()` and `eisa_dma_cleanup()`. There are no corresponding PCI functions.

The only thing PCI-specific about performing DMA with a PCI device is that the device's command register (PCI_CS_COMMAND) in PCI configuration space contains a bit (PCI_CMD_BUS_MASTER) that must be set (with `pci_write_cfg_uintN_isr()`) in order to allow the device to master the bus. The use of this bit is illustrated in “Sample driver_attach() Routine”.

---

Chapter 8
PCI Masters and Coherence

It is possible for prefetching of host memory by the hardware chipset to result in a PCI master reading stale data, even though the proper \texttt{dma\_sync} calls have been made. This does not occur if the mapping is done with \texttt{wsio\_map()} with flags \texttt{IO\_NO\_SEQ} and \texttt{IO\_SAFE} set. See \texttt{pci\_errata(PC15)} in the HP-UX Driver Development Reference for details.
Leveraging Existing Drivers

Multibus Drivers

Some cards for different busses have similar chip sets, making the programming models very similar for the base functionality. Consequently, a single driver can handle the functionality for the different bus cards.

Writing a Multibus Driver

A multibus driver is one in which a similar chip set appears on cards that plug into multiple busses. An example of this is the current SCSI driver. Similar SCSI chips exist in devices on the GSC bus, the EISA bus, and the PCI bus. A single driver, scsi_c720, is capable of controlling these SCSI chips no matter where they live.

Because the programming model of the base functionality is so similar, it makes sense to have a single driver to handle this functionality. Conversely, however, the bus-specific initialization of the nonbase functionality can often be radically different. The WSIO environment supports multibus drivers in the following ways:

- Many of the initialization functions are embedded in bus-independent functions that have bus-dependent implementations. This means that WSIO is responsible for making sure that the right thing is done when a driver calls a generic function like `map_mem_to_host()`. This moves the handling of bus-specific differences out of the driver and into the WSIO environment. Keeping the driver clean of calls specific to the current PCI adapter. See “Bus-Independent Functionality, Bus-Dependent Implementation”.

- Since each bus has a different attach chain, drivers can provide a separate `driver_attach()` routine for each bus. With careful handling, this can localize bus-specific functionality in the `driver_attach()` routines, allowing the `driver_if_init()` routine to handle bus-independent initialization and keeping the rest of the driver routines clean.

Whether or not you are planning to write a multibus driver, it is a good idea to keep as much PCI specifics in the `driver_attach()` routine as possible, just in case a card comes along someday for a new bus that uses...
the same or similar chips as the PCI card you are writing a driver for now. This is only a suggestion, as it does not make sense to compromise your current driver or make a huge and ungainly `driver_attach()` routine if there is clearly no need to.

**Bus-Independent Functionality, Bus-Dependent Implementation**

This class of functions allows multibus drivers to make a single call, allowing the driver environment to hide any bus-dependent implementation.

In PCI, the following features are supported. (There are a host of completely bus-independent functions that, by having no dependency on PCI, are supported by definition.)

- `isrlink()` and `isrunlink()`:
  Set `irq_line` to -1 to have the card supply the IRQ number to the system. See “Automatic IRQ Determination”.

- `isc->if_reg_ptr` value:
  One memory space base register is mapped automatically, subject to the conditions described in “Mapping the Memory Base Register”.

- `wsio_map()`, `wsio_unmap()`, `wsio_dma_alloc()`, `wsio_dma_free()`, and others in the WSIO family of coherent I/O DMA services.

- `dma_setup()` and `dma_cleanup()`

The WSIO functions `wsio_get_interrupts()` and `wsio_get_registers()` are not supported for PCI. See “Unsupported WSIO Functions” for details.
PCI Services Summary

PCI Services are accessed through special PCI functions that allow device and interface drivers to be much smaller and more supportable.

These functions are summarized here and described in detail in the HP-UX Driver Development Reference.

- **pci_desc_bus_transactions_isc()** - Allow a driver to describe the typical bus-performance-transaction size.
- **pci_get_fru_info_isc()** - Get the field replaceable unit (FRU) information for the device associated with an ISC.
- **pci_get_port_hndl_isc()** - Get a system-defined handle for manipulating the range of PCI I/O-space ports.
- **pci_read_cfg_uintN_isc()** - Read an 8-, 16-, or 32-bit unsigned integer from a PCI configuration register.
- **pci_read_port_uintN_isc()** - Read little-endian data from a PCI I/O-space port previously identified by a call to pci_get_port_hndl_isc().
- **pci_unget_port_hndl_isc()** - Delete a handle returned by pci_get_port_hndl_isc()
- **pci_write_cfg_uintN_isc()** - Write an 8-, 16-, or 32-bit unsigned integer into a PCI configuration register.
- **pci_write_port_uintN_isc()** - Write little-endian data to a PCI I/O port previously identified by a call to pci_get_port_hndl_isc().
- **CONNECT_INIT_ROUTINE()** - Associate a driver_if_init() routine with the driver.
- **PCI_ATTACH_DEV_INIT_ERROR()** - Notify WSIO Services that an error occurred during a device's initialization.
- **READ_REG_UINTn_ISC()** - Read and byte-swap 8-, 16-, or 32-bit data from a little-endian bus.
- **WRITE_REG_UINTn_ISC()** - Byte-swap and write 8-, 16-, or 32-bit data to a little-endian bus or a host memory area shared by the driver and a little-endian bus master.
Unsupported WSIO Functions

PCI Services do NOT support the following WSIO functions.

- `wsio_get_interrupts()`
  This function is provided to tell the driver what interrupt line a card is using. This information is primarily used to link or unlink an ISR. In PCI, `isrlink()` and `isrunlink()` should be called with `irq_line` set to -1, indicating that the system should determine the appropriate IRQ from the card, and rendering the `wsio_get_interrupts()` call unnecessary. If the IRQ is needed for some other reason, it can be read from the Interrupt Line register in PCI configuration space.

- `wsio_get_registers()`
  This function is designed to return the base address register for a card's memory-mapped I/O. For PCI this information is available in the ISC at `driver_attach` time as the value `if_reg_ptr`. Also see “Mapping the Memory Base Register”.
Multiprocessor (MP) Safety

All PCI drivers should be coded to be MP safe.

Specifically, this means that they should not rely upon SPL levels to guarantee exclusive access to critical sections, but should instead protect their own critical sections using spinlocks, semaphores, and other methods of MP protection. See Chapter 4, “Multiprocessing,” for details.
Constants and Data Structures

The constant definitions and data structures are defined in the PCI header file, pci.h.

User Visible PCI-Specific Data Structures

typedef struct _pci_id
{
    uint16_t vendor_id;
    uint16_t device_id;
} PCI_ID;

Defined Constants

/* Configuration space offsets. */
#define PCI_CS_VENDOR_ID 0x00
#define PCI_CS_DEVICE_ID 0x02
#define PCI_CS_COMMAND 0x04
#define PCI_CS_STATUS 0x06
#define PCI_CS_REV_ID 0x08
#define PCI_CS_CLASS_PROG_IF 0x09
#define PCI_CS_CLASS_SUB_CLASS 0x0a
#define PCI_CS_CLASS_BASE 0x0b
#define PCI_CS_CACHE_LINE_SIZE 0x0c
#define PCI_CS_LATENCY_TIMER 0x0d
#define PCI_CS_HEADER_TYPE 0x0e
#define PCI_CS_BIST 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_INVALID_EN 0x010
#define PCI_CMD_VGA_PAL_SNOOP 0x020
#define PCI_CMD_PARITY_ERR_RESP 0x040
#define PCI_CMD_WAIT_CYCLE_CNTL 0x080
#define PCI_CMD_SERR_ENABLE 0x100
#define PCI_CMD_FAST_BACK_EN 0x200
A Sample PCI Driver

The following example is a skeleton that demonstrates how to write a PCI device driver in HP-UX using PCI and WSIO Services. The only part of this example that is PCI-specific is the driver_attach() routine. The other parts are typical of all WSIO drivers. They are included here for context and completeness. Chapter 5, “Writing a Driver,” contains more complete information on the structures and functions needed to write a WSIO driver.

We have a hypothetical PCI device, the ZZZ8109C PCI Blender card, for which we want to write a driver.

The blender is a character device, so our driver will be a character device driver. A character device is the counterpart of a block device, and has to do with how a device accesses its data and does DMA. The only type of PCI card that would be a block device would be a SCSI adapter or disk or tape drive controller.

Our example driver is written as a monolithic driver. This means it is both an interface driver (one that touches real hardware and registers) and a device driver (one that has a device special file). Even though we are writing both an interface and a device driver, we specify T_INTERFACE in the wsio_drv_info_t structure, since we cannot specify both.

Following the routine-naming conventions described in “Step 1: Choosing a Driver Name” on page 76, Chapter 5, we name the driver ZZZ and place it in the (arbitrary) class blender.

Sample WSIO Setup and Structures

We include the necessary header files. See the reference pages for each kernel call and data structure your driver uses to find out which headers your driver requires. WSIO drivers generally require the <wsio/wsio.h> header file. PCI drivers also require the <sys/pci.h> header file.

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

Next, we declare the driver’s routines that can be called by the kernel. These are used in the drv_ops_t structure.

```c
int  ZZZ_open();
```
int ZZZ_close();
int ZZZ_read();
int ZZZ_write();
int ZZZ_ioctl();

We need a ZZZ_saved_attach function pointer to store the old head of the attach chain when we add our ZZZ_attach() routine to it in the ZZZ_install() routine. We also need values for vendor ID (ZZZ_VEN_ID) and device ID (ZZZ_DEV_ID) for the comparison in ZZZ_attach().

static int (*ZZZ_saved_attach)();
int ZZZ_VEN_ID = value
/* these should be initialized */
int ZZZ_DEV_ID = value
/* these should be initialized */

The drv_ops_t structure specifies the “external” driver routines to the kernel. The flags specify that the driver should be called on all device closes and that it is MP safe. See “The drv_ops_t Structure Type” on page 80, Chapter 5, for further details.

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 */
    NULL, /* reserved */
    NULL, /* reserved */
    C_ALLCLOSES | C_MGR_IS_MP /* flags */
};

The drv_info_t structure specifies the driver’s name and class. The flags specify that the driver is character type and MP safe and that the configuration, including major number, should be saved and retained across reboots. See “The drv_info_t Structure Type” on page 83, Chapter 5, for further details.

static drv_info_t ZZZ_info =
Writing PCI Device Drivers

A Sample PCI Driver

```c
{
    "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 that it conforms to the Release 10.0 I/O specifications. See “The `wsio_drv_data_t` Structure Type” on page 85, Chapter 5, for further details.

```c
static wsio_drv_data_t ZZZ_data =
{
    "blender",
    /* matches class name for T_INTERFACE drivers */
    T_INTERFACE,
    /* drv_type - either T_DEVICE or T_INTERFACE */
    DRV_CONVERGED, /* drv_flags */
    NULL, /* optional function */
    NULL /* optional function */
};
```

The `wsio_drv_info_t` structure ties the preceding three structures together into a single structure used in the `ZZZ_install()` routine's call to `wsio_install_driver()`. See “The `wsio_drv_info_t` Structure Type” on page 86, Chapter 5, for further details.

```c
static wsio_drv_info_t ZZZ_wsio_info =
{
    &ZZZ_info,
    &ZZZ_ops,
    &ZZZ_data
};
```

Sample WSIO Routines

Sample `driver_install()` Routine

A driver's `driver_install()` routine registers the driver and its structures with WSIO Services and the I/O subsystem. It also links the
The name you can give this routine is restricted. It must begin with the name of your driver, for example, ZZZ, and end with _install, as in ZZZ_install. See “Step 1: Choosing a Driver Name” on page 76, Chapter 5.

```c
int ZZZ_install()
{
    int ret;

    /*
     * Register our driver information with WSIO services
     */
    ret = wsio_install_driver(&ZZZ_wsio_info);

    if (ret)
    {
        /*
         * If the install worked,
         * link ourselves into the pci_attach chain
         */
        ZZZ_saved_attach = pci_attach;
        pci_attach = ZZZ_pci_attach;
    }

    /*
     * Exit, returning the value we got
     * from the wsio_install_driver() call
     */
    return(ret);
}
```

**Sample driver_attach() Routine**

For interface and monolithic drivers, the `driver_attach()` routine is linked into the global attach list for PCI drivers in the `driver_install()` routine.

A driver’s `driver_attach()` routine is called whenever the system finds a piece of hardware it thinks the driver might want to claim (this driver

---

**NOTE**

The name you can give this routine is restricted. It must begin with the name of your driver, for example, ZZZ, and end with _install, as in ZZZ_install. See “Step 1: Choosing a Driver Name” on page 76, Chapter 5.

```c
int ZZZ_install()
{
    int ret;

    /*
     * Register our driver information with WSIO services
     */
    ret = wsio_install_driver(&ZZZ_wsio_info);

    if (ret)
    {
        /*
         * If the install worked,
         * link ourselves into the pci_attach chain
         */
        ZZZ_saved_attach = pci_attach;
        pci_attach = ZZZ_pci_attach;
    }

    /*
     * Exit, returning the value we got
     * from the wsio_install_driver() call
     */
    return(ret);
}
```

**Sample driver_attach() Routine**

For interface and monolithic drivers, the `driver_attach()` routine is linked into the global attach list for PCI drivers in the `driver_install()` routine.

A driver’s `driver_attach()` routine is called whenever the system finds a piece of hardware it thinks the driver might want to claim (this driver

---

**NOTE**

The name you can give this routine is restricted. It must begin with the name of your driver, for example, ZZZ, and end with _install, as in ZZZ_install. See “Step 1: Choosing a Driver Name” on page 76, Chapter 5.

```c
int ZZZ_install()
{
    int ret;

    /*
     * Register our driver information with WSIO services
     */
    ret = wsio_install_driver(&ZZZ_wsio_info);

    if (ret)
    {
        /*
         * If the install worked,
         * link ourselves into the pci_attach chain
         */
        ZZZ_saved_attach = pci_attach;
        pci_attach = ZZZ_pci_attach;
    }

    /*
     * Exit, returning the value we got
     * from the wsio_install_driver() call
     */
    return(ret);
}
```

**Sample driver_attach() Routine**

For interface and monolithic drivers, the `driver_attach()` routine is linked into the global attach list for PCI drivers in the `driver_install()` routine.

A driver’s `driver_attach()` routine is called whenever the system finds a piece of hardware it thinks the driver might want to claim (this driver

---

**NOTE**

The name you can give this routine is restricted. It must begin with the name of your driver, for example, ZZZ, and end with _install, as in ZZZ_install. See “Step 1: Choosing a Driver Name” on page 76, Chapter 5.

```c
int ZZZ_install()
{
    int ret;

    /*
     * Register our driver information with WSIO services
     */
    ret = wsio_install_driver(&ZZZ_wsio_info);

    if (ret)
    {
        /*
         * If the install worked,
         * link ourselves into the pci_attach chain
         */
        ZZZ_saved_attach = pci_attach;
        pci_attach = ZZZ_pci_attach;
    }

    /*
     * Exit, returning the value we got
     * from the wsio_install_driver() call
     */
    return(ret);
}
```

**Sample driver_attach() Routine**

For interface and monolithic drivers, the `driver_attach()` routine is linked into the global attach list for PCI drivers in the `driver_install()` routine.

A driver’s `driver_attach()` routine is called whenever the system finds a piece of hardware it thinks the driver might want to claim (this driver

---

**NOTE**

The name you can give this routine is restricted. It must begin with the name of your driver, for example, ZZZ, and end with _install, as in ZZZ_install. See “Step 1: Choosing a Driver Name” on page 76, Chapter 5.

```c
int ZZZ_install()
{
    int ret;

    /*
     * Register our driver information with WSIO services
     */
    ret = wsio_install_driver(&ZZZ_wsio_info);

    if (ret)
    {
        /*
         * If the install worked,
         * link ourselves into the pci_attach chain
         */
        ZZZ_saved_attach = pci_attach;
        pci_attach = ZZZ_pci_attach;
    }

    /*
     * Exit, returning the value we got
     * from the wsio_install_driver() call
     */
    return(ret);
}
```

**Sample driver_attach() Routine**

For interface and monolithic drivers, the `driver_attach()` routine is linked into the global attach list for PCI drivers in the `driver_install()` routine.

A driver’s `driver_attach()` routine is called whenever the system finds a piece of hardware it thinks the driver might want to claim (this driver

---

**NOTE**

The name you can give this routine is restricted. It must begin with the name of your driver, for example, ZZZ, and end with _install, as in ZZZ_install. See “Step 1: Choosing a Driver Name” on page 76, Chapter 5.

```c
int ZZZ_install()
{
    int ret;

    /*
     * Register our driver information with WSIO services
     */
    ret = wsio_install_driver(&ZZZ_wsio_info);

    if (ret)
    {
        /*
         * If the install worked,
         * link ourselves into the pci_attach chain
         */
        ZZZ_saved_attach = pci_attach;
        pci_attach = ZZZ_pci_attach;
    }

    /*
     * Exit, returning the value we got
     * from the wsio_install_driver() call
     */
    return(ret);
}
```

**Sample driver_attach() Routine**

For interface and monolithic drivers, the `driver_attach()` routine is linked into the global attach list for PCI drivers in the `driver_install()` routine.

A driver’s `driver_attach()` routine is called whenever the system finds a piece of hardware it thinks the driver might want to claim (this driver

---

**NOTE**

The name you can give this routine is restricted. It must begin with the name of your driver, for example, ZZZ, and end with _install, as in ZZZ_install. See “Step 1: Choosing a Driver Name” on page 76, Chapter 5.

```c
int ZZZ_install()
{
    int ret;

    /*
     * Register our driver information with WSIO services
     */
    ret = wsio_install_driver(&ZZZ_wsio_info);

    if (ret)
    {
        /*
         * If the install worked,
         * link ourselves into the pci_attach chain
         */
        ZZZ_saved_attach = pci_attach;
        pci_attach = ZZZ_pci_attach;
    }

    /*
     * Exit, returning the value we got
     * from the wsio_install_driver() call
     */
    return(ret);
}
```

**Sample driver_attach() Routine**

For interface and monolithic drivers, the `driver_attach()` routine is linked into the global attach list for PCI drivers in the `driver_install()` routine.

A driver’s `driver_attach()` routine is called whenever the system finds a piece of hardware it thinks the driver might want to claim (this driver

---

**NOTE**

The name you can give this routine is restricted. It must begin with the name of your driver, for example, ZZZ, and end with _install, as in ZZZ_install. See “Step 1: Choosing a Driver Name” on page 76, Chapter 5.

```c
int ZZZ_install()
{
    int ret;

    /*
     * Register our driver information with WSIO services
     */
    ret = wsio_install_driver(&ZZZ_wsio_info);

    if (ret)
    {
        /*
         * If the install worked,
         * link ourselves into the pci_attach chain
         */
        ZZZ_saved_attach = pci_attach;
        pci_attach = ZZZ_pci_attach;
    }

    /*
     * Exit, returning the value we got
     * from the wsio_install_driver() call
     */
    return(ret);
}
```

**Sample driver_attach() Routine**

For interface and monolithic drivers, the `driver_attach()` routine is linked into the global attach list for PCI drivers in the `driver_install()` routine.

A driver’s `driver_attach()` routine is called whenever the system finds a piece of hardware it thinks the driver might want to claim (this driver

---

**NOTE**

The name you can give this routine is restricted. It must begin with the name of your driver, for example, ZZZ, and end with _install, as in ZZZ_install. See “Step 1: Choosing a Driver Name” on page 76, Chapter 5.

```c
int ZZZ_install()
{
    int ret;

    /*
     * Register our driver information with WSIO services
     */
    ret = wsio_install_driver(&ZZZ_wsio_info);

    if (ret)
    {
        /*
         * If the install worked,
         * link ourselves into the pci_attach chain
         */
        ZZZ_saved_attach = pci_attach;
        pci_attach = ZZZ_pci_attach;
    }

    /*
     * Exit, returning the value we got
     * from the wsio_install_driver() call
     */
    return(ret);
}
```
put its `driver_attach()` function on the `pci_attach` chain, so the system calls it every time a new PCI device is found). The `driver_attach()` routine first checks to see if this is the type of hardware it can claim, then claims it if it wants it and performs whatever initialization the card requires.

- PCI Services will NOT enable a PCI device or function response to memory accesses, I/O accesses, or PCI device or function mastering of the bus. This ensures that a PCI device or function remains completely disconnected from the bus until after driver initialization. It is the responsibility of the driver to do this, as shown in the following sample attach routine.

- The driver is responsible for ensuring that the contents of a memory or I/O base register are not zero. All zeros indicates that either the specified configuration space register is not implemented by the PCI device or function or that the system could not find the resources to map the corresponding space into the system. If alternate register mappings exist, and those base registers are not zero, it is acceptable for the driver to use those mappings instead.

The driver must enable memory access, I/O access, and DMA.

```c
ZZZ_pci_attach(parm, isc)
  uint32_t id;
  struct isc_table_type *isc;
{
  uint8_t rev_id;
  uint16_t command_reg;
  uint32_t base_addr;
  PCI_ID *id = (PCI_ID *)&parm /* for LP64 */

  /*
   * see if this is our card
   */
  if (!(id->vendor_id == ZZZ_VEN_ID && id->device_id == ZZZ_DEV_ID))
    {
      goto exit0;
    }

  /*
   * If we use a standard bus interface chip we need to
   * check subsystem vendor ID and subsystem ID here
   */
```
* to make sure that our driver should be the
driver claiming this device
*/

/*
* Get the card revision
*/
cpci_read_cfg_uint8_isc(isc, PCI_CS_REV_ID, &rev_id);

/*
* We must check the isc->if_reg_ptr
* before we use it. If it’s NULL,
* we read our base register and map it ourselves.
* But if isc->if_reg_ptr isn’t NULL, PCI
* services already did the mapping work for us
*/
if (isc->if_reg_ptr == NULL) {
    /*
    * We need to map our own base address.
    * Save the value in if_reg_ptr.
    * Get our physical base memory address.
    * For ZZZ, memory is at reg 0x10
    */
    pci_read_cfg_uint32_isc(isc, 0x10, &base_addr);

    /*
    * make sure we have a memory BAR
    * instead of an IO BAR
    */
    if (base_addr & 0x01) {
        printf("ZZZ - no memory BAR\n");
        goto exit0;
    }

    /*
    * Mask off the bottom four bits of the PCI
    * memory base register (see PCI spec for 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 us PCI memory space at all.
    */
    if (base_addr == 0) {
        goto err0;
    }

Chapter 8 311
A Sample PCI Driver

/*
 * Get a virtual translation for card registers.
 * Assume there are 512 bytes of registers.
 * Save the value in if_reg_ptr.
 */
if ((isc->if_reg_ptr = map_mem_to_host(isc, base_addr, 512)) == NULL)
{
    goto err0;
}

/*
 * Use if_reg_ptr to access the registers.
 * Enable memory access and bus mastering
 * (note: other bits in the register must be preserved)
 */
pci_read_cfg_uint16_isc(isc, PCI_CS_COMMAND, &command_reg);
pci_write_cfg_uint16_isc(isc, PCI_CS_COMMAND,
    command_reg | PCI_CMD_MEM_SPACE | PCI_CMD_BUS_MASTER);

/*
 * Set up our interrupt handler.
 * Note that -1 is the third argument to isrlink().
 */
if (isrlink(isc, ZZZ_isr, -1, isc, 0) < 0) {
    goto err1;
}

/*
 * set up our init routine to be run later
 */
CONNECT_INIT_ROUTINE(isc, ZZZ_if_init);

/*
 * If everything okay, claim this card
 */
isc_claim(isc, &ZZZ_wsio_info);

/*
 * Exit without error
 */
goto exit0;

err1:
/*
 * clean up the mapping
 */
unmap_mem_from_host(isc, isc->if_reg_ptr, 512);
err0:
    /*
     * indicate that we had an error
     */
    PCI_ATTACH_DEV_INIT_ERROR(isc);

exit0:
    /*
     * Always exit by calling rest of chain
     * Use link established in ZZZ_install()
     */
    return ZZZ_saved_attach(parm, isc);
}

Other Driver Entry Point Routines

The other routines defined by the code above must also be declared and written. These functions include the following list:

- **ZZZ_if_init()**: Initialization of the card after the driver_attach() routine
- **ZZZ_isr()**: The driver’s interrupt service routine
- **ZZZ_open()**: The drv_ops_t-defined entry point for open().
- **ZZZ_close()**: The drv_ops_t-defined entry point for close().
- **ZZZ_read()**: The drv_ops_t-defined entry point for read().
- **ZZZ_write()**: The drv_ops_t-defined entry point for write().
- **ZZZ_ioctl()**: The drv_ops_t-defined entry point for ioctl().

The code for these functions is driver dependent. See “Step 6: Writing Entry Point Routines” on page 119 and “Step 7: Writing Other Driver Routines” on page 150, Chapter 5. See also close(2), ioctl(2), open(2), read(2), write(2).
9 Writing SCSI Device Drivers
This chapter presents routines and conceptual material specifically for
drivers of SCSI devices. Chapter 5, “Writing a Driver,” describes the
general configuration and entry-point driver routines, such as
driver_open and driver_write. If you are writing a SCSI driver, you
must provide routines from both Chapter 5, “Writing a Driver,” and this
chapter.

The HP-UX Driver Development Reference describes the SCSI Services
routines.

SCSI devices can be controlled in two ways, both supported by the SCSI
Services routines. Kernel drivers, following the scsi_disk model, are
the traditional method. They are described in this chapter and in
scsi_disk(7). However, many SCSI devices do not need a special driver.
Instead, user-programs pass ioctl commands to the pass-through driver,
scsi_ctl. The pass-through driver is described in scsi_ctl(7).

The following sections provide the suggested steps for developing a SCSI
driver:

- “SCSI Driver Development, Step 1: Include Header Files”
- “SCSI Driver Development, Step 2: Set Up Structures”
- “SCSI Driver Development, Step 3: Create the driver_install Routine”
- “SCSI Driver Development, Step 4: Create the driver_dev_init() Routine”
- “SCSI Driver Development, Step 5: Analyze Multiprocessor Implications”
- “SCSI Driver Development, Step 6: Create the Entry-Point Routines”
- “SCSI Driver Development, Step 7: Error Handling”
- “SCSI Driver Development, Step 8: Underlying Routines”

The examples in this chapter assume that the name of your driver is
mydriver and that you are following the routine-naming conventions
described in “Step 1: Choosing a Driver Name” on page 76, Chapter 5.
SCSI Driver Development, Step 1: Include Header Files

See reference pages for each kernel call and data structure your driver uses to find out which headers your driver requires.

**NOTE**
Including header files that your driver does not need increases compile time and the likelihood of encountering portability problems. It is not recommended.

**General Header Files**

- `/usr/include/sys/buf.h` I/O buf structure, `buf`.
- `/usr/include/sys/errno.h` Defines errors returned to applications.
- `/usr/include/sys/file.h` Defines open flags
- `/usr/include/sys/io.h` `isc` table structure.
- `/usr/include/sys/alloc.h` Necessary for acquiring and releasing memory.
- `/usr/include/sys/wsio.h` WSIO context data and macro definitions.

**Header Files for SCSI Drivers**

- `/usr/include/sys/scsi.h` SCSI-specific data definitions and ioctl commands.
- `/usr/include/sys/scsi_ctl.h` SCSI subsystem data and macro definitions.

**Header Files for Device Classes**

In addition to the header file created for the specific driver, your driver may need other, device-class-specific files.

- `/usr/include/sys/diskio.h` Data definitions for disk ioctl commands (`DIOC_xxx`). Includes `/usr/include/sys/types.h` and
Writing SCSI Device Drivers

SCSI Driver Development, Step 1: Include Header Files

/usr/include/sys/ioctl.h.

/usr/include/sys/floppy.h  Data definitions for floppy ioctl commands.
/usr/include/sys/mtio.h  Data definitions for magnetic tape ioctl commands.
SCSI Driver Development, Step 2: Set Up Structures

Depending on the characteristics of the driver, you can set it up as a character driver, a block driver, or (as in the case of disk drivers) both.

**NOTE**

Whether the driver is to operate on an MP platform or not, SCSI Services makes use of the locking facilities, and all drivers using SCSI Services must use the provided data-protection routines. It is essential that you include the C_ALLCLOSES and C_MGR_IS_MP flags in the `drv_ops_t` structure and the DRV_MP_SAFE flag in the `drv_info_t` structure. See “SCSI Driver Development, Step 5: Analyze Multiprocessor Implications” for more information.

Determine the driver's name and device class, and put this information in the appropriate structures. (See “Step 3: Defining Installation Structures” on page 80, Chapter 5, for information about these data structures.)

First, declare your driver's routines that can be called by the kernel. These are used in the following structure.

```c
int mydriver_open();
int mydriver_close();
int mydriver_strategy();
int mydriver_psize();
int mydriver_read();
int mydriver_write();
int mydriver_ioctl();
```

The `drv_ops_t` structure specifies the “external” driver routines to the kernel. The C_ALLCLOSES and C_MGR_IS_MP flags are required by SCSI Services. See “The `drv_ops_t` Structure Type” on page 80, Chapter 5, for further details.

```c
static drv_ops_t mydriver_ops =
{
    mydriver_open,
    mydriver_close,
    mydriver_strategy,
    NULL,
};
```
The `drv_info_t` structure specifies the driver's name (`mydriver`) and class (`disk`). Flags define the driver type. The `DRV_MP_SAFE` flag is required by SCSI Services. See “The `drv_info_t` Structure Type” on page 83, Chapter 5, for further details.

```c
static drv_info_t mydriver_info =
{
    "mydriver",
    "disk",
    DRV_CHAR | DRV_BLOCK | DRV_SAVE_CONF | DRV_MP_SAFE,
    -1,
    -1,
    NULL,
    NULL,
    NULL,
    C_ALLCLOSES | C_MGR_IS_MP
};
```

The `wsio_drv_data_t` structure specifies additional information for the WSIO CDIO. The first field should be `scsi_disk` for SCSI device drivers and `scsi` for SCSI interface drivers. See “The `wsio_drv_data_t` Structure Type” on page 85, Chapter 5, for further details.

```c
static wsio_drv_data_t mydriver_data =
{
    "scsi_disk",
    T_DEVICE,
    DRV_CONVERGED,
    NULL,
    NULL,
};
```

The `wsio_drv_info_t` structure ties the preceding three together. See “The `wsio_drv_info_t` Structure Type” on page 86, Chapter 5, for further details.
details.

```c
static wsio_drv_info_t mydriver_wsio_info = {
    &mydriver_info,
    &mydriver_ops,
    &mydriver_data,
};
```
SCSI Driver Development, Step 3: Create the driver_install Routine

The `driver_install` routine causes the information that you created above to be installed into the I/O subsystem, specifically into the WSIO CDIO.

```c
int (*mydriver_saved_dev_init)();

int mydriver_install()
{
    extern int (*dev_init)();

    mydriver_saved_dev_init = dev_init;
    dev_init = mydriver_dev_init;

    /* register driver with WSIO and return any error */
    return(wsio_install_driver(&mydriver_wsio_info));
}
```
SCSI Driver Development, Step 4: Create the driver_dev_init() Routine

You specify the `driver_dev_init` routine from the `driver_install()` routine. The `driver_dev_init` routine calls `scsi_ddsw_init()`, which initializes some fields in the SCSI driver's device-switch table (`scsi_ddsw`). This table is independent of the kernel's device switch tables.

```c
mydriver_dev_init()
{
    dev_t dev = NODEV;
    /*
    * Initialize mydriver_ddsw.blk_major and
    * mydriver_ddsw.raw_major.
    */
    scsi_ddsw_init(mydriver_open, &mydriver_ddsw);
    (*mydriver_saved_dev_init)();
}
```

Setting up the Device-Switch Table (scsi_ddsw)

In order to use SCSI Services effectively, a SCSI driver must define its `scsi_ddsw` device-switch structure. This structure contains pointers to special `dd` routines, some of which are executed indirectly by the standard driver routines, such as `driver_read`. The structure is passed to SCSI Services routines from the `driver_open` routine, which calls the `scsi_lun_open()` SCSI Services routine.

SCSI Services has been set up to control the housekeeping and other processing in the SCSI interface. Therefore, you should have the standard driver routines restrict their operation to calling the appropriate SCSI Services routine, as shown in the examples in “SCSI Driver Development, Step 6: Create the Entry-Point Routines”. Special processing and customization should all be handled in the special `dd` routines.

For a summary of SCSI Services, see “SCSI Services Summary”. For more detailed information, see the HP-UX Driver Development Reference.

The `scsi_ddsw` structure is defined as follows in the header file `<sys/scsi_ctl.h>`:

---

Chapter 9  323
SCSI Driver Development, Step 4: Create the driver_dev_init() Routine

```c
struct scsi_ddsw {
    u_char        blk_major;
    u_char        raw_major;
    int           dd_lun_size;
    int           (*dd_open)();
    void          (*dd_close)();
    int           (*dd_strategy)();
    int           (*dd_read)();
    int           (*dd_write)();
    int           (*dd_ioctl)();
    struct buf    (*dd_start)();
    int           (*dd_done)();
    int           (*dd_pass_thru_okay)();
    int           (*dd_pass_thru_done)();
    int           (*dd_ioctl_okay)();
    struct status_action *dd_status_list;
    int           dd_status_cnt;
    ubit32        dd_flags;
    wsio_drv_info_t *wsio_drv;
};
```

The entries are described below.

- **blk_major**
  - Block and character major numbers; specify them as NODEV. They are initialized by `scsi_ddsw_init()` when it is called from your `driver_dev_init()` routine.

- **raw_major**
  - Block and character major numbers; specify them as NODEV. They are initialized by `scsi_ddsw_init()` when it is called from your `driver_dev_init()` routine.

- **dd_lun_size**
  - The number of bytes to be allocated and attached to the open device tree when `driver_open()` is first executed.

- **dd_open()**
- **dd_close()**
- **dd_strategy()**
- **dd_read()**
- **dd_write()**
- **dd_ioctl()**
  - Pointers to underlying driver-specific routines. When the corresponding `driver_routine` is called by the kernel and transfers control to SCSI Services, SCSI Services performs certain overhead operations and calls these routines for driver-specific operations.

- **dd_start()**
  - Driver specific start routine

- **dd_done()**
  - Driver specific post I/O processing
Writing SCSI Device Drivers

SCSI Driver Development, Step 4: Create the driver_dev_init() Routine

dd_pass_thru_okay()  Driver specific control of pass through I/O

dd_pass_thru_done()  Driver specific notation of pass through I/O

dd_ioctl_okay()  Disallow ioctl commands through the pass through driver

dd_flags  Flag bits, currently only DD_DD defined.

Here is an example of an initialized declaration of the scsi_ddsw:

The first example is the declaration of your driver’s version of the dd routines that can be called by SCSI Services. The routine names are arbitrary. The names in comments are the field names of the scsi_ddsw structure.

```c
int mydriver_dd_open(); /* dd_open */
void mydriver_dd_close(); /* dd_close */
int mydriver_dd_strategy(); /* dd_strategy */
int mydriver_dd_read(); /* dd_read */
int mydriver_dd_write(); /* dd_write */
int mydriver_dd_ioctl(); /* dd_ioctl */
struct buf mydriver_dd_start(); /* dd_start */
int mydriver_dd_done(); /* dd_done */
int mydriver_dd_pass_thru_okay(); /* dd_pass_thru_okay */
int mydriver_dd_pass_thru_done(); /* dd_pass_thru_done */
int mydriver_dd_ioctl_okay(); /* dd_ioctl_okay */
```

The following example shows the scsi_ddsw structure. Specify NULL for routines that are not defined (that is, that you are not using). The first two fields specify the block and character major numbers; they are filled in by the call in driver_dev_init() to the SCSI Services routine scsi_ddsw_init(). The last field points to the wsio_drv_info_t structure that you defined in “SCSI Driver Development, Step 2: Set Up Structures”. The first name in each comment is the field name of the scsi_ddsw structure element.

```c
struct scsi_ddsw mydriver_ddsw =
{
    NODEV, /* blk_major - mydriver_dev_init sets */
    NODEV, /* raw_major - mydriver_dev_init sets */
    sizeof(struct mydriver_lun), /* dd_lun_size */
    mydriver_dd_open, /* dd_open */
    mydriver_dd_close, /* dd_close */
    mydriver_dd_strategy, /* dd_strategy */
    NULL, /* dd_read */
    NULL, /* dd_write */
    mydriver_dd_ioctl, /* dd_ioctl */
};
```
Writing SCSI Device Drivers

SCSI Driver Development, Step 4: Create the driver_dev_init() Routine

mydriver_dd_start,    /* dd_start */
mydriver_dd_done,     /* dd_done */
mydriver_dd_pass_thru_okay, /* dd_pass_thru_okay */
mydriver_dd_pass_thru_done, /* dd_pass_thru_done */
mydriver_dd_ioctl_okay, /* dd_ioctl_okay */
mydriver_dd_status_list, /* dd_status_list */
    sizeof(mydriver_dd_status_list)/
    sizeof(mydriver_dd_status_list[0]),
    /* dd_status_cnt */
mydriver_dd_flags,     /* dd_flag bits DD_DDG */
    &mydriver_wsio_info    /* For Diagnostics Logging;
                            NULL means errors print in dmesg */
};
SCSI Driver Development, Step 5: Analyze Multiprocessor Implications

You need to make your device driver MP safe, regardless of whether it is to operate an MP platform or not. SCSI Services make use of the kernel’s locking facilities, so all drivers that use SCSI Services must use the data-protection routines the kernel provides.

Your drivers must do the following:

• Set the `C_MGR_IS_MP` flag in the `d_flags` field of the driver's `drv_ops_t` structure.

• Set the `DRV_MP_SAFE` flag in the `flags` field of the `drv_info_t` structure.

• Use the driver semaphore, driver lock, LUN lock, and target lock as necessary to provide MP protection. Refer to the `defines` and `structures` in `/usr/include/sys/scs_ctl.h` for details. This is the largest task, and involves looking at the code and determining whether there are data references that must be protected and which locks and semaphores must be used to protect the references. (See “Data Protection for SCSI Drivers” for more details.)

• Build a kernel with your driver.

• Test your driver on a single processor (UP) system with a debug kernel if available. (You can also test it on an MP system.)
SCSI Driver Development, Step 6: Create the Entry-Point Routines

For many of the entry points, SCSI Services perform much of the work. If you use \texttt{physio()}, \texttt{scsi\_strategy()} will be called by your driver's \texttt{driver\_strategy} routine. Hence, you need not create the underlying \texttt{ddsw->dd\_read()} and \texttt{ddsw->dd\_write()} routines. However, if your driver calls \texttt{scsi\_strategy()}, you must specify a \texttt{ddsw->dd\_strategy()} routine.

The \texttt{scsi\_strategy()} routine cannot block because it can be called on the Interrupt Control Stack (ICS) by a \texttt{bp->b\_call} routine.

\textbf{driver\_open()} Routine

\begin{verbatim}
mydriver\_open(dev, oflags)
dev_t dev;
int oflags;
{
    return (scsi\_lun\_open(dev, &mydriver\_ddsw, oflags));
}
\end{verbatim}

\textbf{driver\_close()} Routine

\begin{verbatim}
mydriver\_close(dev)
dev_t dev;
{
    return scsi\_lun\_close(dev);
}
\end{verbatim}

\textbf{driver\_read()} Routine

\begin{verbatim}
mydriver\_read(dev, uio)
dev_t dev;
struct uio *uio;
{
    return scsi\_read(dev, uio);
}
\end{verbatim}
Writing SCSI Device Drivers

SCSI Driver Development, Step 6: Create the Entry-Point Routines

Chapter 9 329

driver_write() Routine

mydriver_write(dev, uio)
dev_t dev;
struct uio *uio;
{
    return scsi_write(dev, uio);
}

driver_strategy() Routine

The driver_strategy() routine does not return anything. It records errors in bp->b_error.

mydriver_strategy(bp)
struct buf *bp;
{
    scsi_strategy(bp);
}

driver_psize() Routine

This example assumes that driver_psize() is never called when the device is closed. Note the use of the SCSI Services m_scsi_lun() function.

mydriver_psize(dev)
dev_t dev;
{
    struct scsi_lun *lp = m_scsi_lun(dev);
    struct mydriver_lun *llp = lp->dd_lun;
    int rshift, nblks, size;

    nblks = llp->nblks;
    rshift = llp->devb_lshift;

    size = rshift > 0 ? nblks >> rshift : nblks << -rshift;

    return size;
}

driver_ioctl() Routine

mydriver_ioctl(dev, cmd, data, flags)
dev_t dev;
int    cmd;
int    *data;
int    flags;
{
    return scsi_ioctl(dev, cmd, data, flags);
}
SCSI Driver Development, Step 7: Error Handling

You can specify one optional list in the driver’s scsi_ddsw: dd_status_list[]. SCSI services access this optional list when an I/O completion occurs on the driver’s SCSI LUN. The SCSI Services internal routine scsi_status_action() determines what to do based upon this list.

The following are examples of very simple lists:

```c
struct sense_action mydriver_sense_list[] = {
    { S_GOOD, S_CURRENT_ERROR, S_RECOVERRED_ERROR,
      SA_ANY, SA_ANY, mydriver_check_residue, SA_DONE |
      SA_LOG_IT_ALWASY, 0 },
    { SA_ANY, SA_ANY, SA_ANY, SA_ANY, SA_ANY,scsi_action,
      SA_DONE + SA_LOG_IT_NEVER, EIO }
};

struct status_action mydriver_status_list[] = {
    { S_GOOD, scsi_action, SA_DONE + SA_LOG_IT_NEVER, 0 },
    { S_CHECK_CONDITION, scsi_sense_action, (int)
      mydriver_sense_list,sizeof(mydriver_sense_list) / 
      sizeof(mydriver_sense_list [0]) },
    { S_CONDITION_MET, scsi_action, SA_DONE + 
      SA_LOG_IT_NEVER, 0 },
    { S_CONDITION_MET, scsi_action, SA_DONE + 
      SA_LOG_IT_NEVER, 0 },
    { S_I_CONDITION_MET, scsi_action, SA_DONE + 
      SA_LOG_IT_NEVER,0 },
    { SA_ANY, scsi_action, SA_DONE + SA_LOG_IT_ALWAYS, EIO }
};
```

Your driver can specify its own routines for handling errors, and can break down errors for more granularity. You can access the Pass-Thru Driver status using the driver’s dd_pass_thru_done() routine, described in “SCSI Driver Development, Step 8: Underlying Routines”.

SCSI Driver Development, Step 8: Underlying Routines

This is where the driver can be as complex as you desire, or as the device requires. The `scsi_lun_open()` routine ensures that the bus, target, and LUN of the driver’s device are open and able to handle I/O. Specific requirements for the device itself should be addressed in the driver’s `ddsw->dd_open()` routine. The same principle applies for `close`, `read`, `write`, and so on.

The call graph in Figure 9-1, “Call Graph of SCSI Routines and Services,” shows how these underlying routines and SCSI services call each other. For a summary list of SCSI Services, see “SCSI Services Summary”. Detailed information on SCSI Services is provided in the HP-UX Driver Development Reference.
Figure 9-1 Call Graph of SCSI Routines and Services

dd_close Routine

The *dd_close()* SCSI function, used to provide driver-specific processing during close, is provided by the driver writer. It can have any unique name. You pass the name to SCSI Services by specifying it in the *dd_close* field of the *scsi_ddsw* structure.

If this routine is defined in the *scsi_ddsw* structure, it is called to perform the actual device close processing. For example, for the *scsi_disk* driver, the *sd_close()* function performs the Test Unit
Ready and Allow Media Removal commands.

**Conditions**

`dd_close()` is called from `scsi_lun_close()` in a process context. The open/close lun semaphore is held when the `dd_close()` function is called. `dd_close()` is not called from within a critical section; it may block.

**Declaration**

```c
void dd_close (
    dev_t  dev
);
```

**Parameters**

- `dev` The device number.

**Return Values**

`dd_close()` does not return a value.

**Example**

```c
#include <sys/scsi_ctl.h>
#define ST_GEOM_LOCKED 0x00000002

void mydriver_dd_close(dev);
dev_t  dev;
{
    struct scsi_lun  *lp = m_scsi_lun(dev);
    struct mydriver_lun  *llp = lp->dd_lun;
    if (dd_blk_open_cnt(lp) == 1) {
        scsi_lun_lock(lp);
        llp->state &= ~ST_GEOM_LOCKED;
        scsi_lun_unlock(lp);
    }
}
```

**dd_ioctl Routine**

The `dd_ioctl()` routine is provided by the driver writer. It can have any unique name. You pass the name to SCSI Services by specifying it in the `dd_ioctl` field of the `scsi_ddsw` structure.
If this routine exists in the scsi_ddsw structure, it is called by scsi_ioctl() if the ioctl command remains unsatisfied by the choices provided within that SCSI Services procedure. If dd_ioctl() does not exist when called, scsi_ioctl() returns an error.

Examine the ioctl commands provided by SCSI Services in scsi_ioctl(), and implement any additional commands needed in your dd_ioctl() routine.

It is in dd_ioctl() and in dd_open(), if implemented, that some of the more specialized features of SCSI Services may be useful, as listed below.

- scsi_cmd()
- scsi_init_inquiry_data()
- scsi_mode_sense()
- scsi_mode_fix()
- scsi_mode_select()
- scsi_wr_protect()

Conditions

dd_ioctl() is called from scsi_ioctl() in a process context. It is not called from within a critical section; it may block.

Declaration

```c
int dd_ioctl (dev_t dev,
              int cmd,
              caddr_t data,
              int flags);
```

Parameters

- `cmd` The command word
- `data` Pointer to the commands arguments
- `dev` The device number
- `flags` The file-access flags
Writing SCSI Device Drivers

SCSI Driver Development, Step 8: Underlying Routines

Return Values

dd_ioctl() is expected to return the following values:

0  Successful completion.

<>0  Error. Value is expected to be an errno.

Example

```
#include <sys/scsi.h>
#include <sys/scsi_ctl.h>

mydriver_dd_ioctl (dev_t dev, int cmd, int *data, int flags);
{
    struct scsi_lun *lp = m_scsi_lun(dev);
    struct mydriver_lun *llp = lp->dd_lun;
    struct scsi_tgt *tp = lp->tgt;
    struct scsi_bus *busp = tp->bus;
    struct inquiry_2 *inq = &lp->inquiry_data.inq2;
    disk_describe_type *ddt;
    int size = (cmd & IOCSIZE_MASK) >> 16;
    int i;

    switch (cmd & IOCCMD_MASK)
    {
        case DIOC_DESCRIBE & IOCCMD_MASK:
            if (cmd != DIOC_DESCRIBE && size != 32)
                return EINVAL;
            ddt = (void *) data;
            i = inq->dev_type;
            bcopy(inq->product_id, ddt->model_num, 16);
            ddt->intf_type = SCSI_INTF;
            ddt->maxsva = llp->nblks - 1;
            ddt->lgblksz = llp->blk_sz;
            ddt->dev_type = i == SCSI_DIRECT_ACCESS ? DISK_DEV_TYPE
            : i == SCSI_WORM ? WORM_DEV_TYPE
            : i == SCSI_CDROM ? CDROM_DEV_TYPE
            : i == SCSI_MO ? MO_DEV_TYPE
            : UNKNOWN_DEV_TYPE;

            if (HP_MO(lp))
                /* Shark lies; fix it to match Series800 */
```
Scsi Driver Development, Step 8: Underlying Routines

```
        ddt->dev_type = MO_DEV_TYPE;
        if (size == 32)
            return 0;
        /* WRITE_PROTECT for SCSI WORM */
        ddt->flags = (llp->state & LL_WP) ? WRITE_PROTECT_FLAG : 0;
        return 0;
    }

    switch (cmd)
    {
    case SIOC_CAPACITY:
        ((struct capacity *) data)->lba = llp->nblks;
        ((struct capacity *) data)->blksz = llp->blk_sz;
        return 0;
    case SIOC_GET_IR:
        return mydriver_wce(dev, SIOC_GET_IR, data);
    case SIOC_SET_IR:
        if (!(flags & FWRITE) && !suser())
            return EACCES;
        if (*data & ~0x1)
            return EINVAL;
        return mydriver_wce(dev, SIOC_SET_IR, data);
    case SIOC_SYNC_CACHE:
        if (llp->state & LL_IR)
            return 0; /* IR not on, just return */
        else
            return mydriver_sync_cache(dev);
    case DIOC_CAPACITY:
        *data = (1llp->devb_lshift > 0 ? llp->nblks >> llp->devb_lshift
                          : llp->nblks << -(llp->devb_lshift));
        return 0;
    ...
    default:
        return EINVAL;
    }
```

**dd_ioctl_okay Routine**

The **dd_ioctl_okay()** SCSI function is provided by the driver writer. It
Writing SCSI Device Drivers

SCSI Driver Development, Step 8: Underlying Routines

can have any unique name. You pass the name to SCSI Services by specifying it in the dd_ioctl_okay field of the scsi_ddsw structure.

\texttt{dd_ioctl_okay()} disallows all ioctl commands through the pass-through driver that are not explicitly allowed by any nonpass-through driver that has the device open concurrently.

\textbf{Conditions}

\texttt{dd_ioctl_okay()} is called from \texttt{sctl_ioctl()} in a process context. It is called within a critical section; it may not block.

\begin{center}
\textbf{NOTE}
\end{center}

\textit{It is desirable to allow SIOC_INQUIRY for the pass-through driver at all times. Therefore, SIOC_INQUIRY is allowed by default (if there is no dd_ioctl_okay() routine). SIOC_INQUIRY is also always allowed if it will not result in I/O (lp->inquirySz > 0), because it does not affect the nonpass-through device driver in any way.}

\begin{center}
\textbf{Declaration}
\end{center}

\begin{verbatim}
int dd_ioctl_okay (  
    dev_t dev,  
    int cmd,  
    caddr_t data,  
    int flags  
);
\end{verbatim}

\textbf{Parameters}

\begin{description}
\item[cmd] The command word
\item[data] Pointer to the commands arguments
\item[dev] The device number
\item[flags] The file-access flags
\end{description}

\textbf{Return Values}

\texttt{dd_ioctl_okay()} is expected to return the following values:

\begin{description}
\item[PT_OKAY] Successful completion.
\item[0] Error.
\end{description}
Examples

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

mydriver_dd_ioctl_okay (
    dev_t  dev,
    int    cmd,
    void    *data,
    int    flags
);
{
    return PT_OKAY;
}
```

**dd_open Routine**

The `dd_open()` SCSI function is provided by the driver writer. It can have any unique name. You pass the name to SCSI Services by specifying it in the `dd_open` field of the `scsi_ddsw` structure.

If this routine exists in the `scsi_ddsw` structure, it is called to perform the actual device “open” processing.

As an example the disk driver's `sd_open()` calls `disksort_init_queue()` for the lun's `lun_disk_queue`. It calls `scsi_init_inquiry_data()` to set the target state for SDTR and WDTR and send the Start Unit, Test Unit Ready, Prevent Media Removal, and Read Capacity commands, if appropriate, for the type of disk being opened.

This routine can be as complicated as you need to ensure the device is properly open the first time (ensured by checking `dd_open_cnt`). Calling the SCSI Service `scsi_init_inquiry_data()` is reasonable, as is performing Test Unit Ready. Changing state in the `scsi_lun` or target structures requires protection.

**Conditions**

`dd_open()` is called from `scsi_lun_open()` in a process context. The open/close lun Semaphore is held when `dd_open()` is called. `dd_open()` is not called within a critical section; it may block.

**Declaration**

```c
dd_open (      
    dev_t  dev,
```
int oflags
)

**Parameters**

*dev*  
The device number

*oflags*  
The flags passed in the open call

**Return Values**

dd_open() is expected to return the following values:

0  
Successful completion.

<0  
Error. The value is expected to be an *errno* value.

**Examples**

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

mydriver_dd_open(dev, oflags)
dev_t dev;
int oflags;
{
    struct scsi_lun *lp = m_scsi_lun(dev);
    struct mydriver_lun *llp = lp->dd_lun;
    struct scsi_tgt *tp = lp->tgt;
    struct inquiry_2 *inq = &lp->inquiry_data.inq2;
    struct capacity cap;
    u_char cdb[12];
    struct sense_hdr *hd;
    struct block_desc *bd;
    struct caching_page *c_pd;
    struct error_recovery *e_pd,
    int ret_size, bpb, error, x;

    /*
    * Only first opens are interesting.
    */
    if (dd_open_cnt(lp) > 1)
        return 0;

    ...

    /*
    * Inquiry.
    * 
    * Call the routine provided by services to do any
```
necessary synchronization with the pass-through
driver. Success here does not imply that there is no
more pending sense data. In fact, the SCSI-2
standard encourages devices not to give Check
Condition status on Inquiry, but to defer it until
a subsequent command. Also, if the inquiry data had
already been cached as a result of a pass-through
driver open or SIOC_INQUIRY, this may not even
result in I/O.

```c
if (error = scsi_init_inquiry_data(dev))
    return error;
```

/*
* Needs protection at LUN and Tgt.
*/
```
scsi_lun_lock(lp);
scsi_tgt_lock(tp);

tp->state |= T_ENABLE_SDTR;

scsi_tgt_unlock(tp);
scsi_lun_unlock(lp);
```
bzero(cdb, sizeof(cdb));
cdb[0] = CMDtest_unit_ready;
if (scsi_cmd(dev, SCB_DONT_PRINT, 6, cdb, 0, 0,
llp->mydriver_msecs, 0,&error))
{
    /*
    * Allow an incomplete open if this is a raw device.
    */
    if (major(dev) == mydriver_ddsw.raw_major)
    {
        scsi_lun_lock(lp);
        lp->state |= L_DISABLE_OPENS;
        scsi_lun_unlock(lp);
        return 0;
    }
```
Writing SCSI Device Drivers

SCSI Driver Development, Step 8: Underlying Routines

... return error;
...

**dd_pass_thru_done Routine**

The `dd_pass_thru_done()` routine is provided by the driver writer. It can have any unique name. You pass the name to SCSI Services by specifying it in the `dd_pass_thru_done` field of the `scsi_ddsw` structure.

If this routine exists in the `scsi_ddsw` structure, SCSI Services executes it on completion of a pass-through I/O. It allows the device driver to make note of any I/Os which have occurred and any resulting status and/or sense data.

The `dd_pass_thru_done()` function is called from within a critical section; it is not permitted to block.

**Declaration**

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

**Parameters**

- `bp` buf structure

**Return Values**

`dd_pass_thru_done()` is declared as returning `int`; however, the return is not used by SCSI services.

**dd_pass_thru_okay Routine**

The `dd_pass_thru_okay()` routine is provided by the driver writer. It can have any unique name. You pass the name to SCSI Services by specifying it in the `dd_pass_thru_okay` field of the `scsi_ddsw` structure.

If a device is opened by a nonpass-through device driver and the driver specifies a `dd_pass_thru_okay()` entry point in its `scsi_ddsw` structure, then the driver has complete control over what pass-through I/Os are allowed. If the driver does not specify a `dd_pass_thru_okay()` entry point, then pass-through I/Os are not allowed.
The `dd_pass_thru_okay()` function is called from within a critical section and may not block.

**Declaration**

```c
dd_pass_thru_okay (dev_t dev, struct sctl_io *sctl_io);
```

**Parameters**

- `dev` The device number
- `sctl_io` Struct containing ioctl information

**Return Values**

`dd_pass_thru_okay()` is expected to return the following values:

- `PT_OKAY` Successful completion.
- `0` Error.

**Example**

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

mydriver_dd_pass_thru_okay(dev, sctl_io)
dev_t dev;
struct sctl_io *sctl_io;
{
    return PT_OKAY;
}
```

**dd_read Routine**

The `dd_read()` routine is provided by the driver writer. It can have any unique name. You pass the name to SCSI Services by specifying it in the `dd_read` field of the `scsi_ddsw` structure.

If this routine exists in the `scsi_ddsw` structure, it is called instead of `physio()` by `scsi_read()`.

`dd_read()` is called in a process context. It is not called from within a critical section; it may block.
**Declaration**

```c
int dd_read (  
    dev_t    dev,  
    struct uio *uio  
);
```

**Parameters**

- `dev` The device number
- `uio` Structure containing transfer information

**Return Values**

`dd_read()` is expected to return the following values:

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

**Example**

```c
mydriver_dd_read(dev, uio)
dev_t dev;
struct uio *uio;
{
    struct scsi_lun *lp = m_scsi_lun(dev);
    struct sf_lun *llp = lp->dd_lun;
    int error;

    scsi_lun_lock(lp);
    while (llp->state & ST_GEOM_SEMAPHORE)
        scsi_sleep(lp, &llp->state, PRIBIO);
    lp->state |= ST_GEOM_SEMAPHORE;
    scsi_lun_unlock(lp);

    sf_update_geometry(dev);
    error = physio(scsi_strategy, NULL, dev, B_READ, minphys, uio);

    scsi_lun_lock(lp);
    llp->state &= ~ST_GEOM_SEMAPHORE;
    scsi_lun_unlock(lp);
    wakeup(&llp->state);

    return error;
```
dd_start Routine

The `dd_start()` routine is provided by the driver writer. It can have any unique name. You pass the name to SCSI Services by specifying it in the `dd_start` field of the `scsi_ddsw` structure.

If this routine exists in the `scsi_ddsw` structure, it is called by `scsi_start()` to allow the driver to perform any necessary processing prior to calling `scsi_start_nexus()`.

The `dd_start()` function is called in the process and interrupt context from within a critical section in `scsi_start()`. `dd_start()` is not permitted to block.

The critical section in `scsi_start()`, from where the `dd_start()` function is called, is mainly protecting the `scsi_lun` structure and guaranteeing that `lp->n_scbs` is consistent with the `dd_start()` function starting a request or not. The critical section also protects the incrementing of `n_scbs` in the `scsi_tgt` structure and the incrementing of the SCSI subsystem unique I/O ID `scsi_io_cnt`.

If this routine does not exist, only “special” I/Os (B_SIOC_IO or B_SCSI_CMD) can be performed.

The driver's `dd_start()` routine must dequeue the I/O from the appropriate list and perform whatever is necessary for the device to operate upon the I/O.

The parameters passed for this purpose are the `lp` and the `scb` parameters. The `scb` has the necessary `cdb[]` array for the SCSI command bytes.

Declaration

```c
struct buf *(*d_start) dd_start (  
    struct scsi_lun *lp,  
    struct scb *scb  
);
```

Parameters

- `lp` The open LUN structure
- `scb` Extra state information for I/O
Return Values

`dd_start()` is expected to return the following values:

- `struct buf *bp` Successful completion.
- `NULL` Error.

Example

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

struct buf *mydriver_dd_start(lp, scb)
struct scsi_lun *lp;
struct scb *scb;
{
    struct mydriver_lun *llp = lp->dd_lun;
    struct buf *bp;
    struct scb *head_scb, *scb_forw, *scb_back;
    int nblks, blkno, x;
    int lshift = llp->devb_lshift;

    /*
     * We could be more granular with locks, but
     * that would most likely cause too much
     * overhead getting/releasing locks.
     */
    scsi_lun_lock(lp);
    if ((bp = mydriver_dequeue(lp)) == NULL)
    {
        goto start_done;
    }
    nblks = bp->b_bcount >> llp->log2_blk_sz;
    if (bp->b_offset & DEV_BMASK)
        blkno = (unsigned) bp->b_offset >> llp->log2_blk_sz;
    else
        blkno = (unsigned) (lshift > 0
            ? bp->b_blkno << lshift
            : bp->b_blkno >> -lshift);

    scb->cdb[0] = (bp->b_flags & B_READ)
        ? CMDread10
        : llp->state & LL_WWV;
```
/* CMDwriteNverify */
: CMDwrite10;
scb->cdb[1] = 0;
scb->cdb[2] = blkno >> 24;
scb->cdb[3] = blkno >> 16;
scb->cdb[4] = blkno >> 8;
scb->cdb[5] = blkno;
scb->cdb[6] = 0;
scb->cdb[7] = nblks >> 8;
scb->cdb[8] = nblks;
scb->cdb[9] = 0;

/* Immediate Reporting (WCE) ON? */
if (llp->state & LL_IR)
  if ((scb->cdb[0] == CMDwrite10) && (bp->b_flags & B_SYN)
    if (llp->active_bp_list != NULL)
      scb->io_forw = llp->active_bp_list;
      head_scb = (void *) llp->active_bp_list->b_scb;
      scb->io_back = head_scb->io_back;
Writing SCSI Device Drivers

SCSI Driver Development, Step 8: Underlying Routines

```c
scb_forw = (void *) scb->io_forw->b_scb;
scb_back = (void *) scb->io_back->b_scb;
scb_forw->io_back = bp;
scb_back->io_forw = bp;
llp->active_bp_list = bp;
}
else
{
    llp->active_bp_list = bp;
    scb->io_forw = scb->io_back = bp;
}
/* Although redundant with caller, set this in case
* completion int */
    bp->b_scb = (long) scb;
start_done:
    scsi_lun_unlock(lp);
    return bp;
}

dd_strategy Routine

The dd_strategy() routine is provided by the driver writer. It can have any unique name. You pass the name to SCSI Services by specifying it in the dd_strategy field of the scsi_ddsw structure.

The dd_strategy() routine is called by scsi_strategy() to perform whatever sorting or queueing the device driver requires for normal I/O. For most drivers, enqueuing to lp->scb_q is necessary; the scsi_disk() driver calls disksort_enqueue().

dd_strategy() is called in a process (and possibly, interrupt) context; it is not allowed to block.

If the driver invokes scsi_strategy(), dd_strategy() is required. If the dd_read() or dd_write() routines are not specified, SCSI Services will assume physio() is to be used.

NOTE

scsi_strategy() calls dd_strategy() holding lun_lock.

```
Declaration

```c
int (*dd_strategy) dd_strategy (  
    struct buf *bp,  
    struct scsi_lun *lp  
);
```

Parameters

- `bp`: transfer buf header
- `lp`: scsi LUN information

Return Values

`dd_strategy()` is expected to return the following values:

- `0`: Successful completion.
- `-1`: Error.

Example

The MP protection is provided for modification of the queues. Here is an example for a tape:

```c
mydriver_dd_strategy(bp)  
struct buf *bp;  
{  
    struct scsi_lun *lp = m_scsi_lun(bp->b_dev);  
    struct st_lun *llp = lp->dd_lun;  
    struct st_static_lun *sllp = llp->static_data;  
    DB_ASSERT(!(bp->b_flags & B_ERROR));  
    sllp->head_pos &= ~HEAD_FORWARD;  
    P_LOG(bp->b_dev, READ_WRITE, bp->b_bcount, "req_size", "Request size");  
    /* Check for valid request size in fixed block mode */  
    if (llp->block_size > 0 && bp->b_bcount % llp->block_size != 0)  
        {  
            NP_LOG(bp->b_dev, READ_WRITE, llp->block_size, "blk_size", "Not a multiple of block size");  
            bp->b_flags |= B_ERROR;  
            bp->b_error = ENXIO;  
            biodone(bp);  
```
A SCSI disk does not use the `lp->scb_q`. Instead, a service from the File System is used, `disksort()`. The following is an example of its use:

```c
def struct buf *bp;
{
    dev_t dev = bp->b_dev;
    struct scsi_lun *lp = m_scsi_lun(dev);
    struct mydriver_lun *llp = lp->dd_lun;

    ASSERT(!(bp->b_flags & B_ERROR));
    if (bpcheck(bp, llp->nblks, llp->log2_blk_sz, 0))
        return -1;

    LOG(bp->b_dev, FUNC_QUEUE, bp->b_blkno, "b_blkno");
    LOG(bp->b_dev, FUNC_QUEUE, bp->b_offset, "b_offset");
    LOG(bp->b_dev, FUNC_QUEUE, bp->b_bcount, "b_bcount");

    return mydriver_enqueue(lp, bp);
}
```

```c
def struct scsi_lun *lp;
struct buf *bp;
{
    int x;
    struct mydriver_lun *llp = lp->dd_lun;
    struct buf *dp;

    dp = &llp->lun_disk_queue;

    /* set B_FIRST to get queue preference */
    if (bp->b_flags & B_SPECIAL)
        bp->b2_flags |= B2_FIRST;
```
/* fake b_cylin 512K per cylinder */
bp->b_cylin = (bp->b_offset >> 19);
disksort_enqueue(dp, bp);

/* Increment counters within this protection */

scsi_enqueue_count(lp, bp);

return 0;
}

Warning

dd_strategy() must exist (be defined as non-NULL in the scsi_ddsw structure) if your driver calls scsi_strategy().

dd_write Routine

The dd_write() routine is provided by the driver writer. It can have any unique name. You pass the name to SCSI Services by specifying it in the dd_write field of the scsi_ddsw structure.

If this routine exists in the scsi_ddsw structure, it is called instead of physio() by scsi_write() .

This routine is called from scsi_write() in a process context. Since it is not called from within a critical section, it may block.

Declaration

int dd_write (dev_t dev, struct uio *uio);

Parameters

dev The device number
uio Structure containing transfer information

Return Values

dd_write() is expected to return the following values:

0 Successful completion.
errno: Error.

Example

```c
#include <sys/scsi_ctl.h>
#define ST_GEOM_SEMAPHORE 2

mydriver_dd_write(dev, uio)
dev_t dev;
struct uio *uio;
{
    struct scsi_lun *lp = m_scsi_lun(dev);
    struct sf_lun *llp = lp->dd_lun;
    int error;

    scsi_lun_lock(lp);
    while ((llp->state & ST_GEOM_SEMAPHORE))
        scsi_sleep(lp, &llp->state, PRIBIO);
    llp->state |= ST_GEOM_SEMAPHORE;
    scsi_lun_unlock(lp);

    sf_update_geometry(dev);
    error = physio(scsi_strategy, NULL, dev, B_WRITE, minphys, uio);

    scsi_lun_lock(lp);
    llp->state &= ~ST_GEOM_SEMAPHORE;
    scsi_lun_unlock(lp);
    wakeup(&llp->state);

    return error;
}
```
Data Protection for SCSI Drivers

The SCSI Services your driver calls take the appropriate locks to provide MP protection. One thing your driver must provide is protection for accessing its own private data and any data under the domain of the SCSI Services, such as `scsi_lun`, `scsi_tgt`, `scsi_bus`, or the SCSI subsystem's data. Locks are defined in `<sys/scsi_ctl.h>`.

Rules for Ordering Locks

The rules for ordering locks and semaphores help the kernel detect deadlocks in their use. When a thread of execution must hold more than one lock or semaphore, it must acquire them in increasing order. The order of locks and semaphores is, in ascending order:

1. LUN lock
2. Target lock
3. Bus lock
4. Subsystem lock

If a thread of execution must hold both the LUN lock and target lock at the same time, the ordering rules assert that the code must acquire the LUN lock before it acquires the target lock.

The spinlocks that are used to implement the LUN, target, bus, and subsystem locks are the normal HP-UX spinlocks.

While a thread of execution holds a lock, the processor's interrupt level is set to SPL6, preventing I/O devices from interrupting that processor. The spinlock associated with `spl*()` services (`spi_lock`) is of lower order than practically all other locks, so code protected by a spinlock cannot call a `spl*()` routine.

Subsystem Lock

The subsystem lock protects the SCSI subsystem's global data. Only SCSI Services access this data, so your driver should have no need for this lock.
Bus Lock

Each scsi_bus structure has a lock associated with it that protects many of the fields in the structure. Most drivers do not need to use the bus lock, because they ordinarily do not access the information maintained in the scsi_bus structure.

You should be aware that some HP device drivers access the B_EXCLUSIVE flag in the state field of the scsi_bus structure.

Target Lock

Each scsi_tgt structure has a lock associated with it that protects some of the fields in the structure. Device drivers can access the open_cnt, sctl_open_cnt, state, and bus fields in this structure. Device drivers may only modify the state field, and must do so under the protection of the target lock. The target lock can also be used to prevent the open_cnt, sctl_open_cnt, or state field from being modified while other conditions are checked or actions are performed.

LUN Lock

Each scsi_lun structure has a lock associated with it that protects the fields in the structure and in the dd_lun private data area. See the following section on the LUN structure to see which fields device drivers can access and modify, and which locks protect those fields.

For the driver_open() routine, the device driver does not have any of the locks available until after the kernel calls scsi_lun_open(), because scsi_lun_open() creates the scsi_bus, scsi_tgt, and scsi_lun structures.

For the driver_close() routine, the situation is similar. The locks are also available when the dd_close() routine is called. When scsi_lun_close() returns control to its caller, the locks are no longer available to your driver.
**SCSI Services Summary**

SCSI Services are commonly used SCSI functions that allow device and interface drivers to be much smaller and more supportable. In addition to providing most commonly used SCSI functions, WSIO SCSI Services also provides a supported pass-through mechanism. (See `scsi_ctl(7)` in the HP-UX Reference for information on pass-through.)

SCSI Services are summarized in Table 9-1, “SCSI Services.” For more detailed information on these services see the HP-UX Driver Development Reference.

<table>
<thead>
<tr>
<th>Service</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>m_scsi_lun()</td>
<td>Returns scsi_lun pointer corresponding to the dev_t parameter passed in.</td>
</tr>
<tr>
<td>disksort_enqueue()</td>
<td>Places I/O requests on queues maintained by SCSI Services.</td>
</tr>
<tr>
<td>scsi_dequeue()</td>
<td>Removes I/O requests from queues maintained by SCSI Services.</td>
</tr>
<tr>
<td>scsi_dequeue_bp()</td>
<td>Externally available to dequeue particular bp from circular list. Intended for use with LVM's B_PFTIMEOUT.</td>
</tr>
<tr>
<td>scsi_ddsw_init()</td>
<td>Called from device driver's <code>driver_dev_init()</code> routine. Causes initialization of <code>blk_major</code> and <code>raw_major</code> fields in the driver's switch table (ddsw).</td>
</tr>
<tr>
<td>scsi_lun_open()</td>
<td>Called from device driver's <code>driver_dev_init()</code> routine. Performs necessary open operations, including the invocation of the calling driver's <code>ddsw-&gt;dd_open()</code> routine.</td>
</tr>
</tbody>
</table>
Writing SCSI Device Drivers

SCSI Services Summary

Table 9-1  

<table>
<thead>
<tr>
<th>Service</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>scsi_init_inquiry_data()</td>
<td>Called from device driver's <code>ddsw-&gt;dd_open()</code> routine. Performs first SCSI Inquiry request to the device.</td>
</tr>
<tr>
<td>scsi_strategy()</td>
<td>The first place in the I/O path that all I/O requests have in common. Its primary purpose is to enqueue the bp to await the necessary resources to allow the request to be sent to the interface driver, and thus, the hardware.</td>
</tr>
<tr>
<td>scsi_read()</td>
<td>Synchronous read routine, which calls <code>physio()</code>.</td>
</tr>
<tr>
<td>scsi_write()</td>
<td>Synchronous write routine, which calls <code>physio()</code>.</td>
</tr>
<tr>
<td>scsi_ioctl()</td>
<td>Iocll commands that are supported by all drivers are implemented here to ensure consistency among drivers.</td>
</tr>
<tr>
<td>scsi_cmd(), scsi_cmdx()</td>
<td>For driver-generated I/O requests. It creates and builds a <code>sctl_io</code> and a <code>bp</code>, attaches the <code>sctl_io</code> to the <code>bp</code>, forwards the <code>bp</code> to the <code>scsi_strategy()</code> routine, and cleans up when the I/O is completed.</td>
</tr>
<tr>
<td>scsi_action()</td>
<td>Must ultimately be called after each I/O attempt completion (as in a retry situation). It may log errors to the <code>dmesg</code> buffer, retry the I/O, or disable tags.</td>
</tr>
</tbody>
</table>
scsi_sense_action() Interprets sense data for SCSI, CCS, or SCSI-2 compliance. It requires that the inquiry data for the device has been initialized by scsi_init_inquiry_data() before it can interpret it.

scsi_snooze() Performs a sleep without tying up the processor. Must not be called by a thread of execution that holds any lock. Currently, this routine is used only by scsi_disk to delay subsequent device access following Inquiry to a particular model of Quantum disk drive.

scsi_log_io() Records the I/O attempt and its results in the dmesg buffer.

<table>
<thead>
<tr>
<th>Service</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>scsi_sense_action()</td>
<td>Interprets sense data for SCSI, CCS, or SCSI-2 compliance. It requires that the inquiry data for the device has been initialized by scsi_init_inquiry_data() before it can interpret it.</td>
</tr>
<tr>
<td>scsi_snooze()</td>
<td>Performs a sleep without tying up the processor. Must not be called by a thread of execution that holds any lock. Currently, this routine is used only by scsi_disk to delay subsequent device access following Inquiry to a particular model of Quantum disk drive.</td>
</tr>
<tr>
<td>scsi_log_io()</td>
<td>Records the I/O attempt and its results in the dmesg buffer.</td>
</tr>
</tbody>
</table>
10 Developing Dynamically Loadable Kernel Modules
The Dynamically Loadable Kernel Module (DLKM) feature provides the means to add a kernel module to a running UNIX system without rebooting the system or rebuilding the kernel. This feature also makes it possible to dynamically remove a kernel module from the UNIX system when the module is no longer needed, thereby freeing system resources for other use.

The DLKM feature is available in HP-UX 11.0. Since device drivers constitute the majority of supported DLKM modules for HP-UX 11.0, this chapter places considerable emphasis on the writing and installation of loadable device drivers.
Dynamically Loadable Kernel Modules

This chapter is included primarily for software programmers who want to write DLKM modules and/or convert existing static (non-loadable) modules to the DLKM module format. It can also be used by system administrators who want to use the kernel configuration tool set to install, configure, and manage DLKM modules.

This chapter is organized into seven sections:

- **“Introduction” on page 362**, presents important terms and concepts essential to understanding the DLKM feature.
- **“Writing DLKM Modules” on page 367**, identifies and describes the file set that makes up a DLKM module, which consists of the module’s object file and other configuration files needed to install the module into the system.
- **“Initializing and Terminating DLKM Modules” on page 376**, describes the steps that each type of DLKM module must take to initialize and terminate itself.
- **“Kernel Configuration Tool Set” on page 387**, describes the tools used to install, configure, and manage DLKM modules.
- **“Development Process” on page 390**, describes the procedures for preparing, installing, and configuring a DLKM module; loading and unloading the module; and testing and debugging the module.
- **“Sample DLKM WSIO Class Driver” on page 409**, presents a complete skeleton of a DLKM Workstation Input/Output (WSIO) class driver in HP-UX.
Developing Dynamically Loadable Kernel Modules

Introduction

The DLKM feature not only provides the infrastructure to load kernel modules into a running kernel, but it also allows a kernel module to be statically linked into the kernel—the way all kernel modules were included in the kernel prior to HP-UX 11.0. Simply setting a flag in one of the module's configuration files determines whether a module is to be configured as dynamically loadable or statically linked.

This section begins by defining important terms and concepts essential to understanding the DLKM feature. Then, it describes the new way kernel modules are packaged in HP-UX 11.0 and identifies the types of kernel modules currently supported by the DLKM feature. Lastly, this section states the many advantages of writing kernel modules in the DLKM format.

Important Terms and Concepts

- **DLKM module**
  A kernel module written in the DLKM format; no configuration (dynamically loadable or statically linked) is implied.

- **Dynamically loadable module**
  A kernel module maintained in an individual object file but is not statically linked into the kernel. (See “Preparing a Statically Linked Module” on page 396 for comparison.) Dynamically loadable modules can be included in the kernel or excluded from the kernel dynamically, without having to relink the entire kernel or reboot the system. The loadable image generated during the configuration of such modules may be demand loaded or unloaded by the system administrator or auto loaded by the kernel.

  Dynamically loadable modules are DLKM modules configured as dynamically loadable.

- **Loadable module**
  Same as dynamically loadable module.

- **Static module**
  A kernel module written prior to HP-UX 11.0.
Developing Dynamically Loadable Kernel Modules

Introduction

- Statically linked module
  A kernel module maintained in an individual object file that is included or excluded from the kernel (`vmunix`), based on whether the feature it supports is required in the system. An included module remains linked into the kernel, while an excluded module remains unlinked. To add a new module or remove an existing module, you have to relink the entire kernel and reboot the system to cause the new kernel configuration to take effect.

Kernel modules written in the DLKM format can either be configured as dynamically loadable modules or as statically linked modules. Kernel modules written prior to HP-UX 11.0—static modules—can only be configured as statically linked modules.

New Module Packaging
As of HP-UX 11.0, each kernel module has its own master and system files, whereas prior to HP-UX 11.0, the kernel modules shared master files and had access to a single system file—the HP-UX system file (`/stand/system` by default). The HP-UX system file is still supported in HP-UX 11.0. This new way of packaging kernel modules together with the new way of writing module source code is what makes the DLKM feature possible. See “Writing DLKM Modules” on page 367.

NOTE
See the master (4) manpage for descriptions of the two kinds of master files, and the config (1M) manpage for a description of the HP-UX system file.

Kernel modules written prior to HP-UX 11.0—static modules—are still fully supported in HP-UX 11.0. You are encouraged to re-package your static modules according to the module packaging architecture introduced in HP-UX 11.0.

DLKM Module Types
The DLKM feature currently supports the following types of kernel modules:

- WSIO class drivers
- WSIO interface drivers
Developing Dynamically Loadable Kernel Modules

Introduction

- STREAMS drivers
- STREAMS modules
- miscellaneous modules—for example, modules containing support functions not required in the statically configured kernel but shared among multiple loadable modules

Advantages

DLKM modules provide many advantages relative to static modules, including:

- reducing time spent on device driver development by streamlining the driver installation process
- making it easier for administrators to install device drivers from other vendors
- improving system availability by allowing device drivers and other modules to be configured into the kernel while the system is running
- conserving system resources by unloading infrequently used modules when not in use
- providing administrators with the ability to demand load and unload modules
- providing the kernel with the ability to automatically load modules

Auto loading occurs when the kernel detects a particular loadable module is required to accomplish some task, but the module is not currently loaded. The kernel automatically loads the module.

NOTE

Auto unloading is not supported in HP-UX 11.0.
Terms and Definitions

**Auto load**  A capability made possible via the DLKM feature. Auto loading occurs when the kernel detects a particular loadable module is required to accomplish some task, but the module is not currently loaded. The kernel automatically loads the module. During an auto load, the kernel also loads any modules that the module being loaded depends upon, just as it does during a demand load.

**CDIO**  Context-Dependent Input/Output. A feature of the HP-UX I/O subsystem that provides a consistent interface for I/O busses and device drivers.

**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.

**DMA**  Direct Memory Access. High-speed transfer of large quantities of data between the computer memory and a peripheral device without involving the computer central processing unit. The central processing unit is halted during the data transfer and resumes operation when all of the information has been transmitted.

**Kernel module**  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.

**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 in order to make it dynamic.

**PCI**  Peripheral Component Interconnect. An
Developing Dynamically Loadable Kernel Modules

Terms and Definitions

industry-standard bus used on HP-UX systems to provide expansion I/O.

Stream

A connection supported by the STREAMS facilities between a user process and a device driver. It is a structure made up of linked modules, each of which processes the transmitted information and passes it to the next module. You can 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.

WSIO

WSIO Workstation Input/Output. A well-defined environment provided for driver implementation on HP-UX workstations and servers.
Writing DLKM Modules

This section explains the process of writing modules in the DLKM format and provides background information specific to device driver development. Since kernel modules written in the DLKM format can either be configured as dynamically loadable or statically linked, device driver developers must write their DLKM modules to accommodate either configuration.

Writing modules in the DLKM format requires writing additional module initialization code called wrapper code, which enables the DLKM mechanism to logically connect and disconnect a loadable module to and from the running kernel. Existing static device drivers that are going to be configured into the system as loadable modules must be converted to the DLKM format, and must be re-packaged according to the module packaging architecture introduced in HP-UX 11.0.

For HP-UX 11.0, the system must be in a run-time state before dynamic module loading is available. Thus, kernel modules required during system boot must be configured as statically linked.

Module Component Files

A DLKM module is distributed as a set of files, which includes the module’s object file and other configuration files needed to install the module into the system. The files are located in a single directory having an arbitrary name; usually the same name as the name of the module (module_name in this chapter). (See also “Creating the Module’s Component Files” on page 394.

The component files for a DLKM module are:

• mod.o—the object file for the module
• master—the master configuration file for the module
• system—the system configuration file for the module
• space.h (optional)—a configuration file that allocates and initializes some module variables (tunable parameters)
• Modstub.o (optional)—an object file supplied by the DLK stubs mechanism that permits dynamic loading of the module when needed
Examples of component files for a DLKM module are given in “Sample DLKM WSIO Class Driver” on page 409.

**mod.o File Definition**

The C program source code for a DLKM module, which is compiled to become `mod.o`, is similar to the source code for a static module, except that it contains the following additional information:

- The source code for a DLKM module includes two additional header files, `<mod_conf.h>` and `<sys/moddefs.h>`.
- The source code for a DLKM module contains two additional sections of code, one for the module's wrapper and one for the module's `_load()` and `_unload()` functions (routines).

**Module Wrapper**

The source code for a DLKM module must contain a wrapper, defined by the `modwrapper` structure, specifying the `_load()` and `_unload()` function entry points and other data structures used by the module. The name of the `modwrapper` structure for a DLKM module must be `module_name_wrapper`.

The wrapper data structures are initialized by the DLKM mechanism, using values taken from the module's configuration files. These structures provide information needed during loading and unloading, such as the values needed to populate a loadable driver's device switch table entries for the major device number it supports.

The code definition of the `modwrapper` structure is defined in header file `moddefs.h` and shown below. The data structures and `#define` statement preceding the `modwrapper` structure are also defined in `moddefs.h`. The `modwrapper` structure contains a pointer to the `modlink` structure, and the `modlink` structure contains pointers to the `mod_operations` and `mod_type` data structures. The `#define` statement indicates the revision (version) number of the `modwrapper` structure, which is 1.0 for HP-UX 11.0.

**NOTE**

The `modwrapper` revision numbers will be used to maintain backward/forward compatibility of DLKM modules within the DLKM infrastructure.
Example 10-1  An Extract from moddefs.h File Showing modwrapper Structure

```c
#define MODREV 10

struct modlink {
    struct mod_operations *ml_ops;
    void *mw_type_data;
};

/*
 * Module type specific linkage structure.
 */

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

struct modwrapper {
    int mw_rev;
    int (*mw_load)();
    int (*mw_unload)();
    void  (*mw_halt)();
    void  *mw_conf_data;
    struct modlink *mw_modlink;
};
```

The elements of the `modwrapper` structure are:

- `mw_rev`— module revision number; use `MODREV` in your module wrapper
- `mw_load`— pointer to module's `load()` function
- `mw_unload`— pointer to module's `unload()` function
- `mw_halt`— currently unused; use `(void (*)())NULL` in your module wrapper
- `mw_conf_data`— pointer to configuration data created by configuration tool `config(1M)`
- `mw_modlink`— pointer to a `modlink` array, which is a NULL terminated array of `modlink` structures that specify the type-specific operations required by the module

The elements of the `modlink` structure are:

- `ml_ops`— pointer to a `mod_operations` structure that depends on the
Developing Dynamically Loadable Kernel Modules

Writing DLKM Modules

The type of DLKM module being defined:

Table 10-1

<table>
<thead>
<tr>
<th>Module Operations</th>
<th>DLKM Module Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>gio_mod_ops</td>
<td>WSIO class driver</td>
</tr>
<tr>
<td>gio_mod_ops</td>
<td>WSIO interface driver</td>
</tr>
<tr>
<td>str_drv_ops</td>
<td>STREAMS driver</td>
</tr>
<tr>
<td>str_mod_ops</td>
<td>STREAMS module</td>
</tr>
<tr>
<td>mod_misc_ops</td>
<td>Miscellaneous module</td>
</tr>
</tbody>
</table>

- `ml_type_data`— pointer to a `mod_type_data` structure

The elements of the `mod_type_data` structure are:

- `mtd_info`— information string returned to `modstat()` system call
- `mtd_pdata`— pointer to type-specific data that depend on the type of loadable module being defined: for both STREAMS drivers and STREAMS modules, use a pointer to the `streams_info_t` structure; for all other types, use `(void *)NULL`

A sample wrapper for a WSIO driver is as follows:

```
/*
 * Wrapper Table
 */
extern struct mod_operations gio_mod_ops;
static drv_info_t module_name_drv_info;
extern struct mod_conf_data module_name_conf_data;

/* module type specific data */
static struct mod_type_data module_name_drv_link = {
    "module_name - Loadable/Unloadable Test Module",
    (void *)NULL
};

static struct modlink module_name_mod_link[] = {
    { &gio_mod_ops, (void *)&module_name_drv_link },
    /* WSIO */
    { NULL, (void *)NULL }
```

Table 10-1 Module Operations
struct modwrapper module_name_wrapper = {
    MODREV,
    module_name_load,
    module_nameUnload,
    (void (*)(void)) NULL,
    (void *)&module_name_conf_data,
    module_name_mod_link
};

When a DLKM module is configured as statically linked, its wrapper is not used by the system.

Load Function

int module_name_load (void *arg);

A module's _load() function is called by the DLKM mechanism whenever the module is loaded from disk into active memory. The function may be given any name (typically module_name_load); a pointer to the _load() function is obtained from the module's wrapper.

The _load() function must perform any initialization tasks required by the module before the module is logically connected to the kernel. Typical initialization tasks include acquiring private memory for the module and initializing devices and data structures.

If the module is unable to initialize itself, the _load() function must free any memory that it allocated and undo any other action that it took prior to the failure, including canceling all outstanding calls to timeout. The _load() function should return 0 on success and an errno value on failure.

The argument passed to the function is type-specific. The use of this argument is described in “Initializing and Terminating DLKM Modules” on page 376.

Unload Function

int module_name_unload (void *arg);

The _unload() function is called by the DLKM mechanism whenever the module is about to be removed from active memory. The function may be given any name (typically module_name_unload); a pointer to the _unload() function is obtained from the module's wrapper.
The _unload() function must clean up any resources that were allocated to the module, and it must remove all references to the module. Typical cleanup tasks include releasing private memory acquired by the module, removing device interrupts, disabling interrupts from the device, and canceling any outstanding timeout requests made by the module.

The module's _unload() function should return 0 on success and an errno value on failure. In the event of failure, the function must leave the module in a sane state, since the module will remain loaded after the return.

The system will never attempt to unload a module that it thinks is busy. However, the system cannot determine under all cases when the module is in use. Currently, a module is considered to be busy when another module that depends on it is also loaded. In addition, WSIO class drivers and STREAMS drivers track the open() and close() calls; these types of modules are busy whenever there is at least one open on the device using the driver. Under most other circumstances, the module must determine for itself whether it is appropriate for it to be unloaded. When a module is still in use, its _unload() function can return a non-zero value to cancel the unload.

The argument passed to the _unload() function is the same type-specific value that was passed to the module's _load() function. The use of this argument is described in “Initializing and Terminating DLKM Modules” on page 376.

**master File Definition**

Each DLKM module has its own master file. The format of the master file includes the following section keywords:

- $VERSION—indicates the version number for the file format. Version is defined as an integer and starts from one. For HP-UX 11.0, enter a single line containing the integer 1.
- $LOADABLE—indicates that the module is dynamically loadable. If this section keyword does not exist, the module can only be statically linked into the kernel.
- $INTERFACE—identifies the interface names and versions on which the module is built. For HP-UX 11.0, enter a single line containing the word base.
Developing Dynamically Loadable Kernel Modules

Writing DLKM Modules

- $TYPE$—indicates the module type and the type specific information. Valid types are $wsio\_class$, $wsio\_intfc$, $streams\_mod$, $streams\_drv$, and $misc$.

- Other sections (if required)—$DRIVER\_DEPENDENCY$, $TUNABLE$, and $DRIVER\_INSTALL$, as described in the "Kernel Module Master File" section of the master (4) manpage.

  In the $DRIVER\_DEPENDENCY$ section, enter the names of all other modules that depend upon this module.

  In the $TUNABLE$ section, enter the names and default values of the tunable parameters (variables) for the module. You may also enter minimum values for tunable parameters.

  In the $DRIVER\_INSTALL$ section, enter the module's name and associated block and/or character major device number(s). To have the system dynamically assign a major number to your module, specify -1 in both the block major and char major fields of $DRIVER\_INSTALL$.

**system File Definition**

Every DLKM module requires a system file. The system file includes the following three mandatory and one optional section keywords:

- $VERSION$—indicates the version number for the file format. Version is defined as an integer and starts from one. For HP-UX 11.0, enter a single line containing the integer 1

  **NOTE**
  
  The version number for the master file and system file must be the same.

- $CONFIGURE$—indicates if the module needs to be configured into the system. If $CONFIGURE$ is Y or y, the module will be configured on the next build; if $CONFIGURE$ is N or n, the module will not be configured on the next build. kmsystem (1M) provides the interface to modify the flag.

- $LOADABLE$—indicates how the module will be configured. If $LOADABLE$ is Y or y, the module will be configured as a dynamically loadable module; if $LOADABLE$ is N or n, the module will be statically linked into the kernel, requiring a reboot. kmsystem provides the
Developing Dynamically Loadable Kernel Modules
Writing DLKM Modules

interface to modify the flag.

NOTE

If $CONFIGURE is N or n, $LOADABLE is ignored.

NOTE

If the master file for the module does not have a $LOADABLE section, then the system file should not have one either.

• $TUNABLE (empty)—place holder for any tunable parameter specified in the associated master file for which you want to specify a value other than the default value. Enter nothing here.

kmtune(1M) is the interface to modify tunable parameters in the module's system description file (a copy of the module's system file that will eventually reside in /stand/system.d/module_name) and the HP-UX system file, /stand/system, by default.

space.h File Definition

An optional component, the space.h file, contains storage allocations and initializations of data structures associated with a DLKM module when the size or initial value of the data structures depend on configurable values such as tunable parameters. In order to communicate these values to the rest of the DLKM module, the values are stored in global variables and accessed by the module via extern declarations in the module's mod.o file.

NOTE

All tunable parameters specified in the master file are defined as global variables in the space.h file. See “Sample DLKM WSIO Class Driver” on page 409, for clarification.

Modstub.o File Definition

An optional component, the Modstub.o file is statically linked into the kernel as a place holder for functions implemented in a loadable module that will be loaded at a later time. Its purpose is to enable the kernel to
resolve references to the absent module’s functions.

_modstub.o contains stubs for entry points defined in the associated loadable module that can be referenced by other statically linked kernel modules currently configured in the system. Access to a stub causes the kernel to auto load the associated loadable module.
Developing Dynamically Loadable Kernel Modules

Initializing and Terminating DLKM Modules

In a traditional system with statically linked modules, the modules are initialized during system boot. Because dynamically loadable modules are loaded after booting the system, they must be initialized differently than modules that are statically linked into the kernel during system boot. This section describes the steps that each type of dynamically loadable module must take to initialize itself and to terminate itself.

In addition to WSIO class drivers, WSIO interface drivers, STREAMS drivers, STREAMS modules, and miscellaneous modules, this section includes the initialization and termination tasks for WSIO monolithic drivers. A monolithic driver is both a class driver (one that has a device special file) and an interface driver (one that touches real hardware and registers).

WSIO Class Drivers

To make a WSIO class driver loadable, the driver must provide a _load() and _unload() function and make minor changes to the way that the driver initializes itself. When a driver is statically linked, it typically links its _init() function into the dev_init chain, and the _init() function calls the next driver's _init() function in the chain. This mechanism is inappropriate for loadable drivers since the dev_init chain is only used during system boot, and there is no mechanism to remove the entry from the chain when the module is unloaded. Therefore, the module's _load() function should perform both the installation tasks (those normally done during install for static modules) and the driver initialization tasks, and the dev_init chain should be ignored.

Additionally, the _load() function is passed a pointer to an updated version of the drv_info_t structure; the driver must use this version of the drv_info structure when it registers itself with WSIO. A sample _load() function for a WSIO class driver is as follows:

**Example 10-3**

```c
static wsio_drv_info_t module_name_wsio_info = { ... };

/*
 * LOAD
```
Developing Dynamically Loadable Kernel Modules

Initializing and Terminating DLKM Modules

/*

static int
module_name_load(void *arg)
{
    if (module_name_debug)
        printf("module_name> Loading\n");

    /* Use drv_info passed to us instead of static version */
    module_name_wsio_info.drv_info = (drv_info_t *) arg;

    /* Register with WSIO */
    if (!wsio_install_driver(&module_name_wsio_info))
    {
        printf("module_name> wsio_install_driver failed!!\n");
        return (ENXIO);
    }

    /* Perform driver-specific initialization, but do not
     * call next function in the dev_init list.
     */
    (void) module_name_init();

    return (0);
}

Initialization for a statically linked WSIO class driver is unchanged from
historical practice. That is, the initial driver entry point is
module_name_install. This function typically installs an _init() function in a list of functions that will be invoked later in the system boot
process, and then the driver registers itself with WSIO. For example:

Example 10-4  WSIO Class Driver _install Function—Example

static void module_name_linked_init (void);
static int (*module_name_saved_init)();

/*
 * INSTALL
 * This function is called if module is statically linked.
 */
int
module_name_install(void)
{
    extern int (*dev_init)(void);
Developing Dynamically Loadable Kernel Modules
Initializing and Terminating DLKM Modules

/* Link my init function into chain */
module_name_saved_init = dev_init;
dev_init = (int (*)(())) & module_name_linked_init;

/**< Register driver with WSIO */
return ( wsio_install_driver(& module_name_wsio_info) );
}

The _init() function in the static case must call the next driver’s _init() function:

Example 10-5  WSIO Class Driver _init Function—Example

/*
 * Device initialization Link
 * Called only for statically linked drivers to link
 * init routine into list.
 */
static void
module_name_linked_init (void)
{
    /* Perform driver-specific initialization */
    (void) module_name_init ();

    /* Call next init function in chain */
    (void) (*module_name_saved_init());
}

The _unload() function frees resources allocated by the driver and unregisters the driver from WSIO. If it is safe to unload the driver when the number of open devices for the driver goes to zero, then the module need not perform any special checks to determine if it is busy. If any step in the _unload() process fails, the driver must undo any action prior to the failure. A sample _unload() function for a WSIO class driver follows:
Example 10-6  
WSIO Class Driver _unload Function—Example

```c
/*
* UNLOAD
*/
static int
module_name_unload(void *drv_infop)
{
    /* This function is only called when the administrator
       attempts to unload the module and there are no open
       devices using the module. If there is some reason that
       the module should not be unloaded, check it now and
       return non-zero.
    */
    if (module_name_no_unload) {
        printf("module_name> I’m BUSY\n");
        return (EBUSY);
    }

    /* Unregister with WSIO */
    if (wsio_uninstall_driver(&module_name_wsio_info) ) {
        /* Uninstall failed! Return to a loaded, functional
           state. */
        printf("module_name> wsio_uninstall_driver failed!!\n");
        return (ENXIO);
    }

    /* Cancel pending timeouts, free allocated memory and
       resources, etc.
    */
    if (module_name_debug)
        printf("module_name> Unloaded\n");

    return (0);
}
```

WSIO Interface Drivers

A WSIO interface driver requires an _attach() function for any
interface card it intends to claim. It will also require a _probe() 
function if it intends to install device probes.

When installing a WSIO interface driver, the driver must register itself
with WSIO and set up the _attach() function and any device probes.
Because the timing of initialization differs between loadable and statically linked modules, some of the steps that are handled automatically during boot for static drivers must be explicitly performed by the _load() function.

The primary difference in making a WSIO interface driver loadable concerns the handling of the attach list. Historically, drivers have added their _attach() functions to a global list, and the _attach() function was responsible for calling the next driver’s _attach() function in the list. However, this method does not allow the driver to remove its _attach() function from the list; thus, this approach cannot support unloading. New support functions have been added to WSIO to allow interface drivers to add and remove their _attach() functions from the attach list. These functions are:

```c
int mod_wsio_attach_list_add(list_type, &attach_func);
int mod_wsio_attach_list_remove(list_type, &attach_func);
```

`list_type` specifies the attach list to use; valid entries are MOD_WSIO_CORE, MOD_WSIO_PCI, and MOD_WSIO_EISA. Both functions return 0 on success and 1 on failure.

These functions should only be called when the module is dynamically loaded. Statically linked modules should continue to use the existing attach chain.

Device probes are normally associated with interface drivers during the initialization of the WSIO Context-Dependent Input/Output (CDIO) at boot time. Since this is only done once, loadable interface drivers must explicitly connect the device probe to the driver’s drv_info structure:

```c
void wsio_activate_probe(char *probe_name, struct drv_info *drv_infop);
```

`probe_name` is the name of the probe as registered by `wsio_register_dev_probe()` or `wsio_register_probe_func()`. `drv_infop` is a pointer to the drv_info structure for this driver.

Finally, similar to class drivers, the _load() function for interface drivers is passed a pointer to an updated version of the drv_info_t structure; the interface driver must use this version of the drv_info structure when it registers itself with WSIO.

The sample code below puts all these concepts together to demonstrate the loading and initialization steps for an interface driver. This sample can be configured as either a loadable module or a statically linked
module.

Example 10-7 WSIO Interface Driver Loading and Initialization Coding—Example

```c
static wsio_drv_info_t module_name_wsio_info = { ... };  
static int (*module_name_saved_attach)();

int module_name_load(void *arg)
{
    /* Use the drv_info passed to us instead of the */
    /* static version */
    wsio_info.drv_info = (drv_info_t *) arg;

    /* Register the driver with WSIO */
    if (!wsio_install_driver(&wsio_info))
        return ENXIO;   /* install failed! */

    /* Add the attach function to the DLKM attach list. */
    mod_wsio_attach_list_add(MOD_WSIO_CORE,
        &module_name_core_attach);

    /* Register the device probe. */
    wsio_register_dev_probe(IF_CLASS, probe_func,
        "probe_name");

    /* The following step is only required for dynamically */
    /* loaded modules: attach the probe function to the */
    /* drv_info structure. */
    wsio_activate_probe("probe_name", wsio_info.drv_info);

    return 0;
}

int module_name_install(void)
{
    /* Add the attach function to the list. */
    module_name_saved_attach=
        core_attach;
    core_attach=
        module_name_attach_linked;

    /* Register the device probe. */
    wsio_register_dev_probe(IF_CLASS, probe_func,
        "probe_name");
```

Chapter 10
Initializing and Terminating DLKM Modules

Developing Dynamically Loadable Kernel Modules

/* Register the driver with WSIO */
return (wsio_install_driver(&wsio_info);
}

/* Common attach function for both dynamic and static
 * modules */
int
module_name_attach(int id, struct isc_table_type *isc)
{
    /* Normal attach function operations */
    ...
}

/* Attach function called from attach chain for static
 * modules
 * only
 */
int
module_name_attach_linked (int id, struct isc_table_type *isc)
{
    module_name_attach(id, isc);
    return ((*module_name_saved_attach)(id, isc));
}

The _unload() function determines if the driver is still busy, and if not, it cleans up all resources that were obtained by the driver. It should free memory that it allocated and remove the attach() function from the attach list. It should also unregister any _probe() function pointers that the driver has registered with other kernel services; for example, the _unload() function should unlink the interrupt service function. If an operation fails while unloading, the _unload() function must be able to restore the driver to a working state and return a non-zero value.

Unlike class drivers, there is no automatic method for the system to determine if an interface driver is still busy. This problem can sometimes be avoided by making class drivers that use the interface driver dependent upon the interface driver. With this dependency relationship in place, the system will not allow the interface driver to be unloaded as long as any class driver that depends upon it is still loaded. If it is not appropriate for the interface driver to rely on dependencies, then it must determine via other means if it is possible to unload the driver. The _unload() function is always free to return a non-zero value, and the module will remain loaded. A sample _unload() function for a WSIO interface driver is as follows:
Developing Dynamically Loadable Kernel Modules

Initializing and Terminating DLKM Modules

Example 10-8  

**WSIO Interface Driver _unload Function—Example**

```c
int module_name_unload(void *arg)
{
    int ret;
    struct isc_table_type *isc;
    void *token, *priv_ptr;

    /* Remove the attach function from the DLKM attach list. */
    if (mod_wsio_attach_list_remove(MOD_WSIO_CORE, &module_name_core_attach)) {
        return(ENXIO);
    }

    /* Unregister the device probe. */
    wsio_unregister_dev_probe(IF_CLASS, "probe_name");

    if (wsio_uninstall_driver(&module_name_wsio_info)) {
        /* uninstall failed - go back to loaded state
         * undo what has been done in _unload routine */
        return ENXIO;
    }
    return(0);
}
```

**WSIO Monolithic Drivers**

The _load() function for a monolithic driver must effectively be the union of the _load() functions for the class and interface drivers described previously. Similarly, the _unload() function must effectively be the union of the _unload() functions for the class and interface drivers.

**STREAMS Drivers**

Initialization of STREAMS drivers is very similar for both the loadable and statically linked module cases. The only difference is that loadable drivers must use the drv_info_t structure that is passed as an argument to the _load() function. The major numbers from this structure must also be used in the streams_info_t structure passed to str_install(). Sample _load() and _install() functions for a STREAMS driver are as follows:
Example 10-9  STREAMS Driver _load and _install Functions—Example

static drv_info_t str_drv_info = { ... };  
static drv_info_t *drv_info_p = &str_drv_info;

static streams_info_t str_info = { ... };  
static drv_ops_t module_name_str_drv_ops = { ... };

int module_name_load(void *arg)
{
    int    retval;
    
    /* Use the drv_info passed to us instead of the static * version */
    drv_info_p = (drv_info_t *) arg;
    str_info.inst_major = drv_info_p->c_major;

    if (module_name_install())
        return ENXIO;
    return 0;
}

int module_name_install(void)
{
    int    retval;
    
    /* Install in cdevsw */
    if ((retval = install_driver(drv_info_p,
                                 &module_name_str_drv_ops)) != 0)
        return (retval);    /* install failed */

    /* Install in Streams */
    if ((retval = str_install(&str_info)) != 0)
    {
        uninstall_driver(drv_info_p);
        return retval;
    }
    return 0;
}

STREAMS drivers, like WSIO class drivers, automatically track open() and close() system calls for the STREAMS device. The system will prevent a STREAMS driver from unloading whenever the device has one
or more open file handles. Of course, the driver can still disallow an unload if this check is insufficient for its needs. A typical \_unload() function for a STREAMS driver is as follows:

**Example 10-10** STREAMS Driver \_unload Function—Example

\begin{verbatim}
module_name\_unload(void *arg)
{
    int    retval;

    /* Uninstall from Streams */
    if ((retval = str_uninstall(&str_info)) != 0)
        return retval;

    /* Free module specific resources, etc. */
    ...

    return 0;
}
\end{verbatim}

**STREAMS Modules**

Loadable STREAMS modules have no special requirements during initialization. The argument passed to the \_load() function should be ignored. Sample \_load() and \_install() functions are as follows:

**Example 10-11** STREAMS Module \_load and \_install Functions—Example

\begin{verbatim}
static streams_info_t str_info = { ... };

int module_name\_load(void *arg)
{
    int    retval;

    if (module_name\_install())
        return ENXIO;
    return 0;
}

int module_name\_install(void)
{
    /* Install in Streams */
    if ((retval = str\_install(&str\_info)) != 0)
        return retval;
    return 0;
}
\end{verbatim}
The system automatically tracks pushes and pops of a STREAMS module on active streams; a module cannot be unloaded while it is pushed onto one or more streams. A typical `unload()` function for a STREAMS module is as follows:

**Example 10-12**

**STREAMS Module unload Function—Example**

```c
int module_name_unload(void *arg)
{
    int    retval;

    /* Uninstall from Streams */
    if ((retval = str_uninstall(&str_info)) != 0)
        return retval;

    /* Free module specific resources, etc. */
    ...
    return 0;
}
```

**Miscellaneous Modules**

Miscellaneous modules can implement any feature within the kernel. As such, a miscellaneous module's `load()` function must address all of the module's specific needs. Similarly, the module's `unload()` function must determine for itself if it is safe to unload. The system will not allow a module to be unloaded if other loaded modules are dependent upon the module. Other than this check, the system performs no other checks when the administrator attempts to remove a miscellaneous module from the kernel.

The argument to the `load()` function is not meaningful and should be ignored.
Kernel Configuration Tool Set

There are a number of tools known collectively as the kernel configuration tool set for installing, configuring, and managing DLKM modules. These tools are identified and briefly described in Table 10-2. The action carried out by a kernel configuration tool depends upon the options you specify during the tool's invocation. For a complete description of the kernel configuration tools and their options, refer to the appropriate manpages.

You, as the driver writer or system administrator, use the kernel configuration tools to install, configure, load, unload, update, or remove kernel modules from the system; and build new kernels. You must have super-user privileges—logged in as user root—to perform these tasks. The tasks you perform using the kernel configuration tools are described in the next section, "Development Process" on page 390.

NOTE

To use the kernel configuration tool set to install and configure static modules, you need to re-package them according to the module packaging architecture described in “Writing DLKM Modules” on page 367. Do not include a $LOADABLE section in the master or system file.

Table 10-2 Kernel Configuration Tool Set

<table>
<thead>
<tr>
<th>Tool/Command</th>
<th>Action</th>
</tr>
</thead>
</table>
| config (1M)  | • First form—generates both the static kernel and dynamically loadable modules; a system reboot is necessary.  
• Second form, -M option—generates the specified loadable module. The newly configured service is available immediately, without requiring a system reboot. |
## Kernel Configuration Tool Set

### kmadmin (1M)
- `-k` option—prints the status of all statically linked modules.
- `-L` option—loads the specified loadable module into the running kernel.
- `-Q`, `-q` option—prints the status of the specified loadable module.
- `-S`, `-s` option—prints the status of all currently loaded or registered loadable modules.
- `-U`, `-u` option—unloads the specified loadable module from the running kernel.

### kminstall (1M)
- `-a` option—adds a module's component files to certain subdirectories of `/usr/conf` and `/stand`.
- `-d` option—deletes a module's component files from the subdirectories of `/usr/conf` and `/stand` and then deletes the module's component files.
- `-u` option—copies a module's updated component files into the subdirectories of `/usr/conf` and `/stand`.

<table>
<thead>
<tr>
<th>Tool/Command</th>
<th>Action</th>
</tr>
</thead>
</table>
| kmadmin (1M) | `-k` option—prints the status of all statically linked modules.  
- `-L` option—loads the specified loadable module into the running kernel.  
- `-Q`, `-q` option—prints the status of the specified loadable module.  
- `-S`, `-s` option—prints the status of all currently loaded or registered loadable modules.  
- `-U`, `-u` option—unloads the specified loadable module from the running kernel. |
| kminstall (1M) | `-a` option—adds a module's component files to certain subdirectories of `/usr/conf` and `/stand`.  
- `-d` option—deletes a module's component files from the subdirectories of `/usr/conf` and `/stand` and then deletes the module's component files.  
- `-u` option—copies a module's updated component files into the subdirectories of `/usr/conf` and `/stand`. |
Table 10-2

<table>
<thead>
<tr>
<th>Tool/Command</th>
<th>Action</th>
</tr>
</thead>
</table>
| kmsystem (1M) | • \(-c\) option—assigns a value (Y or N) to the configuration ($CONFIGURATION) flag of the specified module.  
• \(-l\) option—assigns a value (Y or N) to the loadable ($LOADABLE) flag of the specified module.  
• \(-q\) option—prints the values of the configuration and loadable flags of the specified module. Prints a – (signifies “does not apply”) for the loadable flag of a static module.  
• no options or \(-S\) option only—prints the values of the configuration and loadable flags of all modules. Prints a – for the loadable flags of static modules. |
| kmtune (1M) | • \(-l\) option—prints the values of all system parameters.  
• \(-q\) option—queries the value of the specified system parameter.  
• \(-r\) option—resets the value of the specified parameter to its default value.  
• \(-s\) option—assigns a value to the specified system parameter. |
| kmupdate (1M) | • First form—prepares the system to move the specified static kernel and its associated files to the /stand/vmunix file and /stand/dlkm directory, respectively, during the next system shutdown and startup.  
• Second form, \(\/-M\) option—moves the image of the specified loadable module to /stand/dlkm/mod.d/module_name and registers the module with the kernel either (1) immediately or (2) later at system shutdown. |
Development Process

Developing a DLKM module such as a device driver requires more up-front planning than most application programming projects. At the very least, testing and debugging are more involved, and more knowledge about hardware is required.

This section describes the procedures for developing a DLKM module. Since you can include a DLKM module in the system in one of two ways, either as a dynamically loadable module or as a statically linked module, this section considers both methods of inclusion.

The sequence of steps for developing a DLKM module is shown in Figure 10-1. Detailed descriptions of the procedural steps appear in this section in the same general order as they appear in the figure. The incomplete command line invocations in the figure are only intended to identify the command and basic options needed to implement the specific tasks. Complete command line invocations are included in the detailed descriptions.

As indicated in the figure, developing a DLKM module is an iterative process. You write, test, and debug a module in a piecemeal fashion, building up to the implementation of the complete module.

NOTE

During the first phases of development, possible errors in the module code may panic or damage the system, even parts of the system that may seem unrelated to your module. Testing should be done when no other users are on the system and all production data files are backed up. Ideally, testing would be performed on a restricted-use system set up specifically for the purpose of developing kernel modules.

Before starting the development process, it is always a good idea to make your own backup copy of the currently running kernel (/stand/vmunix) for emergency recovery purposes.
Developing Dynamically Loadable Kernel Modules

Development Process

Figure 10-1: Development Process for DLKM Modules (Sheet 1 of 3)

1. Create module’s component files. Compile module source code using `cc` command.
2. Install module’s component files using `kminstall -a` command.
3. Prepare module as a dynamically loadable module or statically linked module using `kmsystem -c -l` command.
4. Optional: Tune system parameter(s) supplied by module or static kernel using `kmtune -s` command.
5. Loadable or static?
6. Loadable
   - Configure loadable module into system using `config -M` command.
   - Move loadable module’s image into place and register module using `kmupdate -M` command.
7. Static
   - Configure statically linked module into system by building new kernel using `config /stand/system` command.
   - Prepare system to move new kernel into place during next system shutdown and startup using `kmupdate /stand/build/vmunix_test` command.

From Sheets 2, 3

To Sheet 2

To Sheet 3
From Sheet 1

1. Load loadable module using `kmadmin -L` command.

   Optional: Query loadable module using `kmadmin -Q` command.

   If necessary, create device special file(s) for loadable module using `mknod` command.

   Test loadable module using standard testing techniques.

   All tests passed?

   Yes

   No

   Unload loadable module using `kmadmin -U` command.

   Correct errors in module's component files. Recompile module source code using `cc` command.

   Update module's components using `kminstall -u` command.

   To Sheet 1

2. Remove module's components from system using `kminstall -d` command.

   Done
Figure 10-3  Development Process for DLKM Modules (Sheet 3 of 3)

From Sheet 1

2

Load statically linked module by activating new kernel using `shutdown -r` command.

Optional: Query statically linked module using `kmadmin -k` command.

If necessary, create device special file(s) for statically linked module using `mknod` command.

Test statically linked module using standard testing techniques.

All tests passed?

Yes

Remove module's components from system using `kminstall -d` command.

Update module's components using `kminstall -u` command.

Build and load new kernel using `config -u` and `shutdown -r` commands.

Done

No

Correct errors in module's component files. Recompile module source code using `cc` command.

Update module's components using `kminstall -u` command.

To Sheet 1

3
Creating the Module's Component Files

Create the mod.o, master, system, space.h (optional), and Modstub.o (optional) files for the DLKM module in a single directory. It is suggested that you use a subdirectory of /usr/conf such as /usr/conf/module_name.

When choosing a name for your module, choose a unique name to avoid conflict with kernel functions and global variables. Consider using your company's name and something that indicates the module's purpose.

To compile your module source code, use the ANSI C compiler /usr/ccs/bin/cc and the compiler options shown below. You can also use the compiler options found in the makefile /stand/build/config.mk or /stand/build/config.mod.

To compile your module source code, follow these steps:

1. Change directories to the directory containing the module's component files.
2. For a 32-bit target machine, execute the following cc (1) command:

   ```
   /usr/ccs/bin/cc -Ae -I. -I/usr/conf/h +ES1.Xindirect_calls -c \n   -o mod.o -U_hp9000s700 -D_STDC_EXT__ -D_XPG4_EXTENDED \n   -D_HPUX_SOURCE -D_hp9000s800 -D_KERNEL -DKERNEL \n   -Wp,-H300000 +XixdU +Hx0 +R500 -Wl,-a,archive +DA1.1 \n   +DS2.0 +ESsfc \n   module_name.c
   ```

3. For a 64-bit target machine, execute the following cc (1) command:

   ```
   /usr/ccs/bin/cc -Ae -I. -I/usr/conf/h +ES1.Xindirect_calls -c \n   -o mod.o -U_hp9000s700 -D_STDC_EXT__ -D_XPG4_EXTENDED \n   -D_HPUX_SOURCE -D_hp9000s800 -D_KERNEL -DKERNEL \n   -Wp,-H300000 +XixdU +Hx0 +R500 -Wl,-a,archive +DA2.0W \n   +DS2.0 +ESsfc \n   module_name.c
   ```

Installing the Module's Component Files

Once the module's mod.o, master, system, and optional files have been created, call the kminstall command to copy those files to certain subdirectories of /usr/conf and /stand. kminstall creates the required subdirectories if they do not exist.

kminstall expects to find the module's component files in the current working directory. If module_name already exists on the system, kminstall prints a message and fails.
To install a DLKM module's components, follow these steps:

1. Change directories to the directory containing the module's component files.
2. Execute the following `kminstall` command:
   
   ```
   /usr/sbin/kminstall -a module_name
   ```

   `kminstall` copies the module's component files to the appropriate locations. For example, `kminstall` copies the master file to `/usr/conf/master.d/module_name`, and the system file to `/stand/system.d/module_name`. File `/usr/conf/master.d/module_name` is known as the module's master configuration file. File `/stand/system.d/module_name` is known as the module's system description file.

   **NOTE**
   The file locations receiving copies of the module's component files may change in future HP-UX releases.

---

**Preparing the Module for Configuration**

Use the `kmsystem` command to prepare a DLKM module for configuration as either (1) dynamically loadable or (2) statically linked. For HP-UX 11.0, you must prepare all DLKM modules that are required to boot the kernel as statically linked modules.

Use `kmsystem` command to assign values (Y or N) to the configuration ($CONFIGURATION) and loadable ($LOADABLE) flags in the module's system description file. If the loadable flag is not present in the system description file, the module is a static module, and `kmsystem` exits with an error.

**NOTE**
System description files as well as the HP-UX system file must only be modified using the `kmsystem` or `kmtune` command.

---

**Preparing a Dynamically Loadable Module**

To prepare a DLKM module for configuration as dynamically loadable, execute the following `kmsystem` command:
Developing Dynamically Loadable Kernel Modules

Development Process

```
/usr/sbin/kmsystem -c Y -l Y module_name
```

Preparation of a Statically Linked Module

To prepare a DLKM module for configuration as statically linked, execute the following `kmsystem` command:

```
/usr/sbin/kmsystem -c Y -l N module_name
```

Tuning the Module

Use the `kmtune` command to query, set, or reset system (tunable) parameters used by the DLKM module or the static kernel. `kmtune` reads the master configuration files, the system description files, and the HP-UX system file.

For a DLKM module or a static module using 11.0 module packaging, `kmtune` writes any user-modified system parameter to the module’s system description file. For a static module using pre-11.0 module packaging, `kmtune` writes any user-modified system parameter to the HP-UX system file.

To query the value of a specific system parameter, execute the following `kmtune` command:

```
/usr/sbin/kmtune -q system_parameter_name
```

To set the value of a specific system parameter, execute the following `kmtune` command:

```
/usr/sbin/kmtune -s system_parameter_name=value
```

To reset the value of a system parameter to its default value, execute the following `kmtune` command:

```
/usr/sbin/kmtune -r system_parameter_name
```

If you modify a system parameter for a loadable module, you will have to execute `config -M -u` to reconfigure that module for the change to take effect, assuming that the module is not currently loaded (see note below). If the module is currently loaded, you must unload and then reload the module for the change to take effect. If you modify a system parameter for a statically linked module that is currently linked into the kernel, you will have to execute `config -u` to reconfigure the kernel and then shutdown and restart the system for the change to take effect.
NOTE

As a consequence of the DLKM feature, the system now maintains a dynamic symbol table in kernel address space. The dynamic symbol table contains all global symbols (function names and variable definitions—including system parameters) defined in the static kernel plus all global symbols defined in all currently loaded modules. The contents of the dynamic symbol table change as modules are loaded and unloaded; when a module is loaded, its symbol information is added to the table, and when the module is unloaded, its symbol information is deleted.

Configuring the Module into the System

Use the `config` command to configure a DLKM module into the system as either (1) dynamically loadable or (2) statically linked. Use the `kmsystem` command as described in “Preparing the Module for Configuration” on page 395, to prepare the DLKM module for the desired configuration.

When configuring a DLKM module into the system as dynamically loadable, `config` proceeds as follows:

1. reads the module's master configuration file
2. reads the HP-UX system file
3. reads the module's system description file
4. checks the interface functions or symbols used by the module
5. generates several C output files (including a makefile named `config.mod`) describing the system configuration
6. executes the makefile to configure the dynamically loadable module
7. places the generated dynamically loadable module image file under the kernel function set directory associated with the running kernel: 
   `/stand/dlkm/mod_bld.d/module_name/mod_reg`
8. adds an entry for the generated dynamically loadable module to the `mod_reg` file: `/stand/dlkm/mod_bld.d/module_name/mod_reg`

When configuring a DLKM module into the system as statically linked, `config` builds an entire kernel, that is, configures the static kernel and all kernel modules. `config` proceeds as follows:
Developing Dynamically Loadable Kernel Modules

Development Process

1. reads the master configuration files for the kernel modules
2. reads the HP-UX system file
3. reads the system description files for the kernel modules
4. checks the interface functions or symbols used by the kernel modules
5. generates several C output files (including two makefiles named `config.mk` and `config.mod`) describing the system configuration
6. executes the `config.mk` makefile to configure a new kernel, to link the kernel with the appropriate kernel libraries and statically linked modules, and to generate the kernel symbol table
7. executes the `config.mod` makefile to configure the dynamically loadable modules
8. places the newly generated kernel file (`vmunix_test`) in the `/stand/build` directory
9. places the generated symbol table file (`symtab`) and generated dynamically loadable module image files (`module_names`) under the kernel function set directory associated with the newly generated kernel:
   `/stand/build/dlkm/vmunix_test/symtab`
   `/stand/build/dlkm/vmunix_test/mod.d/module_names`
10. adds entries for the generated dynamically loadable modules to the `mod_register` file:
    `/stand/build/dlkm/vmunix_test/mod_register`

---

**NOTE**

The newly generated kernel is a static kernel held on disk, while the running kernel is in active memory. You must shutdown and restart the system to load the newly generated kernel into active memory to become the new running kernel.

---

**Configuring a Dynamically Loadable Module**

To configure a DLKM module as dynamically loadable, execute the following `config` command:

```
/usr/sbin/config -M module_name -u
```

The `-u` option in the `config` command line causes the system to automatically perform the following actions:
1. move the generated dynamically loadable module image file to 
   /stand/dlkm/mod.d/module_name
2. add an entry for the generated dynamically loadable module to 
   /stand/dlkm/mod_register
3. register the generated dynamically loadable module with the running kernel

Your loadable module is ready to load immediately; meaning that you do not have to wait for a reboot to be able to load it. You can load your module at any time.

Configuring a Statically Linked Module

To configure a DLKM module as statically linked, execute the following config command:

/usr/sbin/config -u /stand/system

cfg config builds a new kernel. The -u option in the config command line causes the system to automatically perform the following actions when you shutdown and restart the system:

1. move the newly generated kernel file and its kernel function set directory to their default locations, /stand/vmunix and /stand/dlkm, respectively
2. load the newly generated kernel into active memory to become the running kernel
3. register all of the generated dynamically loadable modules with the running kernel

NOTE

The system saves the previous kernel file and its kernel function set directory as /stand/vmunix.prev and /stand/dlkm.vmunix.prev, respectively, at shutdown time.

At boot time, your DLKM module will be configured as statically linked into the new running kernel.

Updating the Module’s Image

For a DLKM module configured as dynamically loadable, you use the
kmupdate command to update its image and register it with the kernel. Updating a dynamically loadable module's image means moving its image into place and registering it with the kernel either (1) immediately or (2) later at system shutdown. The process is described in "Configuring a Dynamically Loadable Module" on page 398.

Call kmupdate after first calling config without its -u option. If you include the -u option in the config invocation, there is no need to invoke kmupdate. The config -M -u command automatically invokes kmupdate.

To update the image of a dynamically loadable module immediately, execute the following kmupdate command:

    /usr/sbin/kmupdate -M module_name -i

After updating the specified module and assuming the module was loaded originally, kmupdate will reload the module before exiting.

To update the image of a dynamically loadable module at system shutdown, execute the following kmupdate command:

    /usr/sbin/kmupdate -M module_name -a

If you do not specify the -i or -a option, kmupdate will attempt to update the specified loadable module immediately. If the module cannot be updated immediately, the module will be updated at system shutdown.

Updating the Kernel's Image

For a DLKM module configured as statically linked, use the kmupdate command to update the kernel's image with the newly generated kernel. Updating the kernel's image means preparing the system to move the newly generated kernel and its associated files into place during the next system shutdown and startup. The process is described in "Configuring a Statically Linked Module" on page 399.

Call kmupdate after first calling config without its -u option. If you include the -u option in the config invocation, there is no need to invoke kmupdate. The config -u command automatically invokes kmupdate.

To update the image of a statically linked module, execute the following kmupdate command:

    /usr/sbin/kmupdate /stand/build/vmunix_test
**Loading the Module**

For a DLKM module configured as dynamically loadable, use the `-L` option of the `kmadmin` command to load the module. The load operation initiated by the `kmadmin -L` command performs all tasks associated with link editing the module to the running kernel and making the module accessible to the system. Specifically, the load operation performs the following tasks:

1. ensures that the specified loadable module is registered

   **NOTE**

   All dynamically loadable modules need to be registered before they can be demand or auto loaded into the running kernel. `kmupdate` carries out this task.

2. checks what other modules the loadable module depends upon and automatically loads any dependent module that is not currently loaded

3. allocates space in active memory for the specified loadable module

4. loads the specified loadable module from the disk and link-edits it into the running kernel

5. relocates the loadable module's symbols and resolves any references the module makes to external symbols

6. calls the module's `_load()` entry point to do any module-specific initialization and setup

7. logically connects the module to the rest of the kernel, which is often accomplished with the help of module type-specific installation functions accessed through the module's wrapper code

To load a dynamically loadable module, execute the following `kmadmin` command:

```
/usr/sbin/kmadmin -L module_name
```

When the loading completes, an identifier (ID) number prints on the standard output to identify the module that was loaded.

If you want the system to automatically load certain dynamically loadable modules immediately after every system reboot, add the names of the modules to the `/etc/loadmods` file. At boot time, the...
Developing Dynamically Loadable Kernel Modules

Development Process

/sbin/init.d/kminit script will execute the kmadmin command and load the modules listed in /etc/loadmods.

Loading the Kernel

For a DLKM module configured as statically linked, use the -r option of the shutdown(1M) command to load the newly generated kernel by shutting down and rebooting the system. Executing the shutdown -r command terminates all currently running processes and reboots the system, which causes the new kernel configuration to take effect. At that time, all statically linked modules associated with the new kernel are accessible to the system.

To load the new kernel, move to a directory on the root volume such as the / directory and execute the following shutdown command:

/usr/sbin/shutdown -r

Querying the Module

Use the -Q, -q, -S, -s, or -k option of the kmadmin command to view status information about the DLKM module. For a DLKM module configured as dynamically loadable, you have the choice of displaying status for the module by its name or ID number.

To display a dynamically loadable module's status by name, execute the following kmadmin command:

/usr/sbin/kmadmin -Q module_name

To display a dynamically loadable module's status by ID, execute the following kmadmin command:

/usr/sbin/kmadmin -q module_id

Information returned by the -Q and -q options includes:

• the module's name
• the module's ID
• the module's pathname to its object file on disk
• the module's status (loaded or unloaded)
• the module's size
• the module's virtual load address
• the memory size of Block Started by Symbol (BSS) (the memory size of the un-initialized space of the data segment of the module's object file)
• the base address of BSS
• the module's reference or hold count (the number of processes that are currently using the module)
• the module's dependent count (the number of modules that are currently using the module; possible modules using the module are specified in the $DRIVER_DEPENDENCY section of the module's master file)
• the module's unload delay value (currently not used—always 0 seconds)
• the module's descriptive name
• the type of module (WSIO, STREAMS, or Misc)

Depending on the type of module, information on the module's block major number, character major number, and flags may also be printed.

To print the full status for all dynamically loadable modules currently loaded, execute the following kmadmin command:

/usr/sbin/kmadmin -S

To print the brief status for all dynamically loadable modules currently loaded, execute the following kmadmin command:

/usr/sbin/kmadmin -s

To print a list of all statically linked modules, execute the following kmadmin command:

/usr/sbin/kmadmin -k

Creating Device Special Files for the Module

Before devices supported by a driver-type DLKM module can be accessed by the system, you need to create special files in the /dev directory using the mknod(1M) command. You do not have to create these files every time you build and boot a new kernel; you only need to create them when you first add the new module. There must be a special file for each device on your system.

When you set up the special files for a newly added device driver, you
must specify the device type (character or block) and the major and minor device numbers. The kernel recognizes any single device by the major-minor number combination encoded in the device special files.

For a dynamically loadable driver, the system assigns a major number to the driver when it first registers with the system. For a statically linked driver, the system assigns a major number to the driver during system boot. The system remembers assigned major numbers from system boot to system boot.

To have the system dynamically assign a major number to your driver, follow these steps:

1. In the 
   $\text{DRIVER\_INSTALL}$ section of your driver’s master file, specify -1 in both the block major and char major fields.

2. In the 
   $\text{drv\_info\_t}$ structure of your driver source code, specify -1 in both the $\text{b\_major}$ and $\text{c\_major}$ fields of the $\text{drv\_info\_t}$ structure. Also, in the $\text{drv\_info\_t}$ structure, set the following bit values in the flags field: if you have a block driver, set the $\text{DRV\_BLOCK}$ value; if you have a character driver set the $\text{DRV\_CHAR}$ value; if your driver is both a block and a character driver, set both values.

To create device special files for your driver, follow these steps:

1. Use the 
   $\text{lsdev}(1M)$ command to identify the major number assigned to the device driver. $\text{lsdev}$ lists all device drivers configured into the kernel and their block and character major numbers.

   To extract the block major number from the display for a driver named $\text{mydriver}$, execute the following command:

   $\text{/usr/sbin/lsdev \(-h\ \-d\ mydriver\ |\ awk \'(print \$2)\')}$

   To extract the character major number from the display for a driver named $\text{mydriver}$, execute the following command:

   $\text{/usr/sbin/lsdev \(-h\ \-d\ mydriver\ |\ awk \'(print \$1)\')}$

   As an alternative and assuming a dynamically loadable driver, you can execute the following two commands to identify the major number assigned to a driver named $\text{mydriver}$:

   $\text{/usr/sbin/kmadmin \(-L\ mydriver\)}$

   $\text{/usr/sbin/kmadmin \(-Q\ mydriver\)}$

2. Construct a minor number for each device special file that you create for the driver. See the discussion, “Major and Minor Numbers,” in
Chapter 3 for more information about bit assignments and `dev_t`.

3. Use the `mknod` command to create the device special files for the driver.

   In the example below, `mydriver` was dynamically assigned the block and character major numbers 65 and 234, respectively. Its minor number, `0x026000`, is constructed like that of `instr0` (bits 8 through 15 encode 2 as the instance of the interface card, and bits 16 through 19 encode 6 of the device's address).

   ```
   /usr/sbin/mknod /dev/mydriver b 65 0x026000
   /usr/sbin/mknod /dev/mydriver c 234 0x026000
   ```

---

**NOTE**

You may be able to use the `insf` command to install special files in the `/dev` directory. If no options are specified, `insf` creates special files for all new devices in the system. New devices are those devices for which no special files have been previously created. See the `insf` (1M) manpage for more detail.

---

**Testing the Module**

Testing a DLKM module consists of testing all of its functions under a variety of operating conditions and for both configurations: dynamically loadable and statically linked. Debugging a module is largely a process of analyzing the code to determine what could have caused a given problem.

The most common debugging technique is monitoring the kernel code using the `printf()` function to print statements included in the module source code so that you know what your module is doing during run-time operation. There are also debuggers such as `Q4` to examine the kernel code. See the `Q4` description file in the `/usr/contrib/doc` directory for more detail.

You can use a combination of C preprocessor macros, conditional compilation, and control variables to turn `printf()` messages on or off. The template module code “dlclass.c” on page 411 includes a control variable named `dlclass_debug` to determine whether or not to generate the messages. All of the messages can be disabled at once by using `kmtune -s` to change the value of `dlclass_debug` to 0 before configuring `dlclass`. 
Developing Dynamically Loadable Kernel Modules

Development Process

Unloading the Module

Use the \texttt{-U} or \texttt{-u} option of the \texttt{kmadmin} command to unload a DLKM module configured as dynamically loadable. You have the choice of unloading the module by its name or its ID number.

The unloading operation logically disconnects the module from the running kernel and calls the module's \texttt{_unload()} entry point to perform any module-specific cleanup including:

1. canceling all outstanding calls to \texttt{timeout()}
2. disabling device interrupts
3. freeing all active memory allocated to the specified loadable module

To unload a dynamically loadable module by name, execute the following \texttt{kmadmin} command:

\begin{verbatim}
/usr/sbin/kmadmin -U module_name
\end{verbatim}

To unload a dynamically loadable module by ID number, execute the following \texttt{kmadmin} command:

\begin{verbatim}
/usr/sbin/kmadmin -u module_id
\end{verbatim}

Correcting Errors in the Module's Component Files

Correcting errors in the DLKM module is largely a process of analyzing the code to determine what could have caused a given problem. Some common errors are as follows:

- coding problems
- installation problems
- data structure problems
- timing errors
- corrupted interrupt stack
- accessing critical data
- overuse of local driver storage
- incorrect DMA address mapping
Chapter 10

Updating the Module’s Components

Use the \texttt{-u} option of the \texttt{kminstall} command to update a DLKM module’s components in the subdirectories of \texttt{/usr/conf} and \texttt{/stand}. You use this command when you modify one or more of the module’s component files. If you modify the module source code, you must first recompile the source code in accordance to “Creating the Module’s Component Files” on page 394.

\texttt{kminstall} expects to find the module’s component files in the current directory. If \texttt{module_name} already exists on the system, \texttt{kminstall} updates the module. If \texttt{module_name} does not exist on the system, \texttt{kminstall} prints a warning and adds the module’s components to the system.

When updating an existing module, \texttt{kminstall} takes the values of the tunable parameters and the $\texttt{CONFIGURATION}$ and $\texttt{LOADABLE}$ flags from the module’s current system description file and saves them to the module’s new system description file.

To update a DLKM module’s components, follow these steps:

1. Change directories to the directory containing the module’s component files.
2. Execute the following \texttt{kminstall} command:

   \texttt{/usr/sbin/kminstall -u module_name}

\texttt{kminstall} copies the module’s component files to the appropriate locations, thereby overwriting the same-named files at those locations.

Removing the Module’s Components from the System

Use the \texttt{-d} option of the \texttt{kminstall} command to remove a DLKM module’s components from the system. If the specified module is listed in the \texttt{/etc/loadmods} file, \texttt{kminstall} prints a warning message and removes the module entry from \texttt{/etc/loadmods}. If the specified module is currently loaded, \texttt{kminstall} tries to unload the module. If the unload fails, it prints a message and exits with an error; otherwise, \texttt{kminstall} tries to unregister the module. If the unregistration fails, \texttt{kminstall} prints a message and exits with an error.

To remove a DLKM module’s components, execute the following \texttt{kminstall} command:
Developing Dynamically Loadable Kernel Modules

Development Process

```
/usr/sbin/kminstall -d module_name
```

*kminstall* deletes the module’s components from the `/usr/conf` and `/stand` subdirectories.

---

**NOTE**

*kminstall* does *not* delete the original component files for the module.

---

If the removed DLKM module is statically linked into the running kernel, you will have to execute `config -u` to reconfigure the kernel once the module has been removed. Then you will have to shutdown and restart the system for the new configuration to take effect.
Sample DLKM WSIO Class Driver

This section presents a complete skeleton of a DLKM WSIO class driver in HP-UX. The files shown are:

- `dlclass.c`—the template character driver
- `master`—the master file for the sample character driver
- `system`—the system file for the sample character driver; `$CONFIGURE` and `$LOADABLE` flags are both set to `Y`
- `space.h`—a configuration file that sets two flags and allocates space for the sample character driver
- `Makefile`—a makefile for the template character driver; initiates the compiling of `dlclass.c` and places a copy of `mod.o` in the current working directory
- `mod32.mk`—a file called by `Makefile` for a 32-bit target machine; contains `cc` command and appropriate options to compile `dlclass.c`
- `mod64.mk`—a file called by `Makefile` for a 64-bit target machine; contains `cc` command and appropriate options to compile `dlclass.c`

This driver skeleton is a complete, working character driver that does not use any hardware. It can be added to any HP-UX 11.0 kernel and executed as an example.

The character driver supports a pseudo-device. Characters written to the device are passed to the kernel message buffer, so you have to execute the `dmesg(1M)` command to actually see the output. A read sequence results in the printing of the following hardcoded string compiled into the driver: “Reading from loadable WSIO driver ‘dlclass’.”

The character driver supports three tunable parameters:

- `dlclass_no_unload` (integer)—prevents unloading when non-zero; initially set to 0 in `$TUNABLE` section of `master` file
- `dlclass_debug` (integer)—writes debugging output into message buffer when non-zero; initially turned on (set to 1) in `$TUNABLE` section of `master` file
- `dlclass_bufsz` (integer)—size of internal write buffer; initially set to 40 characters in `$TUNABLE` section of `master` file
To install, configure, and load the character driver, log in as user root and perform the following steps:

1. Execute the following command to create a working directory for the driver:
   ```
   mkdir /usr/conf/dlclass.
   ```
2. Change directories to the /usr/conf/dlclass directory.
3. Create all component files for dlclass in addition to `Makefile`, `mod32.mk`, and `mod64.mk` in the /usr/conf/dlclass directory.
4. Execute the `make` command to generate the dlclass driver.

   The `make(1)` command examines the local `Makefile` and generates the dlclass driver according to the rules in the `Makefile`. It compiles `dlclass.c` into `mod.o`. It then installs and configures the driver. The `make` command also loads the driver and extracts the dynamically assigned major number used to create device special file `/dev/dlclass`.

You can read a message from dlclass by executing the following command:

   ```
   cat /dev/dlclass.
   ```

   The following message prints on your computer screen: “Reading from loadable WSIO driver ‘dlclass’.”

You can write a message to dlclass by executing a command such as:

   ```
   echo Hello Hello > /dev/dlclass.
   ```

   To see the output, execute the following command:

   ```
   dmesg | tail.
   ```

   The system responds as follows:

   ```
   dlclass> OPEN -- write buffer size = 40
   dlclass> 'Hello Hello'
   dlclass> CLOSE
   ```

   But if you write a message more than 40 characters, (e.g., `cp master /dev/dlclass`, for example), the `write()` system call will fail because dlclass cannot remember such a long word. You can change the maximum length of the message by using `kmtune -s` to set tunable parameter `dlclass_bufsz` to a higher value.

---

**NOTE**

When dlclass is unloaded (kmadmin -U dlclass), you can auto load dlclass by executing a command such as `cat /dev/dlclass`. 

---

Chapter 10

410
Sample Component Files

diclass.c

/*
 * Loadable/Unloadable Test Driver - dlcld
 */
#include <sys/types.h>
#include <sys/param.h>
#include <sys/errno.h>
#include <sys/malloc.h>
#include <mod_conf.h>
#include <sys/moddefs.h>
#include <sys/io.h>
#include <sys/wsio.h>

/* Entry Points */
int dlcld_install(void);
static int dlcld_load(void *drv_infop);
static int dlcld_unload(void *drv_infop);
static int dlcld_open(dev_t dev, int flags, intptr_t dummy,
    int mode);
static int dlcld_close(dev_t dev, int flags, int mode);
static int dlcld_read(dev_t dev, struct uio *uio);
static int dlcld_write(dev_t dev, struct uio *uio);

/* Local functions */
static int dlcld_init (void);
static void dlcld_linked_init (void);

/* Tunable Parameters */
extern int dlcld_no_unload;
extern int dlcld_debug;
extern int dlcld_bufsz;

/* message for dlcld_read() */
static char dlcld_msg[] = "Reading from loadable WSIO driver
'dlcld'.\n"

/*
 * Wrapper Table
 */
extern struct mod_operations gio_mod_ops;
static drv_info_t dlcld_drv_info;
extern struct mod_conf_data dlcld_conf_data;
/* module type specific data */
static struct mod_type_data dlclass_drv_link = {
    "dlclass - Loadable/Unloadable Test Module",
    (void *)NULL
};

static struct modlink dlclass_mod_link[] = {
    { &gio_mod_ops, (void *)&dlclass_drv_link }, /* WSIO */
    { NULL, (void *)NULL }
};

struct modwrapper dlclass_wrapper = {
    MODREV,
    dlclass_load,
    dlclass_unload,
    (void (*)(()))NULL,
    (void *)&dlclass_conf_data,
    dlclass_mod_link
};

/*
 * Driver Header
 */
static drv_info_t dlclass_drv_info = {
    "dlclass", /* type */
    "pseudo", /* class */
    DRV_CHAR|DRV_PSEUDO|DRV_SAVE_CONF|DRV_MP_SAFE,/* flags */
    -1, /* b_major */
    -1, /* c_major */
    NULL, /* cdio */
    NULL, /* gio_private */
    NULL, /* cdio_private */
};

static drv_ops_t dlclass_drv_ops = {
    dlclass_open, /* d_open */
    dlclass_close, /* d_close */
    NULL, /* d_strategy */
    NULL, /* d_dump */
    NULL, /* d_psize */
    NULL, /* d_mount */
    dlclass_read, /* d_read */
    dlclass_write, /* d_write */
    NULL, /* d_ioctl */
    NULL, /* d_select */
    NULL, /* d_option1 */
Developing Dynamically Loadable Kernel Modules

Sample DLKM WSIO Class Driver

```c
static wsio_drv_data_t dlclass_wsio_data = {
    "pseudo_dlclass", /* drv_path */
    T_DEVICE, /* drv_type */
    DRV_CONVERGED, /* drv_flags */
    NULL, /* drv_minor_build - field not used */
    NULL, /* drv_minor_decode - field not used*/
};

static wsio_drv_info_t dlclass_wsio_info = {
    &dlclass_drv_info,
    &dlclass_drv_ops,
    &dlclass_wsio_data,
};

static int (*dlclass_saved_init)();

/* LOAD */
static int
dlclass_load(void *arg)
{
    if (dlclass_debug)
        printf("dlclass> Loading\n");

    /* Use drv_info passed to us instead of static version */
    dlclass_wsio_info.drv_info = (drv_info_t *) arg;

    /* Register with WSIO */
    if (!wsio_install_driver(&dlclass_wsio_info))
    {
        printf("dlclass> wsio_install_driver failed!!\n");
        return (ENXIO);
    }

    /* Perform driver-specific initialization, but do not */
    /* call next function in the dev_init list. */
    (void) dlclass_init();
}
```
Developing Dynamically Loadable Kernel Modules
Sample DLKM WSIO Class Driver

    return (0);
}

/*
 * UNLOAD
 */
static int
diclass_unload(void *drv_infop)
{
    /* This function is only called when the administrator
     * attempts to unload the module and there are no open
     * devices using the module. If there is some reason that
     * the module should not be unloaded, check it now and
     * return non-zero.
     */
    if (dlclass_no_unload) {
        printf("dlclass> I’m BUSY\n");
        return (EBUSY);
    }

    /* Unregister with WSIO */
    if (wsio_uninstall_driver(&dlclass_wsio_info) ) {
        /* Uninstall failed! Return to a loaded,
         * functional state. */
        printf("dlclass> wsio_uninstall_driver failed!!\n");
        return (ENXIO);
    }

    /* Cancel pending timeouts, free allocated memory
     * and resources, etc.
     */
    if (dlclass_debug)
        printf("dlclass> Unloaded\n");

    return (0);
}

/*
 * INSTALL
 * This function is called if module is statically
 * linked.
 */
int
diclass_install(void)
Developing Dynamically Loadable Kernel Modules

Sample DLKM WSIO Class Driver

```c
{
    extern int (*dev_init)(void);

    /* Link my init function into chain */
    dlclass_saved_init = dev_init;
    dev_init = (int (*)(())) &dlclass_linked_init;

    /* Register driver with WSIO */
    return (wsio_install_driver(&dlclass_wsio_info));
}

/*
 * Device initialization Link
 * Called only for statically linked drivers to link
 * init routine into list.
 */
static void
dlclass_linked_init (void)
{
    /* Perform driver-specific initialization */
    (void) dlclass_init();

    /* Call next init function in chain */
    (void) (*dlclass_saved_init());
}

/*
 * Device initialization
 * This is common code for both statically linked and
 * dynamically loaded modules.
 */
static int
dlclass_init (void)
{
    /* This driver has no initialization code. But if it did,
     * it would go here!
     */

    return 0;       /* return value is ignored. */
}

/*
 * OPEN
 */
static int
dlclass_open(dev_t dev, int flag, intptr_t dummy, int mode)
```
Developing Dynamically Loadable Kernel Modules

Sample DLKM WSIO Class Driver

```c
if (dlclass_debug)
    printf("dlclass> OPEN -- write buffer size = %d\n",
           dlclass_bufsz);

return (0);
}

/*
 * CLOSE
 */
static int
dlclass_close(dev_t dev, int flag, int mode)
{
    if (dlclass_debug)
        printf("dlclass> CLOSE\n");

    return (0);
}

/*
 * READ
 */
static int
dlclass_read(dev_t dev, struct uio *uio)
{
    if (dlclass_debug)
        printf("dlclass>  READ\n");
    if (uio->uio_offset == sizeof(dlclass_msg))
        return (0);
    if (uio->uio_offset > sizeof(dlclass_msg))
        return (ENXIO);

    /* copy out my message */
    return (uiomove((caddr_t)((int)dlclass_msg+uio->uio_offset),
                    sizeof(dlclass_msg)-uio->uio_offset,
                    UIO_READ, uio) );
}

/*
 * WRITE
 */
static int
dlclass_write(dev_t dev, struct uio *uio)
{  
```
Developing Dynamically Loadable Kernel Modules

Sample DLKM WSIO Class Driver

caddr_t bufp;
char *end;
int len, err;

bufp = kmalloc(dlclass_bufsz+1, M_DYNAMIC, M_WAITOK);
len = MIN(uio->uio_resid, dlclass_bufsz);

/* print out user data */
if ( !(err = uiomove(bufp, len, UIO_WRITE, uio)) ) {
    bufp[len] = 0;
    printf("dlclass> '%s'", bufp);
    if (len == dlclass_bufsz) {
        printf("...??\n");
    }
}
kfree(bufp, M_DYNAMIC);
return (err);

master

* master file for "dlclass" module
 *

$VERSION
* Should start from 1, currently only 1 is correct value
1
$$$

$DRIVER_INSTALL
* Driver     Block major     Char major    Required for minimal system
* dlclass    -1    -1     $$$

$LOADABLE
$$$

$INTERFACE
base
$$$

Chapter 10  417
Developing Dynamically Loadable Kernel Modules

Sample DLKM WSIO Class Driver

$DRIVER_DEPENDENCY

$TYPE
dlclass wsio_class pseudo cpmi -1 -1

$TUNABLE
dlclass_no_unload DLCLASS_NO_UNLOAD 0
dlclass_debug DLCLASS_DEBUG 1
dlclass_bufsz DLCLASS_BUFSIZE 40 8

system

* system file for "dlclass" module
*

$VERSION 1
$CONFIGURE Y
$LOADABLE Y
$TUNABLE

space.h

/*
 * Tunable parameters for "dlclass" module
 */

int dlclass_no_unload = DLCLASS_NO_UNLOAD;
int dlclass_debug = DLCLASS_DEBUG;
int dlclass_bufsz = DLCLASS_BUFSIZE;

Sample Makefile

Makefile

MODULE=dlclass
CMODE=-Ae

DEV=/dev/dlclass
KERNEL_BITS=$(getconf KERNEL_BITS)

all: config dev
Developing Dynamically Loadable Kernel Modules
Sample DLKM WSIO Class Driver

config : mod.o
kminstall -a $(MODULE) 2> /dev/null
config -M $(MODULE) -u 2> /dev/null

dev : config $(DEV)

$(DEV) :
kadmin -L $(MODULE)
kadmin -Q $(MODULE) | awk '/Character Major/ { \rc=system("mknod $(DEV) c \$$3  0"); \exit rc; \}
}

mod.o:
make -f mod${KERNEL_BITS}.mk mod.o MODULE=$(MODULE) 
CMODE=$(CMODE)

load : config
kadmin -L $(MODULE) 2> /dev/null

status :
kadmin -s

clobber :
kminstall -d $(MODULE)
rm -f mod.o $(DEV)

mod32.mk
IDENT=-DJADE_BRINGUP_WORKAROUND -DSCSI_ALT_IP_DRIVER
-DSSEMAPHORE_DEBUG -DMPDEBUG -DOSDEBUG -DSYSCALLTRACE -DSTCP
-DHPONCPLUS -DNCFCTL -DDCOMB_FLIPPER -DINET_COSE
-DIVT_INTERCEPT -DLOCAL_SWITCH -DMULTIPLE_LOGICAL_HOSTS
-DRSVP_ISI -DMROUTING -DMMULTICAST -DAPPLETALK -DFDDI_VM
-DSPARSE_PDIR -DAUDIT
-DACLS -DPGPROF -DKI -DM -DLWSYSCALL -DICA_ON -DNEW_RDB
-DRDB -DLVMROOT -D_HPUX -D_LVM -DCONVERGED_IO

IDENT_TMP64=
IDENT_INT=-D__HIGHC__ -D__STDC_EXT__ -D_XPG4_EXTENDED
-D_HPUX_SOURCE -D__UNSUPPORTED -D_hp9000s800 -Dhp9000s800
IDENT_KERN=-D_KERNEL -DKERNELUNDEF=-U__hp9000s700
PAARCH=

Chapter 10 419
Developing Dynamically Loadable Kernel Modules

Sample DLKM WSIO Class Driver

DAFLAG=+DA1.1
DSFLAG=+DS2.0

RM= rm

COPTS=${IDENT} ${IDENT_TMP64} ${OPTIONS} ${UNDEF} ${IDENT_INT}
  ${IDENT_KERN}

OFLAGS=+XixdU

CCOPTS=-Wp,-H300000 ${OFLAGS} +Hx0 +R500 -Wl,-a,archive\${DAFLAG} ${DSFLAG} +ESsfc

CFLAGS= $(COPTS) $(CCOPTS)
CFLAG2= +ES1.Xindirect_calls

# Enable Assembler Warnings (New Flag with 10.0 assembler)
AFLAGS=-ew

# Default target
all: mod.o

mod.o: $(MODULE).c
@echo 'Compiling $(MODULE).c...'
$@$(RM) -rf mod.o
@$(CC) $(CMODE) -I. -I/usr/conf/h $(CFLAG2) -c -o mod.o
$(CFLAGS) 
  @$(NOGLOOPTS) $(MODULE).c

mod64.mk


IDENT_TMP64=-D__TEMP64__

420 Chapter 10
IDENT_INT=-D__HIGHC__ -D__STDC_EXT__ -D_XPG4_EXTENDED
-D_HPUX_SOURCE -D_UNSUPPORTED -D__hp9000s800 -Dhp9000s800

IDENT_KERN=-D_KERNEL -DKERNEL

UNDEF=-U__hp9000s700

PAARCH=64

DAFLAG=+DA2.0W
DSFLAG=+DS2.0

RM= rm

COPTS=${IDENT} ${IDENT_TMP64} ${OPTIONS} ${UNDEF} ${IDENT_INT}
\ ${IDENT_KERN}

OFLAGS=+XixdU

CCOPTS=-Wp,-H300000 ${OFLAGS} +Hx0 +R500 -Wl,-a,archive \${DAFLAG} ${DSFLAG} +ESsfc

CFLAGS= ${COPTS} ${CCOPTS}
CFLAG2= +ES1.Xindirect_calls

# Enable Assembler Warnings (New Flag with 10.0 assembler)
AFLAGS=-ew

# Default target
all: mod.o

mod.o: $(MODULE).c
@echo ‘Compiling $(MODULE).c...’
@$(RM) -f mod.o
@$(CC) $(CMODE) -I. -I/usr/conf/h $(CFLAG2) -c -o mod.o
\ $(CFLAGS) \n    @$(NOGLOOPTS) $(MODULE).c
Developing Dynamically Loadable Kernel Modules
Sample DLKM WSIO Class Driver
11 How To Make Pre 11.0 Drivers 64-Bit Safe
We recommend that you modify existing 32-bit drivers to be 64-bit clean so that they can be compiled to run in either type of kernel.

**NOTE**
For information about modification of Pre-Release 10.0 drivers, see the previous (HP-UX 10.20) driver Development Guide, HP Part No. B2355-90066

**NOTE**
For information about specific driver entry points see the information for that entry point in a previous chapter; for example, `driver_ioctl()`, in “Writing a driver_ioctl() Routine” on page 131, Chapter 5.
How To Make Pre 11.0 Drivers 64-Bit Safe

Introduction

HP-UX 11.0 has two versions of the operating system kernel: a 32-bit kernel and a 64-bit kernel. The 64-bit kernel extends the capabilities of the 32-bit kernel in several ways. Among the extensions are the ability to address larger than 4 gigabytes of physical memory, map up to 16 terrabytes of virtual address space for 64-bit application programs and allow 32-bit and 64-bit applications to coexist on the same system.

To integrate your I/O driver with the 64-bit kernel requires an understanding of the differences between the 32-bit and 64-bit data models. It also requires that you make your I/O driver source “64-bit clean” and robust enough to deliver predictable results when executing in the 64-bit kernel environment.

Important Terms and Concepts

This section begins by defining important terms and concepts essential for migration of drivers to the 64-bit environment. Then it describes the 64-bit data model. Next, it discusses general guidelines, and finally discusses the I/O address space.

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.

ILP32 C language data model where int, long, and pointer data types are 32 bits in size.

LP64 C language data model where the int data type is 32 bits wide, but long and pointer data types are 64 bits wide.

SIO Server I/O: I/O environment for port-server drivers with origins in S/800 systems.

WSIO Workstation and Server I/O: I/O environment for reentrant drivers with origins in S/700 systems and converged with S/800 10.x systems.
The 64-bit Data Model

ILP32 is a C language data model where int, long, and pointer data types are 32 bits wide. This is the data model commonly used by 32-bit operating systems, including the 32-bit HP-UX.

LP64 is another C language data model where the int data type remains 32 bits wide, but long and pointer data types are scaled up to 64 bits. This is the data model that has been adopted by HP for its 64-bit HP-UX implementation.

Table 11-1, “Summary of LP64 and ILP32 Data Models,” summarizes the two data models.

<table>
<thead>
<tr>
<th>Data type</th>
<th>LP64 bit size</th>
<th>ILP32 bit size</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>long</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>long long</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>pointer</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>float</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>double</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>long double</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>struct</td>
<td>depends on members</td>
<td>depends on members</td>
</tr>
<tr>
<td>enum</td>
<td>32</td>
<td>32</td>
</tr>
</tbody>
</table>
LP64 Considerations

Size differences between the ILP32 and LP64 data models occur with long and pointer data types. As a consequence, the default integral data type int can differ in size from long. This subtle difference can cause results unintended by the programmer.

Consider the following example:

```c
main() {
    long L = -1;
    unsigned int i = 1;
    if (L > i)
        printf ("L greater than i\n");
    else
        printf("L not greater than i\n");
}
```

Under ANSI C integral promotion rules, if `sizeof(int)` equals `sizeof(long)` this will print

```
L greater than i
```

However, if `sizeof(int)` is less than `sizeof(long)` this will print

```
L not greater than i
```

Both results are ANSI conforming, correct, and consequences of the value preserving integral promotion rules of ANSI. If the same code is compiled in K&R mode, the unsigned preserving integral promotion rules of K&R will result in the program printing “L greater than i” with ILP32 and LP64.

Suppose we modify this example and change L to int. Under ANSI and K&R integral promotion rules, this will print “L greater than i” for both ILP32 and LP64.

This example illustrates that caution needs to be exercised when a data type is changed to or from long. The HP C compiler provides the option +M to identify code where ANSI and K&R may differ.

Another consideration is the alignment of data objects. With LP64, structures that contain long or pointer data types will be aligned to a double word offset. As an example, consider the following:
The 64-bit Data Model

```c
struct vals {
    int intval;
    long longval;
    int endval;
};
```

The data member `longval` has the strictest alignment requirement of the data members in the structure. In LP64 mode, the entire structure and `longval` are double word aligned. A 32-bit gap will exist between `intval` and `longval`. The compiler may also pad the size of the structure to the next double word size.
General Guidelines

64-bit Clean Headers

Header files contain the data declarations, structures, constants, macros, function prototypes, and external data objects that are the interfaces to modules. The same header files are expected to be compiled with source code for 32-bit and 64-bit drivers, and possibly with 32-bit and 64-bit libraries and applications.

To make this possible, header files need to be examined for declarations and usage that may not be compatible between the ILP32 and LP64 data models. This is referred to as making header files 64-bit clean.

Here are the general guidelines to clean an I/O driver header file:

- Examine declarations where long (or a variant of long) is specified. Where appropriate, fix the size of the declaration by replacing long or a variant of long by int (or a variant of int). Cases where this may be appropriate include application visible data structures that must be sized the same for both ILP32 and LP64, driver data structures that ought to be kept from growing unnecessarily large, and hardware data structures that must be fixed in size with 32-bit data types.

- Examine declarations where int (or a variant of int) is specified. Where appropriate, scale the size of the declaration by replacing int (or a variant of int) by long (or a variant of long). Cases where this may be appropriate include offsets that must displace greater than 4 GBytes in a 64-bit kernel, storage that is overloaded to store a pointer value, and storage for machine register values.

- Examine declarations where pointer data types are specified. If the pointer is a pointer to a function, specify the full ANSI function prototype with all arguments declared. This enables ANSI code to be type checked against the function prototype and expose incompatibilities at compile-time. If the pointer is in an application visible data structure, things get complicated because the application may be a 32-bit application (compiled with ILP32 data types) and your driver may be executing in the 64-bit kernel (compiled with LP64 data types). This situation is discussed in “Writing a driver_ioctl() Routine” on page 131, Chapter 5.
Useful Data Types

int32_t and uint32_t

The data types int32_t and uint32_t represent storage that must be fixed in size to 32 bits. They are declared in the header file _inttypes.h as:

typedef int int32_t;
typedef unsigned int uint32_t;

Consider the following data structure from the 10.20 header file diskio.h:

typedef struct {
    long lba; /* capacity in DEV_BSIZE blocks */
} capacity_type;

Contained in the structure is the data member lba declared as long. This is an application visible data structure that should be fixed in size to avoid compatibility problems between a 32-bit application and a 64-bit driver. For example,

typedef struct {
    int32_t lba; /* capacity in DEV_BSIZE blocks */
} capacity_type;

Since lba is intended to store values greater than or equal to zero and negative values are never stored, the correct declaration for lba is an unsigned type. The following is the 64-bit clean version of capacity_type in release 11.0:

typedef struct {
    uint32_t lba; /* capacity in DEV_BSIZE blocks */
} capacity_type;

intptr_t and uintptr_t

The data types intptr_t and uintptr_t represent storage that must scale in size with a pointer data type. They are declared in the header file _inttypes.h as:

typedef long intptr_t; typedef unsigned long uintptr_t;

Consider the following macro from the 10.20 header file cpu.h:
#define ALIGN_IOBUF(P) ((char *)((uintptr_t)(P) & ~CPU_IOLINE-1))

The pointer P is cast as (int), but the cast does not scale in size with the pointer. The cast is changed to (uintptr_t) in release 11.0 as shown below:

#define ALIGN_IOBUF(P) ((char *)((int)(P) + CPU_IOLINE - 1) & ~((uintptr_t)CPU_IOLINE - 1)))

The cast in the second line explicitly promotes the value CPU_IOLINE - 1 to uintptr_t before complementing the value.

`ptr32_t`

The data type `ptr32_t` represents storage for a 32-bit pointer and is declared in _inttypes.h as:

typedef uint32_t ptr32_t;

For an example see “Writing a driver_ioctl() Routine” on page 131, Chapter 5.

`int64_t` and `uint64_t`

The data types `int64_t` and `uint64_t` represent storage that must be fixed in size to 64 bits. They are declared in the header file _inttypes.h as:

typedef long long int64_t;
typedef unsigned long long uint64_t

Be aware that application code compiled with strict ANSI (compiler option -Aa) will not recognize the data type `long long`. However, `long long` is accepted in extended ANSI (compiler option -Ae) and K&R (compiler option -Ac) modes.

__LP64__

__LP64__ is defined when source code is compiled with LP64 data types. #ifdef __LP64__ may be useful where there are special LP64 considerations. For example, consider the following data structure in the 11.0 header file dma_A.h:

typedef struct {
    lock_t *lock;
    dma_A_chain_type *chain_list;
}
How To Make Pre 11.0 Drivers 64-Bit Safe

General Guidelines

```
#ifdef __LP64__
    uint32_t pad[12]; /* align it to 64 bytes */
#else /* !__LP64__ */
    uint32_t pad[14]; /* align it to 64 bytes */
#endif /* !__LP64__ */
}
dma_A_pool_t;
```

#ifdef __LP64__ is used to pad the data structure so that the size is 64 bytes in both ILP32 and LP64 modes.

Hardware Considerations

I/O drivers often map data structures into I/O hardware registers and memory areas that are accessed by I/O hardware. Care must be exercised to ensure that these structures are fixed in size when compiled with LP64 data types.

Consider the data structure compl_head in the 10.20 header file llio.h:

```
struct compl_head {
    int sema;
    struct compl_entry *link;
    int filler[2];
};
```

A compl_head structure is updated by hardware when an I/O request completes. The link data member is viewed by hardware as a 32-bit value. With LP64, however, link will be scaled to 64 bits.

The first step is to fix the size of the structure. In release 11.0, the structure is declared as:

```
struct compl_head {
    uint32_t sema;
    uint32_t link;
    uint32_t filler[2];
};
```

The next step is to examine code that accesses the compl_head and make corrections as needed.
Software Considerations

While data structures that map onto hardware and must be fixed in size are easily identified, data structures with software considerations can be much more subtle. Rather than elaborating on the many possible programming pitfalls, we present an example. Consider messages sent and received by I/O drivers in the SIO driver environment. In release 10.20, the message header is declared as:

```c
typedef struct {
    shortint    msg_descriptor;
    shortint    message_id;
    int         transaction_num;
    port_num_type from_port
} llio_std_header_type;
```

Notice that no data member in the structure is larger than 32 bits. As such, the size of `llio_std_header_type` is 12 bytes. In ILP32 and LP64 modes, `sizeof(llio_std_head_type)` returns the value 12.

The programming assumption that causes a problem is that the message body immediately follows the header. This assumption holds with ILP32 data types; but with LP64 data types, the message body happens to contain 64-bit data members that cause the compiler to align the body to a 64-bit boundary. The header is also aligned to a 64-bit boundary and this creates a 32-bit hole between the header and body of the message. The message is declared as:

```c
typedef struct {
    llio_std_header_type msg_header;
    union {
        creation_info_type  creation_info;
        do_bind_req_type    do_bind_req;
    } union_name;
} io_message_type;
```

To send a message, SIO drivers add the `sizeof(llio_std_header_type)` to the size of the specific message.
General Guidelines

body type being sent. In LP64 mode, the calculated size of messages are 4 bytes less than the programmer intended.

A possible solution is to explicitly pad the message header as follows:

typedef struct {
    shortint msg_descriptor;
    shortint message_id;
    int transaction_num;
    port_num_type from_port;
    #ifdef __LP64__
        int filler; /* pad to 64-bit boundary */
    #endif /* __LP64__ */
} llio_std_header_type;

ANSI Function Prototypes

With LP64, programmers can no longer assume that int, long, and pointer data types are the same size, and that these data types can be mixed across function call arguments and function return types. Many kernel functions return pointer or long data types. Without the appropriate function prototypes in scope, the compiler incorrectly truncates the return type to int. Programmers need to provide appropriate ANSI function prototypes (or K&R forward declarations) to their I/O drivers.

Compile time errors will result, however, if ANSI function prototypes are visible to the K&R compiler. To prevent such errors, kernel header files use the __() macro to declare prototypes. The macro is defined in the header file stdsym.h as follows:

#define __(_arg) arg

To illustrate the__() macro, consider the following function prototype:

void *fl__((long arg1, int arg2));
Note the double parenthesis surrounding the arguments. This notation is equivalent to the cumbersome form it replaces:

```c
#if defined (_PROTOTYPES)
void *fl(long arg1, int arg2);
#else
void *fl();
#endif
```

Many kernel header files have incorporated ANSI function prototypes in release 11.0. Programmers should make sure that function prototypes are in scope for all kernel interfaces called by their drivers by including the appropriate header files.

To take full advantage of function prototypes, set the compiler option to `-Ae` to compile in extended ANSI mode. If your driver is compiled K&R, now is the time to convert to ANSI.

**kern_svcs.h header file**

A new header file, `kern_svcs.h`, has been added to `/usr/include/sys` and `/usr/conf/h`. This header file declares ANSI function prototypes of common kernel services.

Driver writers do not need to explicitly include `kern_svcs.h`. WSIO drivers include `kern_svcs.h` when `wsio.h` is included.

When porting a driver you should always check your routine declarations and parameters against `kern_svcs.h`.

**conf.h header file**

Among the changes made to `conf.h` are the inclusion of ANSI function prototypes for the function pointers declared in the `cdevsw` and `bdevsw` structures. Driver entry points are expected to match the function prototypes declared therein.

**dma.h header file**

Several changes have been made to the `dma.h` header file. WSIO specific macro definitions have been moved to `wsio.h`. Additionally, function prototypes have been added to the function pointer declarations in the `io_map_cntrl` structure. These prototypes enable type checking of arguments that are passed to the WSIO and SIO mapping services.
Consider the `wsio_dma_alloc` macro:

```c
#define wsio_dma_alloc(Isc, Iova) \
    (*Isc->map_funcs->dma_alloc)(Isc->map_funcs->arg, Iova)
```

The macro calls the function pointed to by `dma_alloc` in the `io_map_cntl` structure. The second argument to the macro is `Iova` which is prototyped by `dma_alloc` as `caddr_t *`. Suppose the driver calls this macro with the following code:

```c
uint32_t iova; /* only 32 bits of storage */
caddr_t pva = wsio_dma_alloc(isc, &iova);
```

This is clearly a type mismatch (and a programming error with LP64) for which the ANSI C compiler will generate a warning message. For DMA hardware that uses 32-bit `iova` values, the driver can do the following:

```c
uint32_t iova; /* only 32 bits of storage */
caddr_t tmp_iova;
caddr_t pva = wsio_dma_alloc(isc, &tmp_iova);
iova = (uint32_t)tmp_iova;
```

### uio.h header file

The `uio.h` header file is the declaration of the `iovec` structure. A pointer to the `iovec` structure is used not only in the `uio` structure, but is also specified as an argument type to the DMA mapping services. The `iovec` structure is declared as:

```c
struct iovec {
    caddr_t  iov_base;
    size_t   iov_len;
};
```

With LP64 data types, the `iov_base` and `iov_len` fields will each require 64 bits of storage and must be aligned to a double word (64bit) address. I/O drivers that declare alternate data structures that map onto the `iovec` structure may require changes. For example, consider a driver that declares a data structure that represents an address/count pair as:

```c
struct ac_pair {
    uint32_t  iov_base;
    uint32_t  iov_len
};
```
The driver must not pass a pointer to an `ac_pair` as an argument to the DMA mapping services where a pointer to an `iovec` is expected.
How To Make Pre 11.0 Drivers 64-Bit Safe

General Guidelines
Example Drivers
Example Drivers

This appendix will contain examples of drivers in the final release, as follows:

- A Device Driver for a RAM Disk,
- A DLKM version of the RAM Disk Driver
- A Monolithic Driver for an IKON 10115 Hardcopy PCI Interface card,
- A Combined User and Kernel Driver for an IKON 10115 Hardcopy PCI Interface
Ram Disk Static Module Device Driver

The following pages show the workstation static module ram disk driver

Ram Disk Static Device Driver: Program Source

/*-----------------*/
/* rdisk.c         */
/* RAM disk driver */
/*-----------------*/

#include "../h/types.h"
#include "../h/param.h"
#include "../h/buf.h"
#include "../h/io.h"
#include "../h/ioctl.h"
#include "../h/errno.h"
#include "../h/conf.h"
#include "../h/sysmacros.h"
#include "../h/diskio.h"
#include "../h/dir.h"
#include "../h/moddefs.h"
#include "../h/mod_conf.h"
#include "../wsio/wsio.h"/* if KERN def’d expects eeprom.h in wsio */

#define RAMDISK_SIZE2*1024*1024

static caddr_t ram_pointer;
static space_t ram_sid;
static uint32_t ram_size;
static int nblks;
int rdisk_install();
void rdisk_init();
int rdisk_open();
void rdisk_close();
int rdisk_ioctl();
int rdisk_read();
int rdisk_write();
void rdisk_strategy();
extern int (*dev_init)();
int (*rdisk_saved_dev_init)();
Example Drivers

Ram Disk Static Module Device Driver

static drv_info_t rdisk_drv_info = {
  "rdisk",
  "pseudo",
  DRV_CHAR | DRV_BLOCK | DRV_PSEUDO,
  -1,
  -1,
  NULL,
  NULL,
  NULL,
};
static drv_ops_t rdisk_drv_ops = {
  rdisk_open,
  rdisk_close,
  rdisk_strategy,
  NULL,
  NULL,
  NULL,
  rdisk_read,
  rdisk_write,
  rdisk_ioctl,
  NULL,
  NULL,
  NULL,
  NULL,
  NULL,
  0,
};

static wsio_drv_data_t rdisk_data = {
  "rdisk_ramdisk",
  T_UNKNOWN,
  DRV_CONVERGED,
  NULL,
  NULL
};
static wsio_drv_info_t rdisk_wsio_drv_info = {
  &rdisk_drv_info,
  &rdisk_drv_ops,
  &rdisk_data
};

/*******/
/* RDISK_INSTALL */
/*******/
int rdisk_install() {
    rdisk_saved_dev_init = dev_init;
    dev_init = rdisk_init;
    return (wsio_install_driver(&rdisk_wsio_drv_info));
}

/*---------------------------------------------------------------------
/* RDISK_INIT */
/*---------------------------------------------------------------------
void
rdisk_init()
{
    ram_size = RAMDISK_SIZE;
    ram_pointer = sys_memall((long)ram_size);
    if (ram_pointer) {
        ram_sid = ldsid(ram_pointer);
        nblks = RAMDISK_SIZE >> DEV_BSHIFT;
    } else {
        nblks = 0;
        ram_size = 0;
    }
    printf("rdisk_init: nblks %x, size %x\n", nblks, ram_size);
    (*rdisk_saved_dev_init)();
}

/*---------------------------------------------------------------------
/* RDISK_OPEN */
/*---------------------------------------------------------------------
rdisk_open(dev,flag)
dev_t dev;
int flag;
{
    if (nblks == 0)
        return ENOSPC;
    return 0;
}

/*---------------------------------------------------------------------
/* RDISK_READ */
/*---------------------------------------------------------------------
int
rdisk_read(dev,uio)
dev_t dev;
struct uio *uio;
{
Example Drivers
Ram Disk Static Module Device Driver

```c
if ((uio->uio_offset & DEV_BMASK) &&
    ((uio->uio_fpflags & FBLKSEEK) == 0))
    return(ENXIO);
return physio(rdisk_strategy,NULL,dev,B_READ,minphys,uio);
}

/*-------------*/
/* RDISK_WRITE */
/*-------------*/
int
rdisk_write(dev,uio)
    dev_t dev;
    struct uio *uio;
{
    if ((uio->uio_offset & DEV_BMASK) &&
        ((uio->uio_fpflags & FBLKSEEK) == 0))
        return(ENXIO);
    return physio(rdisk_strategy, NULL, dev, B_WRITE, minphys,
                  uio);
}

/*----------------*/
/* RDISK_STRATEGY */
/*----------------*/
void
rdisk_strategy(bp)
    struct buf *bp;
{
    size_t size;
    caddr_t start;
    space_t bp_sid;

    bp_sid = bvtospace(bp,bp->b_un.b_addr);
    start = ram_pointer + bp->b_offset;
    /* check transfer in volume and nonzero length */
    if (((start >= ram_pointer + ram_size) || (ram_size == 0)) {
        bp->b_resid = bp->b_bcount;
        bp->b_flags |= B_ERROR;
        bp->b_error = ENOSPC;
        iodone(bp);
        return;
    }
    /* truncate if transfers goes past end of volume */
    size = MIN((int)((uint32_t)ram_pointer + ram_size) -
                (uint32_t)start,
                bp->b_bcount);
```
/* abort if not a multiple of block size */
if (bp->b_bcount & DEV_BMASK) {
    bp->b_resid = bp->b_bcount;
    bp->b_flags |= B_ERROR;
    bp->b_error = ENXIO;
    iodone(bp);
    return;
}
/* do the transfer */
if (bp->b_flags & B_READ)
    privlbcopy(ram_sid,start,bp_sid,bp->b_un.b_addr,bp->b_bcount);
else
    privlbcopy(bp_sid,bp->b_un.b_addr,ram_sid,start,bp->b_bcount);
    iodone(bp);
}

/*******
/* RDISK_IOCTL */
/*******
int
rdisk_ioctl(dev,cmd,arg,flag)
dev_t dev;
int cmd;
caddr_t arg;
int flag;
{
capacity_type *cap;
disk_describe_type *describe;
int size = (cmd & IOCSIZE_MASK) >> 16;

    switch (cmd & IOCCMD_MASK) {
    case DIOC_DESCRIBE & IOCCMD_MASK:
        if (cmd != DIOC_DESCRIBE && size != 32)
            return EINVAL;
        describe = (disk_describe_type *)arg;
        (void) strcpy(describe->model_num,"RDISK_DISK");
        describe->intf_type = UNKNOWN_INTF;
        describe->maxsva = nblks - 1;
        describe->lgbksz = DEV_BSIZE;
        describe->dev_type = DISK_DEV_TYPE;
        describe->flags = 0;
        printf("rdisk:DIOC_DESCRIBE maxsva = 0x%x,lgbksz = 0x%x\n", describe->maxsva,describe->lgbksz);
        return 0;
    }
Example Drivers

Ram Disk Static Module Device Driver

```c
switch (cmd) {
    case DIOC_CAPACITY:
        cap = (capacity_type *) arg;
        cap->lba = nblks;
        printf("rdisk_ioctl:DIOC_CAPACITY cap->lba = 0x%x\n", cap->lba);
        return 0;
    }
    return ENXIO;
}

/*-------------*/
/* RDISK_CLOSE */
/*-------------*/
void
rdisk_close(dev)
dev_t dev;
{
    return;
}

Ram Disk Static Device Driver: 64-bit makefile

ROOT = /usr/conf/lib
LIBS = "
 $(ROOT)/libuipc.a 
 $(ROOT)/libufs.a 
 $(ROOT)/libnfs.a 
 $(ROOT)/libcdfs.a 
 $(ROOT)/libpci.a 
 $(ROOT)/libhp-ux.a 
 $(ROOT)/libdklm.a 
 $(ROOT)/librpc.a 
 $(ROOT)/liblan.a 
 $(ROOT)/libinet.a 
 $(ROOT)/libtun.a 
 $(ROOT)/libtelnet.a 
 $(ROOT)/libnms.a 
 $(ROOT)/libbtlan6.a 
 $(ROOT)/libfcgsc.a 
 $(ROOT)/libvxfs_base.a 
 $(ROOT)/liblvm.a 
 $(ROOT)/libstream.a 
 $(ROOT)/libttrtio.a 
 $(ROOT)/libautofs.a 
 $(ROOT)/libbtlan3.a "
```
$(ROOT)/btlan3_dbg.o \
$(ROOT)/libmaclan.a

#CAUTION: Do not modify below.
SMODWORK = mod_wk.d
SMODSTAT = /usr/conf/km.d
SYMTAB = dlkm.vmunix_test/symtab
MMASTER = /usr/conf/master.d
KMODS =
KMODSTUBS =

CONF=conf
TUNE=tune
IDENT=--DLP64_TMP_NODBG --DSPP_RUNWAY.ERR_ENABLED --DSPP_OBP_BOOT
--D__ROSEVILLE__ --DKGDB_ON --DSTCP --DPFROMCPLUS --D__ROSE__
--DNEW_MFCFLW --DCOMB_FLIPPER --DINET_COSE --DIVFT_INTERCEPT
--DMULTICAST --DAPPLETALK --DFDDI_VM --DAUDIT --DACLS --DPGPROF --DKI
--DMP --DLWYSYSCALL --DNEW_RDB --DRDB --D_LVMROOT --D_LVM
--DCONVERGED_IO
IDENT_TMP64=--D_TEMP64__
ROOT=/usr/conf/lib
IDENT_INT=--D__HIGHC__ --D__STDC_EXT__ --D_KERNEL_BUILD
--D_XPG4_EXTENDED --D_HPUX_SOURCE --D_UNSUPPORTED --D_hp9000s800
--Dhp9000s800
IDENT_KERN=--D_KERNEL --DKERNEL
UNDEF=--U_hp9000s700
DAFLAG=+DA2.0W
DSFLAG=+DS2.0
SPACED=space_LP64
MASTERD=master_LP64
MAPFILE=/usr/conf/gen/mapfile
COPTS=${IDENT} ${IDENT_TMP64} ${OPTIONS} ${UNDEF} ${IDENT_INT}
${IDENT_KERN}
OFLAGS=+XixdU
K_CCOPTS=--Wp,-H300000 ${OFLAGS} +Hx0 +R500 -Wl,-a,archive \ 
  ${DAFLAG} ${DSFLAG} +ESsfc +ESssf
K_CCOPTS_NOOPT=--Wp,-H300000 ${OFLAGS_NOOPT} +Hx0 +R500
  -Wl,-a,archive \ 
  ${DAFLAG} ${DSFLAG} +ESsfc +ESssf
CFLAGS= -w ${COPTS} ${K_CCOPTS}
CFLAG2= +ES1.Xindirect_calls
HPUX=vmunix_test
LIBHPUX= $(ROOT)/libhp-ux.a
SPACE= /stand/build/space.h

Appendix A 447
Example Drivers

Ram Disk Static Module Device Driver

SPACEDIR= /usr/conf/space.h.d
DIR=.
ORDER=/usr/conf/gen/linkorder
LOADOPTS= -PF ${ORDER} -P -noshared -k ${MAPFILE} +nodefaultmap
+Accept \ 
  TypeMismatch -e rdb_bootstrap -a archive
LIBK=
SPECFLAGS= -L$(ROOT) ${LIBK}
LIBUSRDRV=`if [ -f $(ROOT)/libusrdrv.a ]; then 
  echo $(ROOT)/libusrdrv.a; fi;`

# Default target
#all: $(DIR)/$(HPUX)
all: rdisk.o

LOCORE=locore.o
DBGS= virt_mem.o telnet.o wtermio.o termio3.o termio2.o
termio1.o termio0.o \ 
  strtdio.o streams.o security.o multiproc.o proc_man.o nfs.o
  fddi4_dbg.o \ 
  fddi3_header.o xport_diag.o netdiag.o btlan3_dbg.o
lan_dbg1.o lan_dbg0.o \ 
  low_level.o pci_debug.o fc_gsc.o fcms.o wsio_scsi.o
wsio_drivers.o \ 
  lv_lvm.o vxfs.o file_sys.o sio_drivers4.o sio_drivers3.o
sio_drivers2.o \ 
  sio_drivers1.o io.o
OFILES=
SZLIMIT= -l 29308928

$(SPACE): $(SPACEDIR)
@- # FORCEBUILD=$(FORCEBUILD)
-@$(RM) -f $@
$(MKDIR) -p $(@D)
@echo "*/n * space.h*n * Copyright 1994 by Hewlett Packard" > $@
@echo " */" >> $@
@for file in `$(LS) $(SPACEDIR) | $(GREP) -E '^[^\.-#]+\.h$'`; do 
  echo "#include \"$(SPACEDIR)/$file\"" >> $@; 
done

##
Ram Disk Static Module Device Driver

APPENDIX A

## Example Drivers

### Ram Disk Static Module Device Driver

#### HP-UX Kernel Makefile

```bash
## HP-UX Kernel Makefile

@(#) config.sys $Date: 1998/06/19 13:53:41 $Revision: r11ros/3 PATCH_11.00 (PHKL_15687)

# Set these to alternate path if we’re running pre-10.0
BIN=/usr/bin
CKRN=/usr/ccs/bin
SBIN=/usr/sbin
SBINS=/usr/sbin
AWK=$(BIN)/awk
SHELL=/sbin/sh
SH=/sbin/sh
RM=$(BIN)/rm
CHMOD=$(BIN)/chmod
MKDIR=$(BIN)/mkdir
GREP=$(BIN)/grep
LS=$(BIN)/ls
AR=TMPDIR=/tmp $(CKRN)/ar
CC=$(CKRN)/cc
LD=$(CKRN)/ld
SZ=$(CKRN)/size
UTIL=/usr/conf/gen
MKFUNCNAMES=/usr/lbin/sw/bin/mkfuncnames
FUNCNAMES=/stand/build/function_names

KSYMTAB= $(SBINS)/kernsymtab
CHKMIF= $(SBINS)/kmckintfc

rdisk.o: rdisk.c $(SPACE)
@- # FORCEBUILD=$(FORCEBUILD)
@echo 'Compiling rdisk.c...'
@$(RM) -f rdisk.o
@$(CC) -I. \
$(CFLAG2) -c -o rdisk.o $(FLAGS) $(NOGLOOPTS) rdisk.c

Root = /usr/conf/lib
LIBS = \
Example Drivers

Ram Disk Static Module Device Driver

$(ROOT)/libuipc.a 
$(ROOT)/libufs.a 
$(ROOT)/libnfs.a 
$(ROOT)/librpc.a 
$(ROOT)/libcdcf.s.a 
$(ROOT)/libpci.a 
$(ROOT)/libhp-ux.a 
$(ROOT)/libdlkm.a 
$(ROOT)/liblan.a 
$(ROOT)/libinet.a 
$(ROOT)/libnet.a 
$(ROOT)/libtun.a 
$(ROOT)/libtelnet.a 
$(ROOT)/libnm.s.a 
$(ROOT)/libfcgsc.a 
$(ROOT)/libvxfs_base.a 
$(ROOT)/liblvma.a 
$(ROOT)/libstream.a 
$(ROOT)/libstrtio.a 
$(ROOT)/libautof.s.a 
$(ROOT)/libbtlan3.a 
$(ROOT)/btlan3_dbg.o 
$(ROOT)/libmaclan.a

#CAUTION: Do not modify below.
SMODWORK = mod_wk.d
SMODSTAT = /usr/conf/km.d
SYMTAB = dlkm.vmunix_test/symtab
MMMASTER = /usr/conf/master.d
KMODS =
KMODSTUBS =
CONF=conf
TUNE=tune
IDENT_TMP64=
ROOT=/usr/conf/lib
IDENT_INT=-D__HIGHC__ -D__STDC_EXT__ -D_KERNEL_BUILD -D_XPG4_EXTENDED -D_HPUX_SOURCE -D_UNSUPPORTED -D_hp9000s800 -Dhp9000s800
IDENT_KERN=-D_KERNEL -DKERNEL
UNDEF=-U__hp9000s700
DAFLAG=+DA1.1
DSFLAG=+DS2.0
SPACED=space
MASTERD=master
MAPFILE=
COPTS=${IDENT} ${IDENT_TMP64} ${OPTIONS} ${UNDEF} ${IDENT_INT} ${IDENT_KERN}
OFLAGS=+XixdU
K_CCOPTS=-Wp,-H300000 $(OFLAGS) +Hx0 +R5000 -W1,-a,archive \\
   ${DAFLAG} ${DSFLAG} +ESsfc +ESssf
K_CCOPTS_NOOPT=-Wp,-H300000 $(OFLAGS_NOOPT) +Hx0 +R5000
   -W1,-a,archive \\
   ${DAFLAG} ${DSFLAG} +ESsfc +ESssf
CFLAGS= -w $(COPTS) $(K_CCOPTS)
CFLAG2= +ES1.Xindirect_calls
HPUX=vmunix_test
LIBHPUX= $(ROOT)/libhp-ux.a
SPACE= /stand/build/space.h
SPACEDIR= /usr/conf/space.h.d
DIR=.
ORDER=/usr/conf/gen/linkorder
LOADOPTS= -PF $(ORDER) -O -P -R 20000 -N -e rdb_bootstrap -a archive
LIBK=
SPECVFLAGS= -L$(ROOT) ${LIBK}
LIBUSRDRV=`if [ -f $(ROOT)/libusrdrv.a ]; then \\
echo $(ROOT)/libusrdrv.a; fi;`

# Default target
#all: $(DIR)/$(HPUX)
all: rdisk.o

LOCORE=locore.o
DBGS= virt_mem.o wtermio.o termio3.o termio2.o termio1.o termio0.o telnet.o \ 
   strti.o streams.o security.o proc_man.o multiproc.o nfs.o fddi4_dbg.o \ 
   fddi0_header.o fddi3_header.o xport_diag.o netdiag.o lan_dbgl.o \ 
   lan_dbg0.o low_level.o pci_debug.o fcms.o fc_gsc.o \ 
   wsio_scsi.o \ 
   wsio_drivers.o graf.o cent.o lv_lvm.o vxfs.o file_sys.o sio_drivers4.o \ 

Appendix A
Example Drivers
Ram Disk Static Module Device Driver

...
Example Drivers

Ram Disk Static Module Device Driver

UTIL=/usr/conf/gen
MKFUNCNAMES=/usr/lbin/sw/bin/mkfuncnames
FUNCNAMES=/stand/build/function_names

KSYMTAB= $(SBINS)/kernsymtab
CHKMIF= $(SBINS)/kmckintfc

rdisk.o: rdisk.c $(SPACE)
@- # FORCEBUILD=${FORCEBUILD}
@echo 'Compiling rdisk.c...'
@$<RM> -f rdisk.o
@$<CC> -I. \
$(CFLAG2) -c -o rdisk.o $(CFLAGS) $(NOGLOOPTS) rdisk.c
Example Drivers
Ram Disk Static Module Device Driver

Ram Disk Static Device Driver: master file and general information

master file

* This is a master file for rdisk
$DRIVER_INSTALL
*
* Driver  Block major  Char major  Required for rdisk
-1-1
$$$

ramdiskinfo general information

root@glenpci3:/work >
root@glenpci3:/work > lsdev | grep rdisk
  130   69      rdisk   pseudo
root@glenpci3:/work >
root@glenpci3:/work > mknod /dev/dsk/ramdisk b 69 0
root@glenpci3:/work > mknod /dev/rdsk/ramdisk c 130 0
root@glenpci3:/work >
root@glenpci3:/work > ll /dev/rdsk/ramdisk
  crw-rw-rw- 1 root   sys      130 0x00000f Mar 20 08:45
  /dev/rdsk/ramdisk
root@glenpci3:/work >
root@glenpci3:/work > ll /dev/dsk/ramdisk
  brw-rw-rw- 1 root   sys       69 0x00000f Mar 20 03:44
  /dev/dsk/ramdisk
root@glenpci3:/work >
root@glenpci3:/work > newfs -Fhfs -s 128 /dev/rdsk/ramdisk
mkfs (hfs): Warning - 26 sector(s) in the last cylinder are not allocated.
mkfs (hfs): /dev/rdsk/ramdisk - 128 sectors in 1 cylinders of 7 tracks, 22 sectors
0.1Mb in 1 cyl groups (16 c/g, 2.52Mb/g, 64 i/g)
Super block backups (for fsck -b) at: 16
root@glenpci3:/work >
root@glenpci3:/work > mount /dev/dsk/ramdisk /tmp_mnt
root@glenpci3:/work >
root@glenpci3:/work > ls /tmp_mnt lost+found
root@glenpci3:/work > cp rdisk.c /tmp_mnt
root@glenpci3:/work > ls /tmp_mnt
  blk2.c    lost+found
root@glenpci3:/work >
Ram Disk WSIO DLKM Driver

The following pages show the workstation Demand Loadable Kernel Module ram disk device driver.

RamDisk DLKM Device Driver: Program Source

/*-----------------*/
/* blk2.c          */
/* RAM disk driver */
/*-----------------*/

#include "../h/types.h"
#include "../h/param.h"
#include "../h/buf.h"
#include "../h/io.h"
#include "../h/ioctl.h"
#include "../h/errno.h"
#include "../h/conf.h"
#include "../h/sysmacros.h"
#include "../h/diskio.h"
#include "../h/dir.h"
#include "../h/moddefs.h"
#include "../h/mod_conf.h"
#include "../wsio/wsio.h" /* if KERN def’d expects eeprom.h in wsio */

#define RAMDISK_SIZE 2*1024*1024

static caddr_t ramptr = NULL;
static uint32_t ram_size = 0;
static int nblks = 0;
static space_t ramsid;

int blk2_install();
void blk2_init();
int blk2_open();
void blk2_close();
int blk2_ioctl();
int blk2_read();
int blk2_write();
Example Drivers

Ram Disk WSIO DLKM Driver

void blk2_strategy();
extern int (*dev_init)();
int (*blk2_saved_dev_init)();
int blk2_dlkm = 0;

extern struct mod_operations gio_mod_ops;
static int blk2_load();
static int blk2_unload();

/* static */ struct mod_type_data blk2_drv_link = {
   "blk2", NULL
};

/* static */ struct modlink blk2_mod_link[] = {
   { &gio_mod_ops, &blk2_drv_link },
   { NULL, NULL }
};

/* static */ struct mod_conf_data mc = { MCD_VERSION,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
...
Example Drivers
Ram Disk WSIO DLKM Driver

blk2_read,
blk2_write,
blk2_ioctl,
NULL,
NULL,
NULL,
NULL,
NULL,
NULL,
0,
};

static wsio_drv_data_t blk2_data = {
"blk2_ramdisk",
T_UNKNOWN,
DRV_CONVERGED,
NULL,
NULL
};

static wsio_drv_info_t blk2_wsio_drv_info = {
&blk2_drv_info,
&blk2_drv_ops,
&blk2_data
};

/*---------------*/
/*  BLK2_INSTALL */
/*---------------*/

int blk2_install() {
blk2_saved_dev_init = dev_init;
dev_init = blk2_init;
blk2_dlkm = 0;
return (wsio_install_driver(&blk2_wsio_drv_info));
}

/*------------*/
/* BLK2_LOAD */
/*------------*/

int blk2_load(drv_info_arg)
Example Drivers

Ram Disk WSIO DLKM Driver

```c
void *drv_info_arg;
{
if (drv_info_arg == NULL) {
printf("blk2_load null ptr passed in\n");
return 1;
}
blk2_wsio_drv_info.drv_info = (struct drv_info *)drv_info_arg;
if (!wsio_install_driver(&blk2_wsio_drv_info)) {
printf("blk2_load wsio install driver failed\n");
return ENXIO;;
}
blk2_dlkm = 1;
blk2_init();
return 0;
}

/*****************************************************************************/
/* BLK2_UNLOAD */
/*****************************************************************************/
int blk2_unload(drv_info_arg)
void *drv_info_arg;
{
blk2_wsio_drv_info.drv_info = (struct drv_info *)drv_info_arg;
wsiq_uninstall_driver(&blk2_wsio_drv_info);
if (ramptr) {
sys_memfree(ramptr,(long)ram_size);
}
return 0;
}

/*****************************************************************************/
/* BLK2_INIT */
/*****************************************************************************/
void
blk2_init()
{
ram_size = RAMDISK_SIZE;
ramptr = sys_memall((long)ram_size);
if (ramptr) {
ramsid = ldsid(ramptr);
nblks = RAMDISK_SIZE >> DEV_BSHIFT;
} else {
```
Example Drivers
Ram Disk WSIO DLKM Driver

nblks = 0;
ram_size = 0;
}
if (!blk2_dlkm)
(*blk2_saved_dev_init());
}

/*-------------*/
/* BLK2_OPEN */
/*-------------*/
blk2_open(dev,flag)
dev_t dev;
int flag;
{
if (ram_size == 0)
return ENOSPC;
return 0;
}

/*-----------*/
/* BLK2_READ */
/*-----------*/
int
blk2_read(dev,uio)
dev_t dev;
struct uio *uio;
{
if ((uio->uio_offset & DEV_BMASK) &&
((uio->uio_fpflags & FBLKSEEK) == 0))
return(ENXIO);
return physio(blk2_strategy,NULL,dev,B_READ,minphys,uio);
}
int blk2_write(dev,uio)
dev_t dev;
struct uio *uio;
{
if ((uio->uio_offset & DEV_BMASK) &&
((uio->uio_fpflags & FBLKSEEK) == 0))
return(ENXIO);
return physio(blk2_strategy,NULL,dev,B_WRITE, minphys,uio);
}
/* BLK2_STRATEGY */
/*-------------------*/

void
blk2_strategy(bp)
struct buf *bp;
{
    size_t size;
    caddr_t start;
    space_t bp_sid;

    bp_sid = bvtospace(bp,bp->b_un.b_addr);
    start = ramptr + bp->b_offset;
    /* check transfer in volume and nonzero length */
    if ((start >= ramptr + ram_size) || (ram_size == 0)) {
        bp->b_resid = bp->b_bcount;
        bp->b_flags |= B_ERROR;
        bp->b_error = ENOSPC;
        iodone(bp);
        return;
    }
    /* truncate if transfer goes past end of volume */
    size = MIN((int)((uint32_t)ramptr + ram_size) - (uint32_t)start,
                bp->b_bcount);
    /* abort if not a multiple of block size */
    if (bp->b_bcount & DEV_BMASK) {
        bp->b_resid = bp->b_bcount;
        bp->b_flags |= B_ERROR;
        bp->b_error = ENXIO;
        iodone(bp);
        return;
    }
    /* do the transfer */
    if (bp->b_flags & B_READ)
        privlbcopy(ramsid,start,bp_sid,bp->b_un.b_addr,bp->b_bcount);
    else
        privlbcopy(bp_sid,bp->b_un.b_addr,ramsid,start,bp->b_bcount);
    iodone(bp);
}

/*-------------------*/
/* BLK2_IOCTL */
Example Drivers
Ram Disk WSIO DLKM Driver

/*-----------*/
int
blk2_ioctl(dev,cmd,arg,flag)
dev_t dev;
int cmd;
caddr_t arg; /* was int arg, changed to caddr_t for 64-bit */
int flag;
{
capacity_type *cap;
disk_describe_type *describe;
int size = (cmd & IOCSIZE_MASK) >> 16;

switch (cmd & IOCCMD_MASK) {
case DIOC_DESCRIBE & IOCCMD_MASK:
if (cmd != DIOC_DESCRIBE && size != 32)
    return EINVAL;
describe = (disk_describe_type *)arg;
(void) strcpy(describe->model_num,"BLK2_DISK");
describe->intf_type = UNKNOWN_INTF;
describe->maxsva = nblks - 1;
describe->lgblk.sz = DEV_BSIZE;
describe->dev_type = DISK_DEV_TYPE;
describe->flags = 0;
printf("blk2:DIOC_DESCRIBE maxsva = 0x%x,lgblk.sz = 0x%x\n", describe->maxsva,describe->lgblk.sz);
return 0;
}

switch (cmd) {
case DIOC_CAPACITY:
    cap = (capacity_type *) arg;
    cap->lba = nblks;
    printf("blk2_ioctl:DIOC_CAPACITY cap->lba = 0x%lx\n",cap->lba);
    return 0;
}
return ENXIO;
}
/*-----------*/
/* BLK2_CLOSE */
/*-----------*/

void

Appendix A
Example Drivers
Ram Disk WSIO DLKM Driver

```
blk2_close(dev)
dev_t dev;
{
    return;
}
```

**Ram Disk DLKM Device Driver: Makefile**

```
# makefile for building blk2 DLKM module
#
MODULE=blk2
CMODE=-Ae

BDEV=/dev/dsk/ramdisk
CDEV=/dev/rdsk/ramdisk

KERNEL_BITS=$$(getconf KERNEL_BITS)

all: mod.o

config:mod.o
kminstall -a $(MODULE) 2> /dev/null
cconfig -M $(MODULE) -u 2> /dev/null

bdev:config $(BDEV)
cdev:config $(CDEV)

$(BDEV):
kadmin -L $(MODULE)
kadmin -Q $(MODULE) | awk '/Block Major/ { \ 
rc=system("mknod $(BDEV) b "$3 " 0"); \ 
exit rc; \ 
}’

$(CDEV):
kadmin -Q $(MODULE) | awk '/Character Major/ { \ 
rc=system("mknod $(CDEV) c "$3 " 0"); \ 
exit rc; \ 
}’

mod.o:
make -f mod${KERNEL_BITS}.mk MODULE=$(MODULE) \ 
CMODE=$(CMODE)
```
Ram Disk DLKM Device Driver: mod32.mk

# generic 32 bit kernel module makefile
#
IDENT=-DLP64_TMP NODBG -DSPP_RUNWAY_ERR_ENABLED -DSPP_OBP_BOOT
-DROSEVILLE -DKGDB_ON -DSTCP -DHPONCPLUS -DROSE
-DNEW_MFCTL_W -DCOMB_FLIPPER -DINET COSE -DIVT_INTERCEPT
-DMULTICAST -DPAPPLETALK -DFDDI VM -DAUDIT -DACLS -DPGPROF -DNI
-DMP -DLWSYSCALL -DNEW RDB -DRDB -DLVMRD -DLVM
-DCONVERGED IO

IDENT_TMP64=

IDENT_INT=-DHIGHC -DSTDC_EXT -DKERNEL_BUILD
-D_XPG4_EXTENDED -DHPUX SOURCE -D_UNSUPPORTED -Dhp9000s800
-Dhp9000s800

IDENT_KERN=-DKERNEL

UNDEF=-Uhp9000s700

PAARCH=

DAFLAG=+DA1.1
DSFLAG=+DS2.0

RM=rm

COPTS=$(IDENT) $(IDENT_TMP64) $(OPTIONS) $(UNDEF) $(IDENT_INT)
$(IDENT_KERN)

OFLAGS=+XixdU
Example Drivers

Ram Disk WSIO DLKM Driver

CCOPTS=-Wp,-H300000 $(OFLAGS) +Hx0 +R500 -Wl,-a,archive \$(DAFLAG) $(DSFLAG) +ESsf +ESssf

CFLAGS= -w $(CCOPTS) $(COPTS)
CFLAG2= +ES1.Xindirect_calls

# Enable Assembler Warnings
AFLAGS=-ew

# Default target
all: mod.o

mod.o:$(MODULE).c
@echo 'Compiling $(MODULE).c...'
@$(RM) -f mod.o
$(CC) $(CMODE) -I. -I/usr/conf/h \$(CFLAG2) -c -o mod.o $(CFLAGS) \$(NOGLOOPTS) $(MODULE).c

Ram Disk DLKM Device Driver: mod64.mk

#
# generic 64 bit kernel module makefile
#
IDENT_TMP64=-D__TEMP64__
IDENT_INT=-D__HIGHC__ -D__STDC_EXT__ -D_KERNEL_BUILD -D_XPG4_EXTENDED -D_HPUX_SOURCE -D_UNSUPPORTED -D_hp9000s800 -Dhp9000s800
IDENT_KERN=-D_KERNEL -DKERNEL
UNDEF=-U_hp9000s700
PAARCH=64
DAFLAG=+DA2.0W
DSFLAG=+DS2.0
Example Drivers
Ram Disk WSIO DLKM Driver

RAM=rm

COPTS=${IDENT} ${IDENT_TMP64} ${OPTIONS} ${UNDEF} ${IDENT_INT} ${IDENT_KERN}
OFLAGS=+XixdU

CCOPTS=-Wp,-H300000 ${OFLAGS} +Hx0 +R500 -W1,-a,archive \
${DAFLAG} ${DSFLAG} +ESsfc +ESssf

CFLAGS= -w ${COPTS} ${CCOPTS}
CFLAG2= +ES1.Xindirect_calls

# Enable Assembler Warnings
AFLAGS=-ew

# Default target
all: mod.o

mod.o:$(MODULE).c
@echo 'Compiling $(MODULE).c...'
@$(RM) -f mod.o
$(CC) $(CMODE) -I. -I/usr/conf/h \${CFLAG2} -c -o mod.o $(CFLAGS) \
$(NOGLOOPTS) $(MODULE).c

Ram Disk DLKM Device Driver: master and system files

master file

* This is a master file for blk2
$VERSION
1
$$$

$DRIVER_INSTALL
*
* Driver     Block major     Char major    Required for
*
blk2-l-1
$$$

$LOADABLE
$$$

Appendix A  465
Example Drivers
Ram Disk WSIO DLKM Driver

$INTERFACE
base
$$$

$DRIVER_DEPENDENCY
$$$

$TYPE
blk2 wsio_class ramdisk cbp -1 -1
$$$

$TUNABLE
$$$

system file
* system file for DLKM blk2 ramdisk driver
$VERSION1
$CONFIGUREY
$LOADABLEY
* $TUNABLE

Ram Disk DLKM Device Driver: miscellaneous information

doit - script to install the blk2 module

#!/bin/sh
kminstall -d blk2
kminstall -a blk2
kmsystem -c Y -l Y blk2
config -M blk2
kmupdate -M blk2
kmadmin -L blk2
kmadmin -S
dofs - script to make a file system using the blk2 module

#!/bin/sh
newfs -Fhfs -s 128 /dev/rdsk/ramdisk
mount /dev/dsk/ramdisk /tmp_mnt
bdf
Example Drivers
Ram Disk WSIO DLKM Driver

ramdiskinfo installation information

root@glenpci3:/work >
root@glenpci3:/work > lsdev -h -d blk2
  130 69 blk2 pseudo
root@glenpci3:/work >
root@glenpci3:/work > mknod /dev/dsk/ramdisk b 69 0
root@glenpci3:/work > mknod /dev/rdsk/ramdisk c 130 0
root@glenpci3:/work >
root@glenpci3:/work > lsdev -h -d blk2
  130 69 blk2 pseudo
root@glenpci3:/work >
root@glenpci3:/work > mount /dev/dsk/ramdisk /tmp_mnt
root@glenpci3:/work >
root@glenpci3:/work > ls /tmp_mnt
lost+found
root@glenpci3:/work > cp blk2.c /tmp_mnt
root@glenpci3:/work > ls /tmp_mnt
blk2.c lost+found
root@glenpci3:/work >
Example Drivers
PCI Monolithic Hardcopy Interface Driver

PCI Monolithic Hardcopy Interface Driver

The following pages show all of the IKON PCI monolithic driver

IKON PCI Monolithic Driver: Header File

```c
#define IKON_VENDOR_ID 0x11d5
#define IKON_DEVICE_ID1 0x0115
#define IKON_DEVICE_ID2 0x0117
#define DIRY 128 /* Device and Interface Ready */
#define DRDY 64 /* Device Ready */
#define INTF 32 /* Interrupt Flag */
#define EMPT 16 /* Fifo empty Standard Fifo is 16kbytes */
#define NHLF 8 /* Fifo not half full */
#define NFLL 4 /* Fifo not full */
#define PRNT 2 /* Print mode */
#define BIT8 1 /* 8 bit data path */
#define DIRY_LE 0x80000000 /* endian swapped DIRY */
#define CBSY 8 /* Centronics Busy signal asserted */
#define PMTY 4 /* Paper Empty */
#define ONLN 2 /* Device is online */
#define CFLT 1 /* Device indicates fault */
#define CHAN1DONE (1 << 12)
#define CHAN1CLEARINT (1 << 11)
uint32_t iklp_swap(uint32_t);
uint32_t iklp_swap(uint32_t value)
{
    return (uint32_t)((value >> 24)
                      | ((value & 0x00ff0000) >> 8)
                      | ((value & 0x0000ff00) << 8)
                      | ((value & 0x000000ff) << 24));
}
static char SCCSid[]="@(#)Driver Class Printer Driver v1.0\n";
struct _ikonregs {
    uint32_t interrupt_mask; /* card base + 0x00 */
    uint32_t mode; /* card base + 0x04 */
    uint32_t device_control; /* card base + 0x08 */
    uint32_t interface_control; /* card base + 0x0c */
```
Example Drivers

PCI Monolithic Hardcopy Interface Driver

```c
uint32_t interface_status;/* card base + 0x10 */
uint32_t device_status;/* card base + 0x14 */
uint32_t reverse_data;/* card base + 0x18 */
uint32_t reserved1[9];/* card base + 0x1c */
uint8_t out_byte;/* card base + 0x40 */
uint8_t outbytereserved[3];/* card base + 0x41 */
uint32_t reserved2;/* card base + 0x44 */
uint32_t command_out;/* card base + 0x48 */
uint32_t reserved3;/* card base + 0x4c */
uint32_t out_word;/* card base + 0x50 */
uint32_t reserved4[37];/* card base + 0x54 */
uint32_t interrupt_control;/* card base + 0xe8 */
uint32_t dma0mode;/* card base + 0x100 */
uint32_t dma0pciadd;/* card base + 0x104 */
uint32_t dma0localadd;/* card base + 0x108 */
uint32_t dma0cnt;/* card base + 0x10c */
uint32_t dma0descriptor;/* card base + 0x110 */
uint32_t dma1mode;/* card base + 0x114 */
uint32_t dma1pciadd;/* card base + 0x118 */
uint32_t dma1localadd;/* card base + 0x11c */
uint32_t dma1cnt;/* card base + 0x120 */
uint32_t dma1descriptor;/* card base + 0x124 */
uint32_t dmastatus;/* card base + 0x128 */
uint32_t dma0arb;/* card base + 0x12c */
uint32_t dma1arb;/* card base + 0x130 */
}*pikonregs; /* card’s registers */

struct _ikon_chain {
  uint32_t addr;
  uint32_t cnt;
  uint32_t localaddr;
  uint32_t *next
}*pikon_chain;
```
**Example Drivers**

**PCI Monolithic Hardcopy Interface Driver**

---

**IKON PCI Monolithic Driver: Program Source**

/*-----------------------------*/
/* Driver Writing Class Printer Lab */
/* DMA transfers using PCI IKON card */
/*-----------------------------*/

#include "../wsio/wsio.h"
#include "../h/types.h"
#include "../h/errno.h"
#include "../h/uio.h"
#include "../h/io.h"
#include "../h/param.h"
#include "../machine/eisa.h"
#include "../h/buf.h"
#define PCI_LITTLE_ENDIAN_ONLY
#include "./iklp.h"
#include "../wsio/pcl.h"

int iklp_install();
int iklp_write(dev_t,struct uio*);
int iklp_open(dev_t,int);
int iklp_close(dev_t,int);
void iklp_strategy(struct buf*);
void iklp_attach(uint32_t,struct isc_table_type *);
int iklp_isr(struct isc_table_type *,int);
int iklp_start_dma();
void iklp_minphys(struct buf*);

static drv_info_t iklp_info =
{
"iklp",/* driver name */
"printer",/* driver class */
DRV_CHAR,/* type */
-1,/* block major num */
-1,/* char major num */
NULL, NULL, NULL/* always NULL */
};

static drv_ops_t iklp_ops =
{
 iklp_open,/* open */
 iklp_close,/* close */
 NULL,/* strategy */
 NULL,/* dump */
 NULL,/* psize */

Appendix A
圳,/* reserved */
圳,/* read */
icklp_write,/* write */
圳,/* ioctl */
圳,/* select */
圳,/* option1 */
圳, NULL, NULL, /* reserved */
圳,/* link */
C_ALLCLOSES|C_MAP_BUFFER_TO_KERNEL/* device flags */

static wsio_drv_data_t iklp_data =
{
"iklp_driver" ,/* driver path */
T_DEVICE ,/* driver type */
DRV_CONVERGED, /* driver flag */
NULL, /* minor build */
NULL/* minor decode */
};

static wsio_drv_info_t iklp_wsio_info =
{
&iklp_info,
&iklp_ops,
&iklp_data
};

static uint32_t *pchainmem;
static uint32_t *pburstout;
static int iklp_xcl_open =0;
struct isc_table_type *ikon_isc =NULL;
struct dma_parms iklp_dma_parms;
struct buf *iklp_bp =NULL;
int iklp_bytes =0;
int iklp_wordbytes =0;
int iklp_total_wordbytes =0;

extern int (*pci_attach)();
int (*iklp_saved_pci_attach)();

iklp_install()
{
 iklp_saved_pci_attach = pci_attach;
 //save head pointer */
pci_attach = iklp_attach; /*make my dev_init head*/
return (wsio_install_driver(&iklp_wsio_info) );

Appendix A

471
Example Drivers
PCI Monolithic Hardcopy Interface Driver

}{

iklp_attach(uint32_t idparm, struct isc_table_type *isc) {
    PCI_ID *id = (PCI_ID *)&idparm;
    uint16_t tmp16;
    uint32_t bar;

    bzero((caddr_t)&iklp_dma_parms,(size_t)sizeof(struct dma_parms));
    if (id->vendor_id!=IKON_VENDOR_ID)
        return (iklp_saved_pci_attach)(idparm,isc);
    if(id->device_id!=IKON_DEVICE_ID1 &&
      id->device_id!=IKON_DEVICE_ID2)
        return (iklp_saved_pci_attach)(idparm,isc);
    if (ikon_isc)
        return (iklp_saved_pci_attach)(idparm,isc);
    /*only 1 card */

    ikon_isc=isc;
    pci_read_cfg_uint16_isc(isc,PCI_CS_COMMAND,&tmp16);
    pci_write_cfg_uint16_isc(isc, PCI_CS_COMMAND,
                           tmp16 | PCI_CMD_MEM_SPACE | PCI_CMD_BUS_MASTER);
    pci_read_cfg_uint32_isc(isc,0x18,&bar);
    if (bar!=(uint32_t)isc->if_reg_ptr) {
        pikonregs=map_mem_to_host(isc,(caddr_t)bar,(size_t)0x2000);
        if (pikonregs==NULL) {
            printf("Unable to map cards memory space\n");
            return (iklp_saved_pci_attach)(idparm,isc);
        }
    }
    isc->buffer = sys_memall((long)NBPG);
    if (isc->buffer == NULL) {
        printf("Unable to map tmp buf space\n");
        return (iklp_saved_pci_attach)(idparm,isc);
    }
    isc->if_drv_data = (caddr_t)ldsid(isc->buffer);
    pchainmem=( (uint32_t) pikonregs) +0x140;
    pburstout=((uint32_t) pikonregs) +0x1000;
    if( isrlink(isc,iklp_isr,-1,(long)isc,0)!=0 )
        return (iklp_saved_pci_attach)(idparm,isc);
    isc_claim(isc,&iklp_wsio_info);
    printf(&SCCSid[4]);
    pikonregs->interrupt_mask= 0;
    pikonregs->interrupt_control =
    iklp_swap( (1 << 8) | (1<<11) | (1<<16) | (1<<19) );

472 Appendix A
pikonregs->dmallocadd = iklp_swap(pburstout);
return (iklp_saved_pci_attach)(idparm,isc);
}

iklp_open(dev_t dev,int flag)
{
uint32_t interface_status,device_status;

if (iklp_xcl_open)
return EMFILE;
device_status=iklp_swap(pikonregs->device_status);
if(device_status&ONLN && !(device_status&PMTY&CFLT)) {
  iklp_xcl_open++;
  return 0;
} else {
  printf("pDevice Status Register: %x\n",device_status);
  return EIO;
}
}

iklp_close(dev_t dev,int flag)
{
  iklp_xcl_open=0;
  return 0;
}

int iklp_write(dev_t dev,struct uio *uio)
{
if (!pikonregs->interface_status & DIRY_LE)
return EIO;
return physio(iklp_strategy,NULL,dev,B_WRITE,iklp_minphys,uio);
}

void iklp_strategy(struct buf *bp)
{
if (bp->b_bcount < 0) {
  bp->b_flags |= B_ERROR;
goto stratdumpit;
} else if (bp->b_bcount == 0) {
goto stratdumpit;
}
  bzero((caddr_t)&iklp_dma_parms,
  (size_t)sizeof(struct dma_parms));
  iklp_dma_parms.channel = BUS_MASTER_DMA;
if ((uint32_t)bp->b_un.b_addr & 0x3) {
    privlbcopy(bp->b_spaddr, bp->b_un.b_addr, (space_t)ikon_isc->if_drv_data,
               ikon_isc->if_drv_data, (space_t)ikon_isc->if_drv_data, ikon_isc->buffer, (size_t)bp->b_bcount);
    iklp_dma_parms.spaddr = (space_t)ikon_isc->if_drv_data;
    iklp_dma_parms.addr = ikon_isc->buffer;
    iklp_dma_parms.count = bp->b_bcount;
} else {
    iklp_dma_parms.spaddr = bp->b_spaddr;
    iklp_dma_parms.addr = bp->b_un.b_addr;
    iklp_dma_parms.count = bp->b_bcount;
}

iklp_dma_parms.dma_options = ((bp->b_flags & B_READ)
? (DMA_8BYTE|DMA_READ) : (DMA_8BYTE|DMA_WRITE));
if (dma_setup(ikon_isc, &iklp_dma_parms)) {
    bp->b_flags |= B_ERROR;
    goto stratdumpit;
}

iklp_dma_parms.chain_index=0;
iklp_bp=bp;
iklp_bp->b_resid=iklp_bp->b_bcount;
iklp_start_dma();
return;

stratdumpit:
bp->b_resid = bp->b_bcount;
iodone(bp);
return;

}

iklp_isr(struct isc_table_type *isc,int arg2)
{
    int i;

    if (isc!=ikon_isc)
        return 0; /* not my isc */
    if(iklp_bp==NULL)
        return 1; /* no outstanding bp’s to service */
    if( !(iklp_swap(pikonregs->dmastatus)) & CHAN1DONE )
        return 1; /* dma not done, keep going */
    pikonregs->dmastatus=iklp_swap(CHAN1CLEARINT);
    /* Clear the int */
    if(iklp_bytes>0) {
        /*send the rest of the unaligned bytes*/
        for(i=0;i<iklp_bytes;i++) {
            /*...*/
        }
    }
Example Drivers

PCI Monolithic Hardcopy Interface Driver

```c
pikonregs->out_byte = *(uint8_t *)
   (iklp_bp->b_un.b_addr+i+iklp_total_wordbytes);
}
}
iklp_bp->b_resid-=iklp_bytes;
}

if (iklp_bp->b_resid==0)  {
dma_cleanup(ikon_isc,&iklp_dmaparms);
iodone(iklp_bp);
iklp_bp=NULL;
} else
iklp_start_dma();
return 1;
}
int iklp_start_dma()
{
struct iovec *chain_ptr;
/* LP64 issue: can’t use addr_chain_type */

chain_ptr=iklp_dmaparms.chain_ptr;
chain_ptr=chain_ptr+iklp_dmaparms.chain_index;
iklp_bytes = (int)chain_ptr->iov_len % 4;
iklp_wordbytes = (int)chain_ptr->iov_len - iklp_bytes;
if(iklp_dmaparms.chain_index==0)
iklp_total_wordbytes=iklp_wordbytes;
else
iklp_total_wordbytes+=iklp_wordbytes;
iklp_dmaparms.chain_index++;
iklp_bp->b_resid-=iklp_wordbytes;
pikonregs->dmalmode=iklp_swap(0x1d07);
pikonregs->dmalpciadd=iklp_swap( (uint32_t)
   chain_ptr->iov_base);
pikonregs->dmalcnt=iklp_swap(iklp_wordbytes);
if (iklp_swap(pikonregs->dmastatus) & CHAN1DONE )
pikonregs->dmastatus=iklp_swap( 1<<8 | 1<<9);
else {
   /*interrupt won’t work so abort */
dma_cleanup(ikon_isc,&iklp_dmaparms);
    iklp_bp->b_flags|=B_ERROR;
    iodone(iklp_bp);
    iklp_bp=NULL;
}
}

void iklp_minphys(struct buf *bp)
{

Appendix A

Example Drivers
PCI Monolithic Hardcopy Interface Driver

```c
if (bp->b_bcount > NBPG)
bp->b_bcount = NBPG;
}
```

IKON PCI Monolithic Driver: Master File

```bash
$DRIVER_INSTALL
***********************************************************
*************
* Driver install table
*
* This table contains the name of drivers which have
* converged I/O header structures and install entry
* points. Drivers in this table should not
* be defined in the driver table above.
***********************************************************
*************
* Driver Block major Char major Required for
iklp               -1              -1

$$$
```

IKON PCI Monolithic Driver: 32-bit Makefile

```bash
ROOT = /usr/conf/lib
LIBS = \
$(ROOT)/libuiipc.a \n$(ROOT)/libbufs.a \n$(ROOT)/libnfs.a \n$(ROOT)/librpc.a \n$(ROOT)/libcdfs.a \n$(ROOT)/libpci.a \n$(ROOT)/libhp-ux.a \n$(ROOT)/libd1km.a \n$(ROOT)/liblan.a \n$(ROOT)/libinet.a \n$(ROOT)/libtun.a \n$(ROOT)/libtelnet.a \n$(ROOT)/libnms.a \n$(ROOT)/libfgsc.a \n$(ROOT)/libvxfs_base.a \n$(ROOT)/liblvm.a \n$(ROOT)/libstream.a \n$(ROOT)/libstrtio.a \n$(ROOT)/libautofs.a 
```

---

Appendix A
Example Drivers
PCI Monolithic Hardcopy Interface Driver

$(ROOT)/libbtlan3.a \
$(ROOT)/btlan3_dbg.o \
$(ROOT)/libmaclan.a

#CAUTION: Do not modify below.
SMODWORK = mod_wk.d
SMODSTAT = /usr/conf/km.d
SYM TAB = dlkm.vmunix_test/symtab
MMASTER = /usr/conf/master.d
KM ODS =
KM O DSTUBS =

CONF=conf
TUNE=tune
IDENT=-DBTLAN3 -DGKDB_ON -DSTCP -DHPONCPLUS -D__ROSE__
-DNEW_MFC TL_W -DCOMB_FLIPPER -DINET_COSE -DIVT_INTERCEPT
-DMULTICAST -DAPPLETALK -DFDDI_VM -DAUDIT -DACL S -DPGP PROF -DKI
-DM -DLWSYS CALL -DNEW_RDB -DRDB -DLVMROOT -DLVM
-DCONVERGED_IO
IDENT_TMP64=
ROOT=/usr/conf/lib
IDENT_INT=-D__HIGHC__ -D__STDC_EXT__ -D_KERNEL_BUILD
-D_XPG4_EXTENDED -D_HPUX_SOURCE -D_UNSUPPORTED -D_hp9000s800
-Dhp9000s800
IDENT_KERN=-D_KERNEL -DKERNEL
UNDEF=-U__hp9000s700
DAFLAG=+DA1.1
DSFLAG=+DS2.0
SPACED=space
MASTERD=master
MAPFILE=
COPTS=${IDENT} ${IDENT_TMP64} ${OPTIONS} ${UNDEF} ${IDENT_INT}
${IDENT_KERN}
OFLAGS=+XixdU

K_CC OPTS=-Wp,-H300000 $(OFLAGS) +Hx0 +R500 -Wl,-a,archive \\
-$(DAFLAG) $(DSFLAG) +ESsfc +ESsfs
K_CC OPTS_NOOPT=-Wp,-H300000 $(OFLAGS_NOOPT) +Hx0 +R500
-Wl,-a,archive \\
-$(DAFLAG) $(DSFLAG) +ESsfc +ESsfs

CFLAGS= -w $(COPTS) $(K_CC OPTS)
CFLAG2= +ES1.Xindirect_calls
HPUX=vmunix_test
LIBHPUX= $(ROOT)/libhp-ux.a
SPACE= /stand/build/space.h
Example Drivers
PCI Monolithic Hardcopy Interface Driver

SPACEDIR= /usr/conf/space.h.d
DIR=.
ORDER=/usr/conf/gen/linkorder
LOADOPTS= -PF $(ORDER) -O -P -R 20000 -N -e rdb_bootstrap -a archive
LIBK=
SPECVFLAGS= -L$(ROOT) $(LIBK)
LIBUSRDRV=`if [ -f $(ROOT)/libusrdrv.a ]; then 
  echo $(ROOT)/libusrdrv.a; fi;`

# Default target

#all: $(DIR)/$(HPUX)
all: iklp.o

LOCORE=locore.o
DBGS= virt_mem.o wtermio.o termio3.o termio2.o termio1.o
    termio0.o telnet.o 
    strpio.o streams.o security.o proc_man.o multiproc.o nfs.o
    fddi4_dbg.o 
    fddi0_header.o fddi3_header.o xport_diag.o netdiag.o
lan_dbg1.o 
    lan_dbg0.o low_level.o pci_debug.o fcms.o fc_gsc.o
wsio_scsi.o 
    wsio_drivers.o graf.o cent.o lv_lvm.o vxfs.o file_sys.o
sio_drivers4.o 
    sio_drivers3.o sio_drivers2.o sio_drivers1.o io.o
OFILES=
SZLIMIT= -l 29308928

$(SPACE): $(SPACEDIR)
@- # FORCEBUILD=$(FORCEBUILD)
-@$(RM) -f @$
$(MKDIR) -p @$
@echo "/*\n * space.h\n * Copyright 1994 by Hewlett Packard" > @$
@echo " */" >> @$
@for file in `$(LS) $(SPACEDIR) | $(GREP) -E "^\."[^-]*$\n```
IKON PCI Monolithic Driver: 64-bit Makefile

IDENT=--DLP64_TMP_NODBG --DSPP_RUNWAY_ERR_ENABLED --DSPP_OBP_BOOT --D__ROSEVILLE__ --DGDB_ON --DSTCP --DHPONCPLUS --D__ROSE__
Example Drivers

PCI Monolithic Hardcopy Interface Driver

-DNEW_MCTL_W -DCOMB_FLIPPER -DINET_COSE -DIVT_INTERCEPT
-DMULTICAST -DAPPLETALK -DFDDI_VM -DAUDIT -DACLS -DPGPROF -DKI
-DMP -DLWSYSCALL -DNEW_RDB -DRDB -D_LVMROOT -D_LVM
-DCONVERGED_IO
IDENT_TMP64=-D___TEMP64__
ROOT=/usr/conf/lib
IDENT_INT=-D__HIGHC__ -D__STDC_EXT__ -D_KERNEL_BUILD
-D_XPG4_EXTENDED -D_HPUX_SOURCE -D_UNSUPPORTED -D__hp9000s800
-Dhp9000s800
IDENT_KERN=-D_KERNEL -DKERNEL
UNDEF=-U__hp9000s700
DAFLAG=+DA2.0W
DSFLAG=+DS2.0
COPTS=${IDENT} ${IDENT_TMP64} ${OPTIONS} ${UNDEF} ${IDENT_INT}
${IDENT_KERN}
OFLAGS=+XixdU
K_CCOPTS=-Wp,-H300000 $(OFLAGS) +Hx0 +R500 -Wl,-a,archive 
   ${DAFLAG} ${DSFLAG} +ESsfc +ESssf
K_CCOPTS_NOOPT=-Wp,-H300000 $(OFLAGS_NOOPT) +Hx0 +R500 
   -Wl,-a,archive 
   ${DAFLAG} ${DSFLAG} +ESsfc +ESssf
CFLAGS= -w ${COPTS} ${K_CCOPTS}
CFLAG2= +ES1.Xindirect_calls
LOADOPTS= -PF ${ORDER} -P -noshared -k ${MAPFILE} +nodefaultmap 
   +Accept 
   TypeMismatch -e rdb_bootstrap -a archive

# Default target
#all: ${DIR}/${HPUX}
all: iklp.o

# Set these to alternate path if we’re running pre-10.0
BIN=/usr/bin
CKRN=/usr/ccs/bin
SBIN=/usr/sbin
SBINS=/usr/sbin
AWK=$(BIN)/awk
SHELL=/sbin/sh
SH=/sbin/sh
RM=$(BIN)/rm
CHMOD=$(BIN)/chmod
MKDIR=$(BIN)/mkdir
GREP=$(BIN)/grep
LS=$(BIN)/ls
AR=TMPDIR=/tmp $(CKRN)/ar
#CC=$(CKRN)/cc
CC=/bin/cc
LD=$(CKRN)/ld
SZ=$(CKRN)/size

iklp.o: iklp.c
@echo 'Compiling iklp.c...'
@${RM} -f iklp.o
@${CC} -Ae -I. \n${CFLAG2} -c -o iklp.o ${CFLAGS} ${NOGLOOPTS} iklp.c
PCI Combined User and Kernel Driver

The following pages show all of the IKON PCI combined kernel and user driver.

PCI Combined Driver: Header File

```c
/*----------*/
/* ikuser.h */
/*------------------*/
/* public defines */
/* for user mapped */
/* IKON regs driver */
/*------------------*/
struct ikonregs {
    uint32_t interrupt_mask;
    uint32_t mode;
    uint32_t device_control;
    uint32_t interface_control;
    uint32_t interface_status;
    uint32_t device_status;
    uint32_t reverse_data;
    uint32_t reserved0;
    uint32_t reserved1[8];
    uint32_t out_byte;
    uint32_t reserved2;
    uint32_t command_out;
    uint32_t out_word;
};

#define IKUSER_REGS _IOR('z',3,long)
```

PCI Combined Driver: Kernel Program Source

```c
/*----------*/
/* ikuser.c */
/*-------------------------*/
/* iomap access driver for */
/* IKON PCI interface */
/*-------------------------*/

#include "../h/types.h"
#include "../h/param.h"
```
Example Drivers
PCI Combined User and Kernel Driver

#include "../h/errno.h"
#include "../h/io.h"
#include "../h/ioctl.h"
#include "../h/mman.h"
#include "../machine/pde.h"
#include "../wsio/wsio.h"
#include "../wsio/pci.h"
#include "./ikuser.h"

#define IKON_VENDOR_ID 0x11d5
#define IKON_DEVICE_ID1 0x0115
#define IKON_DEVICE_ID2 0x0117
#define IKON_PCI_MEM_BASE 0x10
#define IKON_PCI_IO_BASE 0x14
#define IKON_REGS_BASE 0x18

#define debug 0

#ifdef debug
#define static
#endif

int (*ikuser_saved_attach)();
int ikuser_install();
int ikuser_attach();
int ikuser_open();
int ikuser_close();
int ikuser_ioctl();

extern int (*pci_attach)();

static drv_ops_t ikuser_ops = {
  ikuser_open,
  ikuser_close,
  NULL,
  NULL,
  NULL,
  NULL,
  NULL,
  NULL,
  ikuser_ioctl,
  NULL,
  NULL,
  NULL,
  NULL,
  NULL,
  NULL,
};
Example Drivers

PCI Combined User and Kernel Driver

```
    static drv_info_t ikuser_info = {
            "ikuser", /* driver name */
            "printer", /* driver class */
            DRV_CHAR | DRV_SAVE_CONF | DRV_MP_SAFE, /* type */
            -1, /* block major num */
            -1, /* char major num */
            0, 0, 0, /* always NULL */
    };

    static wsio_drv_data_t ikuser_data = {
            "ikonhc_intfc", /* for probes */
            T_INTERFACE, /* driver type */
            DRV_CONVERGED, /* driver flags */
            0, 0, /* minor_build, minor_decode */
    };

    static wsio_drv_info_t ikuser_wsio_drv_info = {
            &ikuser_info,
            &ikuser_ops,
            &ikuser_data
    };

    static caddr_t ikphys_addr = NULL;
    static pgcnt_t ikphys_size = 0;
    static int ikuser_attached = 0;
    static int ikuser_opened = 0;
    static caddr_t ikuser_mapped = NULL;

    /*----------------------*/
    /*    IKUSER_INSTALL    */
    /*----------------------*/

    ikuser_install() {
        int ret = wsio_install_driver(&ikuser_wsio_drv_info);
        if (ret) {
            ikuser_saved_attach = pci_attach;
            pci_attach = ikuser_attach;
        }
        return ret;
    }

    /*-------------------*/
```
/*   IKUSER_ATTACH   */
/*-------------------*/

ikuser_attach(idparm,isc)
uint32_t idparm;
struct isc_table_type *isc;
{
  PCI_ID *id = (PCI_ID *)&idparm;
  uint32_t saveaddr,regsize;
  ubit16 cmdreg;
  uint8_t rev_id;
  caddr_t mapregs,physregs;
  pgcnt_t regsizep;

  if (!(id->vendor_id == IKON_VENDOR_ID))
    return ikuser_saved_attach(idparm,isc);
  if ((id->device_id != IKON_DEVICE_ID1) &&
    (id->device_id != IKON_DEVICE_ID2)) {
    return ikuser_saved_attach(idparm,isc);
  }
  /* get the rev ID of the chip */
  pci_read_cfg_uint8_isc(isc,PCI_CS_REV_ID,&rev_id);
  /* check for ikon regs access */
  pci_read_cfg_uint32_isc(isc,IKON_REGS_BASE,&saveaddr);
  /* services map first memory BAR < 8K and pass it in */
  if_reg_ptr *
  if ((uint32_t)saveaddr == (uint32_t)isc->if_reg_ptr) {
    msg_printf("IKUSER Attach failed: regs previously mapped\n");
    return ikuser_saved_attach(idparm,isc);
  }
  /* we want the regs BAR which is the second memory BAR */
  /* need to use the PCI spec method of determining size */
  /* requested */
  pci_write_cfg_uint32_isc(isc,IKON_REGS_BASE,0xffffffff);
  pci_read_cfg_uint32_isc(isc,IKON_REGS_BASE,&regsize);
  if (regsize < sizeof(struct ikonregs)) {
    msg_printf("IKUSER Attach failed: invalid reg size value\n");
    return ikuser_saved_attach(idparm,isc);
  }
  pci_read_cfg_uint16_isc(isc, PCI_CS_COMMAND, &cmdreg);
  pci_write_cfg_uint16_isc(isc, PCI_CS_COMMAND, cmdreg | PCI_CMD_MEM_SPACE);
  /* iomap public calls require the map size in pages */
  regsizep = regsize >> PGSHIFT;
}
Example Drivers
PCI Combined User and Kernel Driver

mapregs = (caddr_t)saveaddr;
#ifdef __LP64__
mapregs = EXTEND_ADDR(mapregs);
#endif /* __LP64__ */
/* map the regs user access prot and since we are using
  * the kernel_iomap_public call we also have kernel
  * access */
physregs = kernel_iomap_public(NULL,mapregs,
regsizep,(prot_t)PROT_URW);
if (physregs == NULL) {
  msg_printf("IKUSER_ATTACH: unable to public map regs
  ");
  return ikuser_saved_attach(idparm,isc);
}

ikphys_addr = physregs;
ikphys_size = regsizep;
ikuser_attached = 1;
is_claim(isc,&ikuser_wsio_drv_info);
#ifdef debug
msg_printf("IKUSER_ATTACH: public mapped %d pages for regs @
0x%lx\n",
ikphys_size,ikphys_addr);
#endif
return ikuser_saved Attach(idparm,isc);

 这是一个示例驱动程序的代码片段，展示了如何在用户和内核驱动之间映射寄存器。代码中使用了`kernel_iomap_public`函数来实现映射，并且确保了寄存器的用户访问权限和内核访问权限。
/*   IKUSER_CLOSE   */
/*-----------------*/

ikuser_close(dev)
dev_t dev;
{
    int ret;
    caddr_t mapregs = ikphys_addr;

    #if debug
    printf("IKUSER_CLOSE: open flag %d mapped 0x%lx\n", ikuser_opened,ikuser_mapped);
    #endif
    if (ikuser_mapped) {
        ret=user_iounmap(mapregs,ikphys_size);
        if (ret == 0)
            ikuser_mapped = NULL;
    }
    ikuser_opened = 0;
    return 0;
}

/*--------------*/
/* IKUSER_IOCTL */
/*--------------*/

ikuser_ioctl(dev,cmd,data,flags)
dev_t dev;
int cmd;
caddr_t data;
int flags;
{
    #ifdef __LP64__
    #define _IKUSER_REGS_32   _IOR('z',3, int)
    #endif /* __LP64__ */
    int *addr32 = (int *)data;

    switch (cmd) {
        case IKUSER_REGS:
            if (ikuser_mapped == NULL)
                ikuser_mapped=user_iomap(NULL,mapregs,ikphys_size);
            if (ikuser_mapped)
                ikuser_mapped=user_ioumap(NULL,mapregs,ikphys_size);
            if (ikuser_mapped) {
Example Drivers
PCI Combined User and Kernel Driver

```c
*addr = (long)ikuser_mapped;
#ifdef debug
printf("IKUSER_IOCTL: IKUSER_REGS, addr 0x%lx\n",
ikuser_mapped);
#else
*addr = 0x0;
ret = ENOMEM;
#else
*addr32 = (int)ikuser_mapped;
#endif
#else
default:
#endif /* __LP64__ */
default:
#else
printf("\tcritical default\n");
#endif
ret = ENXIO;
}
return ret;
```
Example Drivers
PCI Combined User and Kernel Driver

```c
#include <sys/types.h>
#include <sys/errno.h>
#include "../machine/pde.h"
#include "/ikuser.h"

#define DIRY_LE 0x80000000

void sendbyte();

main()
{
    int c, fd;
    long cardloc;
    struct ikonregs *cp;

    fd = open("/dev/ikuser", 2);
    if (fd < 0) {
        perror("/dev/ikuser open failed\n");
        exit(0);
    }
    if (ioctl(fd, IKUSER_REGS, &cardloc) < 0) {
        perror("ioctl call failed");
        close(fd);
        exit(0);
    }
    cp = (struct ikonregs *) cardloc;
    printf("card addr = 0x%lx\n", cp);
    fflush(stdout);
#if 1
    while ((c = getchar()) != EOF) {
        sendbyte(cp, (unsigned char)c);
        if (c == \'\n\') sendbyte(cp, (unsigned char)\'\r\');
    }
#endif
    close(fd);
}

void sendbyte(ikregs, src)
{
    struct ikonregs *ikregs;
    unsigned char src;
    {
        unsigned char *dest;
        uint32_t *stat, foo;
        int i = 0;
        dest = (unsigned char *)(&ikregs->out_byte);
        stat = (uint32_t *) &ikregs->interface_status;
        *dest = src;
        while ((*stat & DIRY_LE) == 0);
    }
}
```

Appendix A 489
Example Drivers
PCI Combined User and Kernel Driver

PCI Combined Driver: 32-bit Makefile

```
ROOT = /usr/conf/lib
LIBS = \\
$(ROOT)/libuiipc.a \\
$(ROOT)/libufs.a \\
$(ROOT)/libnfs.a \\
$(ROOT)/librpc.a \\
$(ROOT)/libcdfs.a \\
$(ROOT)/libpci.a \\
$(ROOT)/libhp-ux.a \\
$(ROOT)/libdkkm.a \\
$(ROOT)/liblan.a \\
$(ROOT)/libinet.a \\
$(ROOT)/libnet.a \\
$(ROOT)/libtun.a \\
$(ROOT)/libtelnet.a \\
$(ROOT)/libnmms.a \\
$(ROOT)/libfcgsc.a \\
$(ROOT)/libvxfs_base.a \\
$(ROOT)/liblvmlm.a \\
$(ROOT)/libstream.a \\
$(ROOT)/libstrrio.a \\
$(ROOT)/libautofs.a \\
$(ROOT)/libbtlan3.a \\
$(ROOT)/btlan3_dbg.o \\
$(ROOT)/libmaclan.a

#CAUTION: Do not modify below.
SMODWORK = mod_wk.d
SMODSTAT = /usr/conf/km.d
SYMTAB = dlkm.vmunix_test/symtab
MMASTER = /usr/conf/master.d
KMODS =
KMODSTUBS =

CONF=conf
TUNE=tune
IDENT=-DBTLAN3 -DKGDB_ON -DSTCP -DHPCONPLUS -D__ROSE__
-DNEW_MFCRTL_W -DCOMB_FLIPPER -DINET_COSE -DIVT_INTERCEPT
-DMULTICAST -DAPPLETALK -DFDDI_VM -DAUDIT -DACLS -DPGPROF -DKI
-DMP -DLWSYSCALL -DNEW_RDB -DRDB -D_LVMROOT -D_LVM
-DCONVERGED_IO
IDENT_TMP64=
ROOT=/usr/conf/lib
IDENT_INT=-D__HIGHC__ -D__STDC_EXT__ -D_KERNEL_BUILD
```
Example Drivers

PCI Combined User and Kernel Driver

-D_XPG4_EXTENDED -D_HPUX_SOURCE -D_UNSUPPORTED -D_hp9000s800
-Dhp9000s800
IDENT_KERN=-D_KERNEL -DKERNEL
UNDEF=-U__hp9000s700
DAFLAG=+DA1.1
DSFLAG=+DS2.0
SPACED=space
MASTERD=master
MAIPFILE=
COPTS=${IDENT} ${IDENT_TMP64} ${OPTIONS} ${UNDEF} ${IDENT_INT}
${IDENT_KERN}
OFLAGS=+XixdU
K_CCOPTS=-Wp,-H300000 $(OFLAGS) +Hx0 +R500 -Wl,-a,archive \ 
    ${DAFLAG} ${DSFLAG} +ESsfc +ESssf
K_CCOPTS_NOOPT=-Wp,-H300000 $(OFLAGS_NOOPT) +Hx0 +R500 
    -Wl,-a,archive \ 
    ${DAFLAG} ${DSFLAG} +ESsfc +ESssf
CFLAGS= -w ${COPTS} ${K_CCOPTS}
CFLAG2= +ES1.Xindirect_calls
HPUX=vmunix_test
LIBHPUX= $(ROOT)/libhp-ux.a
SPACE= /stand/build/space.h
SPACEDIR= /usr/conf/space.h.d
DIR=.
ORDER=/usr/conf/gen/linkorder
LOADOPTS= -PF ${ORDER} -O -P -R 20000 -N -e rdb_bootstrap -a
archive
LIBK=
SPECVFLAGS= -L$(ROOT) ${LIBK}
LIBUSRDRV=`if [ -f $(ROOT)/libusrdrv.a ]; then 
    echo $(ROOT)/libusrdrv.a; fi;`

# Default target

#all: ${DIR}/${HPUX}
all:ikuser.o
LOCORE=locore.o
DBGS= virt_mem.o wtermio.o termio3.o termio2.o termio1.o
    termio0.o telnet.o \ 
    strcio.o streams.o security.o proc_man.o multiproc.o nfs.o
    fddi4_dbg.o \ 
    fddi0_header.o fddi3_header.o xport_diag.o netdiag.o
    lan_dbg1.o \ 

Appendix A 491
Example Drivers
PCI Combined User and Kernel Driver

```
lan_dbg0.o low_level.o pci_debug.o fcms.o fc_gsc.o
wsio_scsi.o  
wsio_drivers.o graf.o cent.o lv_lvm.o vxfs.o file_sys.o
sio_drivers4.o 
 sio_drivers3.o sio_drivers2.o sio_drivers1.o io.o

OFILES=
SZLIMIT= -l 29308928

$(SPACE): $(SPACEDIR)
@- # FORCEBUILD=${FORCEBUILD}
-@$(RM) -f @
$(MKDIR) -p @$@
@echo "\"/\"n * space.h\"n * Copyright 1994 by Hewlett Packard" > @
@echo " */" >> @
@for file in `$(LS) $(SPACEDIR) | $(GREP) -E '^[[^.~#]+.h$$'`;  
do 
  echo "#include "$(SPACEDIR)$/$$file"" >> @ ; 
  done
```

## HP-UX Kernel Makefile
##
# @(#) config.sys $Date: 1998/06/19 13:53:41 $Revision: r11ros/3 PATCH_11.00 (PHKL_15687)

# Set these to alternate path if we’re running pre-10.0
BIN=/usr/bin
CKRN=/usr/ccs/bin
SBIN=/usr/sbin
SBINS=/usr/sbin

AWK=$(BIN)/awk
SHELL=/sbin/sh
SH=/sbin/sh
RM=$(BIN)/rm
CHMOD=$(BIN)/chmod
MKDIR=$(BIN)/mkdir
GREP=$(BIN)/grep
LS=$(BIN)/ls
AR=TMPDIR=/tmp $(CKRN)/ar
CC=$(CKRN)/cc
LD=$(CKRN)/ld
SZ=$(CKRN)/size
UTIL=/usr/conf/gen
MKFUNCNAMES=/usr/lbin/sw/bin/mkfuncnames
FUNCNAMES=/stand/build/function_names

KSYMTAB= $(SBINS)/kernsymtab
CHKMIF= $(SBINS)/kmckintfc

# An ordered list of all of the files used to link the # kernel. Used in several places below.
LDFILES=$(LOCORE) $(CONF).o ${XOBS} $(DBGS) $(OFILES) $(KMODS) $(KMODSTUBS) $(LIBUSRDRV) $(LIBS) $(LIBS) $(SPECVFLAGS) $(FUNCNAMES).o

# Debug kernels (-DOSDEBUG) produce symbolic stack # traces as a debugging # aid. Non-debug kernels produce purely numeric stack # traces. They’re not as convenient, but the memory # overhead (~300KB) of storing the # names of every kernel function is too high for the # production kernel.
# To link the function names and addresses into the # kernel we need a list of them. nm(1) can provide it, # but to run nm we need the hp-ux # file. (We can’t use the libraries because not # everything in them ends # up in every kernel.) To get around this circular # dependency we first link the kernel with a set of dummy # name and address tables. Then we # nm the hp-ux file we just built producing a new set of # tables containing the real data and relink the kernel # including the real data # this time. (This data lives in $(FUNCNAMES).[co].)
#
# NOTE: The space overhead of the function name and # address tables, as well as the link-time overhead, is # only paid by debug kernels.

${DIR}/${HPUX}: $(CONF).o $(LOCORE) $(DBGS) $(KMODS) $(KMODSTUBS) $(LIBS) $(MAPFILE)
$- # FORCEBUILD=$(FORCEBUILD)
$@ $(RM) -f $@ $(SYMTAB) $(FUNCNAMES).[co]
@echo `Loading the kernel...’
Example Drivers
PCI Combined User and Kernel Driver

```bash
@{
  echo '/* pro forma nameloader data structures */';
  echo '/* (see h/nameloader.h and sys/nameloader.c) */';
  echo 'void *builtin_function_addresses[] = { 0 };';
  echo 'void **function_addresses = &builtin_function_addresses;';
  echo 'char *builtin_function_names[] = { 0 };';
  echo 'void **function_names = &builtin_function_names;';
  echo 'int builtin_function_count = 0;';
  echo 'int function_count = 0;';
  echo 'int functions_compiled_in = 0;';
  echo 'char linkstamp[] = "$(LC_ALL=C;date)";';
} > ${FUNCNAMES}.c
@${CC} -I. ${CFLAG2} -c -o ${FUNCNAMES}.o ${CFLAGS} ${NOGLOOPTS} ${FUNCNAMES}.c
@$(SH) -c '${LD} -o ${DIR}/${HPUX} ${LOADOPTS} ${LDFILES} ;
  echo LdReT $$?' 2>&1 | 
  $(AWK) '/LdReT/ { exit $$2 } 
  !/(1594)|Can not optimize ADDILs/ { print }'
@case '${CFLAGS}' in
  *-DOSDEBUG*)
    if [ -x ${MKFUNCNAMES} ];
      then
        ${MKFUNCNAMES} ${DIR}/${HPUX} > ${FUNCNAMES}.c;
        ${CC} -I. ${CFLAG2} -c -o ${FUNCNAMES}.o ${CFLAGS} ${NOGLOOPTS} ${FUNCNAMES}.c;
        ${LD} -o $@ ${LOADOPTS} ${LDFILES};
      fi;
  ;;
  esac
@echo "Generating kernel symbol table..."
@$KSYMTAB} ${DIR}/${HPUX} ${SYMTAB}
@$SH} $(UTIL)/kernsize -a "$(SZ) ${DIR}/${HPUX}" $(SZLIMIT)
@$CHMOD} 755 ${DIR}/${HPUX}
@$RM} -f $(LOCORE} $(DBG) $(KMODS)

$(CONF).o: $(CONF).c $(SPACE)
  # FORCEBUILD=$(FORCEBUILD)
@echo 'Compiling $(CONF).c...'
@$RM} -f $(CONF).o
@$CC} -I. 
  ${CFLAG2} -c -o $(CONF).o $(CFLAGS} $(NOGLOOPTS} $(CONF).c

$(LOCORE) $(DBG):
  # FORCEBUILD=$(FORCEBUILD)
```
Example Drivers
PCI Combined User and Kernel Driver

@g{RM} -f $(LOCORE) ${DBGS}
g{AR} x ${LIBHPUX} $(LOCORE) ${DBGS}

ikuser.o: ikuser.c $(SPACE)
g- # FORCEBUILD=${FORCEBUILD}
g{echo} 'Compiling ikuser.c...'
g{RM} -f ikuser.o
g{CC} -I. \
g{CFLAG2} -c -o ikuser.o ${CFLAGS} ${NOGLOOPTS} ikuser.c

PCI Combined Driver: 64-bit errata
and both 32-bit and 64-bit usage warning

The 32-bit code will compile and run on a 64-bit system, returning an apparently correct address. Released 11.0 kernels, however, will get a segmentation fault core dump when access is attempted from either a 32-bit or 64-bit application.

WARNING

The kernel trap handler will catch alignment problems when a user program attempts to access I/O space using this combined driver. This type of programming error will result in a relatively benign core dump out of the user program. However, if the I/O hardware does not respond to the access by a user program to an address that was mapped (e.g., an unimplemented register), or does not respond to the type of access (e.g., a 16-bit access of a card that only accepts 32-bit accesses) an hpmc will most likely occur resulting in a system crash.

Appendix A 495
Example Drivers
PCI Combined User and Kernel Driver
Appendix B

B Data Structures, Defines, Routines, Flags, and Code Examples
This appendix contains reference information about structures, device types, packet types, and values used in SAP, TYPE, MTU, kind, and packet type fields used in networking calls described in Chapter 7, “Creating Networking Device Drivers.”

It contains reference information for the following sections:

• Interface Group MIB Objects - Contains MIB II structure definitions used by network management applications.

• MAC Types and Protocol Types - Contains a list of devices used by the hw_ift structure to indicate the device media type.

• Message Types for DLPI Primitive and Acknowledgement - Contains definitions of requests and acknowledgments for HP-UX DLPI device dependent primitives.

• MIB Event and Event Record - Contains the definition of the event message when the driver is required to send an event as part of a call to network management code.

• MTU Values - Contains a list of the defined MTU values for current HP provided protocols.

• Packet Headers - Contains a list of the packet header structures for LAN media used by HP-UX.

• Packet Types - Contains a list of the packet type values for inbound and outbound packets.

• Protocol Kinds and Values - Contains a list of the defined values for the dl_proto_kind field of a HP-UX DLPI bind request.

• SAP Values for IEEE 802.2 LLC Packets - Contains a list of the SAP values for IEEE 802.2 LLC packets.

• TYPE Values for Ethernet and SNAP protocols - Contains a list of TYPE values for Ethernet and SNAP packets.
Interface Group MIB Objects

This section contains MIB II structure definitions used by network management applications. These structures are described in the subsection “Network Management Support” on page 240, Chapter 8, and are defined in the `<h/mib.h>` header file.

**Standard MIB II**

```c
typedef struct {
    int ifIndex;
    char ifDescr[64];
    int ifType;
    int ifMtu;
    gauge ifSpeed;
    unsigned char ifPhysAddress[8];
    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;
```

**Extended MIB II for 802.3 and 802.5**

```c
/* Types for 802.3 extended MIB */

typedef struct {
    int dot3StatsIndex;
    counter dot3StatsAlignmentErrors;
} dot3_statsEntry;
```
Data Structures, Defines, Routines, Flags, and Code Examples

Interface Group MIB Objects

counter dot3StatsFCSErrors;
counter dot3StatsSingleCollisionFrames;
counter dot3StatsMultipleCollisionFrames;
counter dot3StatsSQETestErrors;
counter dot3StatsDeferredTransmissions;
counter dot3StatsLateCollisions;
counter dot3StatsExcessiveCollisions;
counter dot3StatsInternalMacTransmitErrors;
counter dot3StatsCarrierSenseErrors;
counter dot3StatsFrameTooLongs;
counter dot3StatsInternalMacReceiveErrors;
} mib_Dot3StatsEntry;

typedef struct {
  int    dot3CollIndex;
  int    dot3CollCount;
  counter dot3CollFrequencies;
} mib_Dot3CollEntry;

/* Types for 802.5 extended MIB */

typedef char MACADDRESS[6];

typedef struct {
  int    dot5IfIndex;
  int    dot5Commands;
  int    dot5RingStatus;
  int    dot5RingState;
  int    dot5RingOpenStatus;
  int    dot5RingSpeed;
  MACADDRESS dot5UpStream;
  int    dot5ActMonParticipate;
  MACADDRESS dot5Functional;
} mib_Dot5Entry;

typedef struct {
  int    dot5StatsIfIndex;
  counter dot5StatsLineErrors;
  counter dot5StatsBurstErrors;
  counter dot5StatsACErrors;
  counter dot5StatsAbortTransErrors;
  counter dot5StatsInternalErrors;
  counter dot5StatsLostFrameErrors;
  counter dot5StatsReceiveCongestions;
  counter dot5StatsFrameCopiedErrors;
  counter dot5StatsTokenErrors;
} mib_Dot5StatsEntry;
counter dot5StatsSoftErrors;
counter dot5StatsHardErrors;
counter dot5StatsSignalLoss;
counter dot5StatsTransmitBeacons;
counter dot5StatsRecoverys;
counter dot5StatsLobeWires;
counter dot5StatsRemoves;
counter dot5StatsSingles;
counter dot5StatsFreqErrors;
} mib_Dot5StatsEntry;
MAC Types and Protocol Types

This section contains a list of device types used by the hw_ift structure (mac_type field) to indicate the device media type.

DEV_8023 For IEEE 802.3 device.
DEV_8025 For IEEE 802.5 device.
DEV_ETHER For Ethernet device.
DEV_FDDI For FDDI device.
DEV_ATM For ATM device.
DEV_FC For Fibre Channel device.

The flags (defined in sio/lan_dlpikrn.h) listed below are used by the hw_ift structure (llc_flags) to indicate the protocol type and encapsulation method.

IEEE For IEEE 802.2 type.
SNAP For SNAP type.
ETHERTYPE For Ethernet type.
NOVELL For Novell packet type.

The hw_ift structure is described in “hw_ift_t Structure Description and Initialization” on page 184, Chapter 8.
Message Types for DLPI Primitive and Acknowledgment

This section contains definitions of requests and acknowledgments for HP-UX DLPI device dependent primitives.

```c
/*
 * DL_HP_BIND, DL_HP_UNBIND, and DL_HP_LOOKUP_PROTO
 */
typedef struct {
    u_long dl_proto_kind;
    /* Kind will determine size of sap */
    u_char *dl_sap;  /* sap info. */
    u_long (*dl_proto_func)(); /* Interrupt routine */
    u_long dl_proto_info; /* Read queue pointer */
} dl_hp_proto_t;

/*
 * DL_HP_PROMISCON, DL_HP_PROMISCOFF
 */
typedef struct {
    u_long dl_level;
    /* physical, SAP level or ALL multicast */
    u_long (*dl_proto_func)(); /* Interrupt Routine. */
    u_long dl_proto_info; /* Read queue pointer */
} dl_hp_promiscon_t;

/*
 * DL_HP_ENABMULTI_ADDR, DL_HP_DISABMULTI_ADDR,
 * DL_HP_SET_PHYS_ADDR, DL_HP_GET_PHYS_ADDR, and
 * DL_HP_GET_STATISTICS.
 */
typedef struct {
    u_long dl_data_len;
    u_char *dl_data;
} dl_hp_data_t;
```

Supported DLPI primitives are summarized in “Summary of DLPI Primitives and IOCTLs” on page 223, Chapter 8.
MIB Event and Event Record

This section defines the event message that the driver is required to send an event as part of a call to network management code.

```c
struct evrec {
    struct event ev;
    struct evrec *evnext;
}

struct event {
    timeval time;    /* timestamp */
    int code;        /* event code */
    int len;         /* byte count of data in info */
    char info[MAXEVINFO]; /* event specific info */
}
```
## MTU Values

This section lists the defined MTU values (Maximum Transmission Unit without header) for the current HP provided protocols.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ETHERMTU</td>
<td>1500 bytes, max Ethernet packet size.</td>
</tr>
<tr>
<td>IEEE8023_MTU</td>
<td>1497 bytes, max IEEE 802.3 packet size.</td>
</tr>
<tr>
<td>SNAP8023_MTU</td>
<td>1492 bytes, max SNAP 802.3 packet size.</td>
</tr>
<tr>
<td>IEEE8025_4_MTU</td>
<td>4170 bytes, max packet size for 4M bit Token Ring.</td>
</tr>
<tr>
<td>SNAP8025_4_MTU</td>
<td>4170 bytes, max SNAP packet size for 4M bit Token Ring.</td>
</tr>
<tr>
<td>IEEE8025_16_MTU</td>
<td>4170 bytes, max packet size for 16M bit Token Ring.</td>
</tr>
<tr>
<td>SNAP8025_16_MTU</td>
<td>4170 bytes, max SNAP packet size for 16M bit Token Ring.</td>
</tr>
<tr>
<td>FDDI_MTU</td>
<td>4352 bytes, max SNAP packet size for FDDI.</td>
</tr>
</tbody>
</table>

These values are defined in the `netinet/if_ether.h`, `netinet/if_ieee.h`, and `sio/fddio.h` files.
Packet Headers

This section lists the packet header structures for LAN media used in HP-UX.

```c
struct ether_hdr {
    u_char destaddr[6]; /* Ethernet destination address */
    u_char sourceaddr[6]; /* Ethernet source address */
    u_short type; /* Ethernet type value */
};

struct ieee8023_hdr {
    u_char destaddr[6]; /* IEEE 802.3 destination address */
    u_char sourceaddr[6]; /* IEEE 802.3 source address */
    u_short length; /* byte count of packet length */
    u_char dsap; /* dsap value */
    u_char ssap; /* ssap value */
    u_char ctrl; /* ctrl value */
};

struct snap8023_hdr {
    u_char destaddr[6]; /* IEEE 802.3 destination address */
    u_char sourceaddr[6]; /* IEEE 802.3 source address */
    u_short length; /* byte count of packet length */
    u_char dsap; /* dsap value */
    u_char ssap; /* ssap value */
    u_char ctrl; /* ctrl value */
    u_char hdr_fill[3]; /* padding for alignment */
    u_short type; /* type value */
};

struct ieee8024_hdr {
    u_char frame_ctrl; /* frame control field */
    u_char destaddr[6]; /* IEEE 802.4 destination address */
    u_char sourceaddr[6]; /* IEEE 802.4 source address */
    u_char dsap; /* dsap value */
    u_char ssap; /* ssap value */
    u_char ctrl; /* ctrl value */
};
```
struct snap8024_hdr {
    u_char frame_ctrl; /* frame control field */
    u_char destaddr[6];
    /* IEEE 802.4 destination address */
    u_char sourceaddr[6];
    /* IEEE 802.4 source address */
    u_char dsap;    /* dsap value */
    u_char ssap;    /* ssap value */
    u_char ctrl;    /* ctrl value */
    u_char hdr_fill[3]; /* padding for alignment */
    u_char type[2];    /* type value */
}

struct ieee8025_sr_hdr {
    u_char access_ctl; /* access control field */
    u_char frame_ctrl; /* frame control field */
    u_char destaddr[6];
    /* IEEE 802.5 destination address */
    u_char sourceaddr[6];
    /* IEEE 802.5 source address */
    u_char rif[18]; /* IEEE 802.5 source routing information */
    u_char dsap;    /* dsap value */
    u_char ssap;    /* ssap value */
    u_char ctrl;    /* ctrl value */
}

struct snap8025_sr_hdr {
    u_char access_ctl; /* access control field */
    u_char frame_ctrl; /* frame control field */
    u_char destaddr[6];
    /* IEEE 802.5 destination address */
    u_char sourceaddr[6];
    /* IEEE 802.5 source address */
    u_char rif[18]; /* IEEE 802.5 source routing information */
    u_char dsap;    /* dsap value */
    u_char ssap;    /* ssap value */
    u_char ctrl;    /* ctrl value */
    u_char orgid[3]; /* organization ID */
    u_short type;    /* type value */
}

struct fddi_hdr {
    u_char pad[3];    /* pad characters */
    u_char fc;        /* frame control field */
    u_char destaddr[6]; /* IEEE 802.5 destination address */
    u_char sourceaddr[6];
}
Data Structures, Defines, Routines, Flags, and Code Examples

Packet Headers

/* IEEE 802.5 source address */
  u_char dsap;     /* dsap value */
  u_char ssap;     /* ssap value */
  u_char ctrl;     /* ctrl value */
}

struct snapfddi_hdr {
  u_char pad[3];   /* pad characters */
  u_char fc;       /* frame control field */
  u_char destaddr[6];
/* IEEE 802.5 destination address */
  u_char sourceaddr[6];
/* IEEE 802.5 source address */
  u_char dsap;     /* dsap value */
  u_char ssap;     /* ssap value */
  u_char ctrl;     /* ctrl value */
  u_char orgid[3]; /* organization ID = 00, 00, 00 */
  u_short type;    /* type value; IP = 0x800, ARP = 0x806 */
}

struct snapfddi_hdr_info {
  u_char destaddr[6];  /* FDDI destination address */
  u_short type;       /* type value */
}

struct ieee8022_hdr {
  u_char dsap;        /* dsap value */
  u_char ssap;        /* ssap value */
  u_char ctrl;        /* ctrl value */
}

struct snap8022_hdr {
  u_char dsap;        /* dsap value */
  u_char ssap;        /* ssap value */
  u_char ctrl;        /* ctrl value */
  u_char orgid[3];    /* organization ID */
  u_short type;       /* type value */
}

These structures are contained in the netinet/if_ether.h, netinet/if_ieee.h, and sio/fddio.h header files.
Packet Types

This section lists the packet type values for inbound and outbound packets.

ETHER_PKT     Ethernet packet.
SNAP8023_PKT  SNAP packet over IEEE 802.3 media.
IEEE8023_PKT  IEEE 802.3 packet.
SNAP8025_PKT  SNAP packet over IEEE 802.5 media.
IEEE8025_PKT  IEEE 802.5 packet.
SNAPFDDI_PKT  SNAP packet over FDDI media.
SNAPFDDI_LLA_PKT  SNAP (for DLPI) packet over FDDI media.
FDDI_UI_PKT   Native FDDI packet.
FDDI_LLA_PKT  Native FDDI (for DLPI) packet.

These packets are defined in the netinet/if_ether.h file.
Protocol Kinds and Values

This section lists the defined values for the `dl_proto_kind` field of HP-UX DLPI bind request.

```c
definitions protocol_kinds {
    LAN_SAP, LAN_TYPE, LAN_CANON, LAN_SNAP, LAN_SNAP_EXT
} =

LAN_SAP       For IEEE 802.[2/3] packet with SAP values as part of protocol format.
LAN_TYPE      For Ethernet packet with TYPE value as part of protocol format.
LAN_SNAP      For SNAP type of protocol format.
LAN_SNAP_EXT  For SNAP extension type protocol format.
```

These kinds are enumerated in the `sio/lan_dlpikrn.h` header described in “DLPI Sequence” on page 205, chapter 8.
SAP Values for IEEE 802.2 LLC Packets

This section lists the SAP values, defined in netinet/if_EISA.h, for IEEE 802.2 LLC packets.

IEEESAP_IP  0x06, for IP protocol
IEEESAP_SNP  0xAA, for SNAP protocol
TYPE Values for Ethernet and SNAP Protocols

This section lists the TYPE values, defined in `netinet/if_ether.h`, for Ethernet and SNAP packets.

ETHERTYPE_IP 0x0800, for IP protocol
ETHERTYPE_ARP 0x0806, for ARP protocol
C How to Design a Networking Trace/Log Subformatter and a Sample Subformatter
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Subsystems are typically an individual program or set of programs that act in concert. Each subsystem requires an associated subformatter, however several subsystems may use the same subformatter. Subformatter design depends on how logging and tracing are used in the subsystem.

Subsystems also have the capability to provide filtering or formatting options.
Designing a Subformatter

This section deals with the design of the actual function that is called in response to the formatter reading in a record containing data for a specific subsystem. If the data passes the global filters and appears to be a good message, the subformatter for the subsystem is called. The subformatter information is held in a table containing the subsystem ID, mnemonic, subformatter function, options function, message catalog, group name, and the subsystem options data structure.

The entry point for the subsystem options and formatter functions must follow a standard interface, defined in the `subsys_N_get_option` and `subsys_N_format` manpages. This interface provides the subformatter with a pointer to the buffer containing the complete message, including header and body. Additionally, parameters that are intended to be passed through to the formatter utility functions are included. These make up the bulk of the call. Most subformatter developers need only be interested in the few parameters.

The subformatter called for the subsystem must be able to handle both trace and log data. These can be separated into separate functions once the subformatter has been invoked, but there is no provision for the formatter to call more than one function for a given specific subsystem ID.

Formatting requirements for tracing are often different from logging. The developer should take this into consideration in designing the subformatters.

A subformatter developer should view the action of tracing or logging as a communication from the subsystem to the user, a user who sees only a message from the subsystem and not the medium that carried the message. The subformatter developer should consider the design of the subformatter in relation to the types of information that come from the subsystem. For logging, providing a few pieces of information, such as logging event and a couple of data items may be adequate. The subformatter can assemble the formatted output from a message in its message catalog based on the event ID, and the additional data can be inserted into the message. This method is employed by the ARPA logging subsystems.

Tracing information can be more of a problem, partly because it usually contains much more data, especially in the case of link-level packet
tracing (PDU in or out tracing). The subformatter may have to know how
the packet was constructed, which layer sits on top, and so on.

For tracing or logging, the subsystem should pass only as little data as
possible to output complete, useful information. Flags are available for
subformatters to control the format of the output data. For example,
some flags request that output be limited to one line, and other flags
request that every possible piece of information be decoded.
Subformatters are expected to tailor their output according to these
flags.

Alternative Subformatter Implementations

As shown in Figure 7-6 on page 255, Chapter 8, device driver developers
must also provide for HP-UX network trace/log data formatting in the
lib_N_fmt.sl file. Two developer-provided subsystem formatter
functions required in this file must be named subsys_N_get_option and
subsys_N_format. At the option of the developer, these routines can
accomplish their formatting by using the basic calls of netfmt (see the
following “HP-UX Subsystem Formatter Functions” section) or by using
the well-developed subformatter components in libnsfmt.sl (see the
following “HP-UX Subsystem Formatter Functions” section).

Subformatter Responsibilities

The subformatter has few responsibilities, other than transforming the
data, as follows:

1. Perform subsystem filtering (if this feature is provided)
2. Print the header.
3. If console logging is on, format a terse message; or otherwise, format a
   message in accordance with the format flags.
4. Write the formatted message.

These responsibilities can be performed with the help of the utility
functions provided for subformatters. In fact, for consistency and
efficiency, all subformatters should use these functions:

   tl_banner_check()
tl_trace_kind()
tl_log_class()
tl_header_format1()
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

Use of the functions above ensures that all headers look alike, thus helping users find useful landmarks to guide them through the output.

\texttt{tl_format_write()}
\texttt{tl_format_fprintf()}

Subformatters are free to call the output functions above for each line or for several lines of output, as appropriate.

\texttt{tl_raw_format()}

Use of the above function ensures that all hexadecimal dumps look alike and behave consistently.

By using the common functions, the underlying implementation of I/O may be changed more easily, thus allowing easier porting, further performance enhancements, alternative output schemes, etc.

Console logging is determined by a flag passed in the subformatter call. When this flag is enabled, console logging is in effect, which means that the subformatter should use a terse, one-line message instead of a more verbose explanation. The \texttt{tl_format_write()} and \texttt{tl_format_fprintf()} functions print messages on the console.

Terse (one-line) trace formatting is determined by a flag passed in the subformatter call. When this flag is enabled, the subformatter should only print one line per trace message. This mode of formatting is used to get a summary of the trace file contents. Additional flags also control the behavior of terse formatting.

Nice (detailed) trace formatting is determined by a flag passed in the subformatter call. When this flag is enabled, the subformatter should attempt to identify and label every piece of data in the trace.

If neither terse nor nice is enabled, raw formatting should be used.

\textbf{NOTE}

Each subformatter should follow Hewlett-Packard standards in formatting the data output. These standards are implemented in the HP-UX subformatters of trace and log data for Hewlett-Packard networking interfaces. Study the output of these HP-UX subformatters as templates for the data output of any new subformatter. Simple examples are provided in “HP-UX Formatting Library Routines” on page 524, for the \texttt{format_link_nice}, \texttt{format_link_terse}, and \texttt{format_link_raw} routines.
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

HP-UX Subsystem Formatter Functions

The following routines are supplied by the subsystem for the formatter to call when subsystem specific actions need to take place. Subsystem specific actions include parsing filter files and formatting the subsystem's trace or log data. These functions must be placed in a shared library.

Shared libraries are usually created by compiling all modules with the "+z" option to cc and linking them together using the options to ld:

```
-b +e subsys formatter function +e subsys get options function
```

which is then configured in the /etc/nettlgen.conf file using the nettlconf command. The +e option to ld must be used to prevent symbol collisions among the different subformatter libraries. For further explanation and details, refer to the ld(1) manpage. The formatter and user interface commands use the configuration information each time they are invoked.

Development and support of subsystem subformatters are responsibilities of the device driver developer for that subsystem.

`subsys_N_format()` The subsystem developer must provide a `subsys_N_format()` routine to format a single trace or log message from the N subsystem.

This routine, along with the shared library that contains it, is configured with the nettlconf command (see nettlconf(1M)) into the nettlgen.conf configuration file. `subsys_N_format()` is the default name; the value of N is the subsystem ID number assigned by HP (see “Assign Subsystem ID” on page 258, Chapter 8). The actual function name can be redefined with the nettlconf command.

At run time, the netfmt command loads the library and calls the routine whenever data from the subsystem is encountered. The `subsys_N_format()` routine may discard the message based on filter information supplied by the user in the options file, as determined by the `subsys_N_get_options()` routine associated with the subsystem. It returns 0 if no errors are encountered, otherwise it returns a -1.

The routine is defined as:

```
int subsys_N_format(ss_N_fmt_flag_type Flags,  
                    char *BinaryMsgPtr,  
                    char *OptionsPtr,  
                    int32_t  MsgCatFD,
```
int32_t ErrorFD,
int32_t OutputFileCount,
fp_result Outputfiles[],
char *TimeBuffer,
int32_t TimeBufferLength,
int32_t PrintOp,
int32_t UserCount,
user_acct_result Users[],
err_num *Status)

Flags
The type of Flags is defined as:

typedef struct {
    unsigned verbosity_bit: 1;
    unsigned console_logging: 1;
    unsigned highlight_bit: 1;
    unsigned nice_mode_bit: 1;
    unsigned terse_mode_bit: 1;
    unsigned terse_link_mode_bit: 1;
    unsigned terse_time_mode_bit: 1;
    unsigned map_to_names_bit: 1;
    unsigned reserved: 24;
} ss_N_fmt_flag_type;

verbosity_bit When this bit is set, a high level of verbosity has been
selected (high verbosity is the default).

console_logging This bit is set if console logging is enabled, in which
case the subformatter should only call the
tl_header_format1() routine and provide very
minimal additional information (to be kept to one line)

highlight_bit This bit is set if highlighted output is enabled
(highlighted output enabled is the default).

nice_mode_bit This bit is set when nice formatting has been enabled
(nice output not enabled is the default). Nice
formatting is the most descriptive mode of formatting.
All possible information should be displayed in this
mode of output. Nice mode is not usually used for log
messages.

terse_mode_bit This bit is set when terse formatting has been
enabled (terse output not enabled is the default). Terse
formatting should be limited to one line of output per
trace record. Terse mode is not usually used for log
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

messages.

terse_link_mode_bit
If terse mode is enabled then terse_link_mode_bit is a flag that should cause the link name to be included in the output.

terse_time_mode_bit
If terse mode is enabled then terse_time_mode_bit is a flag that should cause the timestamp to be included in the output.

map_to_names_bit
This bit is set when numbers should be resolved into names whenever possible (mapping numbers to names is enabled by default). For example, an IP address should be displayed as a hostname if the map_to_names_bit flag is set.

BinaryMsgPtr Pointer to a buffer that contains the binary trace/log message to be formatted. The buffer contains the trace/log header, struct tl_msg_hdr (as follows) from the /usr/include/ntl.h file, followed by the trace/log data (from ktrc_write or klogg_write):

typedef struct {
    unsigned short    hdr_len;
    short             subsystemid;
    int               device_id;
    tl_msg_flag_type  flags;
    set_of_32         kind;
    set_of_32         class;
    set_of_32         version;
    unsigned int      dropped_events;
    unsigned int      dropped_data;
    unsigned int      data_len;
    unsigned int      orig_data_len;
    struct timeval    time;
    int               invoke_id;
    int               path_id;
    unsigned short    log_instance;
    uid_t             uid;
    unsigned int      connection_id;
} tl_msg_hdr_type;
NOTE
For tracing, the data may be truncated by the nettl command facilities. Check the `tl_msg_hdr > data_len` field to find out how much data was captured.

**OptionsPtr**  
Pointer to a data structure defined by the subsystem for communication between the `subsys_N_get_options()` routine and the `subsys_N_format()` routine. If no options are used, then this pointer is null. The actual type of the structure pointed to by `OptionsPtr` is entirely up to the subsystem developer.

**MsgCatFD**  
File descriptor of the subsystem message catalog configured in `nettlgen.conf`. Subsystems should not open their own message catalog files.

**ErrorFD**  
File descriptor that refers to be file that will receive any error messages.

**OutputFileCount**  
Number of output files to receive the formatted trace/log messages. For HP-UX, this parameter must have a value of 1.

**OutputFiles**  
Array of structures, each of which contains a file pointer and a result.

```c
typedef struct {
    int fd;
    int result;
} fp_result;
```

This file receives the formatted trace/log messages. Only one output file is used for HP-UX, `OutputFiles[0].fd`. (`OutputFiles[0].result` is ignored.) This output file will have been opened by the formatter driver. Fatal errors on HP-UX should be reported through the return code and status parameters. Fatal and nonfatal error messages should be written to the file referenced by `ErrorFD`.

**TimeBuffer**  
String containing the formatted timestamp from the trace/log header.

**TimeBufferLength**  
Length of the `TimeBuffer` string, not counting the null

---

**Appendix C**  
521
Designing a Subformatter

terminator.

PrintOp
For HP-UX, this parameter must have a value of 0.

UserCount
For HP-UX, this parameter must have a value of 0.

Users
For HP-UX, this parameter must have a value of NULL.

Status
Contains the error value if the routine returned a −1.

subsys_N_get_options() This routine is supplied by the subsystem developer to process options for the N subsystem.

The netfmt command calls this routine whenever a filter configuration file is encountered that contains lines beginning with the subsystem name. It is subsys_N_get_options() routine's responsibility to read the subsystem specific options information from the filter command file and store any necessary information. It returns a −1 in the event of a fatal error.

This routine is defined as:

```c
int subsys_N_get_options(get_opt_parms_type *Get_Opt_Parms_Ptr)
```

define get_opt_parms_type

Defined in /usr/include/fmt.h as:

typedef struct {
    int *status_ptr;
    FILE *subsys_strm;
    FILE *error_strm;
    FILE *log_strm;
    int ss_id;
    char *ss_name;
    nl_catd ss_msg_cat;
    get_opt_flag_type ss_n_get_opt_flag;
    char **ss_options_ptr;
    int ss_output_fd;
    char *options_file_name;
    int *options_filename_printed;
} get_opt_parms_type;

status_ptr Contains the error code of the routine if the returned value is −1.

subsys_strm FILE pointer to the file that refers to the temporary file containing the options specifically for the N subsystem.
This file is created by the caller prior to invoking subsys_N_options() routine, and each line has been converted to lower case. All comments, blank lines and lines for other subsystems are already removed. In addition, the keyword identifying this subsystem has been stripped off each line, so only the options for this particular subsystem are in the file. Due to a special encoding of line number and other data, the tl_get_line() routine must be used to get option lines from this stream file.

**error_strm**  
FILE pointer to the file that will receive error messages.

**log_strm**  
FILE pointer to the file that will receive a summary of all options and files in effect for the subsystem, generated by subsys_N_get_options() routine. The nettl command reports the contents of this file after all the subsystems have finished reading their respective filter command files.

**ss_id**  
Subsystem ID number for the subsystem as found in the configuration file.

**ss_name**  
Subsystem name for the subsystem as found in the configuration file.

**ss_msg_cat**  
File descriptor pointing to the message catalog for the subsystem as found in the configuration file.

**ss_n_get_opt_flag**  
Type of flag is defined as:

```c
typedef struct {
    u_int trace_log_bit: 1;
    u_int parse_only_bit: 1;
    u_int reserved: 30;
} get_opt_flag_type;
```

**trace_log_bit**  
This flag is not needed and should not be used by subsys_N_get_options.

**parse_only_bit**  
This flag is set when the subsys_N_get_options() routine does not need to process the information in the file, only parse the input and check for syntax and semantic errors.

**ss_options_ptr**  
Pointer to a pointer to a data structure containing the
Designing a Subformatter

Specific information processed by the subsys_N_get_options() routine and passed on to the subsys_N_format() routine to handle special formatting. This structure should be allocated and initialized by subsys_N_get_options() routine.

- ss_output_fd: File descriptor referring to the file receiving the formatter output.
- options_file_name: To be filled in.
- options_filename_printed: To be filled in.

HP-UX Formatting Library Routines

The 802.3/Ethernet LAN product provides the ability to format information from upper layer protocols such as IP, TCP, UDP, ARP, DUX, and NFS, from traces taken at the link layer. This capability makes it much easier to analyze networking dialogs than examining raw hex data and manually determining what the protocols were sending.

In addition, the formatter also provides the ability to filter the trace output so that only dialogs taking place with a particular TCP port would be displayed. The filters include Ethernet type, 802.2 SAPs, IP addresses, UDP ports, and RPC information.

This existing base of capabilities makes desirable the leveraging of this code to support other link products as they are released. This section describes a set of routines available in the base netfmt product that new link products can and should take advantage of.

Link subformatters may take advantage of the ARPA decoding routines to format link level packets, as follows:

1. The link subformatter calls the appropriate set_up_*() function to prepare the decoder for filtering.
2. The link subformatter then calls filter_packet() to see if the value in the packet will pass the user-specified filters.
3. If the filters pass, then the link subformatter may call the format_link_*() functions to produce formatted output.

**NOTE**

These decoding routines are the only supported case where routines in one shared library may call those in another. Subsystems should not
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

By using these routines, a link product trace formatter needs to format only the information in its link header, not including the 802.2 information. (Other routines take care of the rest.) Note also that the trace formatter does not directly perform I/O, which is performed through the three provided formatting routines. Using these provided routines allows future changes to be made to the look of the formatted output without modifying the link format code. Using these routines also promotes consistency among links.

The following basic algorithm, for PDU_IN and PDU_OUT trace kinds only, is consistent with these objectives. (Formatting other kinds of trace messages must be done by using the routines described in the “Example: Using HP-UX Formatting Library Routines.” section.

1. Extract local link information from the packet (do not print it).
2. Call set_up_8022() to extract information for 802.2 and upper layer protocols. This routine also handles SNAP.
3. Call filter_packet() to determine if the packet meets any filter criteria. Return without printing if it fails.
4. Based on the setting of the global variables nice_fmt and terse_fmt, call one of the following routines:
   • format_link_nice()
   • format_link_terse()
   • format_link_raw()

When calling these routines, include string buffers containing any link-specific information extracted in step 1.

Figure 7-7 on page 274, Chapter 8, identifies these subroutines and shows their relationship.

set_up_8022() Routine Description This routine sets up global information used by both the filter function and the three formatting functions.

This routine walks through the buffer and copies protocol header information to appropriate global variables used by the filter and formatter. Call this routine for each PDU_IN or PDU_OUT trace event.
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

**set_up_8022(buf_ptr, len, dst_addr, src_addr)**

```c
u_char *buf_ptr;
int len;
 u_char *dst_addr;
 u_char *src_addr;
```

**buf_ptr**  Pointer to beginning of the 802.2 information. Should not include MAC info.

**len**  Length of the buffer (excluding MAC header).

**dst_addr**  Pointer to the 6-byte destination MAC address (extracted by local methods from the MAC header).

**src_addr**  Pointer to the 6-byte source MAC address (extracted by local methods from the MAC header).

**set_up_link() Routine Description**  This routine sets up global information only for the link layer and does not attempt to extract any upper layer information from the traced packet.

---

**NOTE**  Use this routine only if the packet being formatted cannot be handled by **set_up_8022**.

---

**set_up_link(buf_ptr, len, dst_addr, src_addr)**

```c
u_char *buf_ptr;
int len;
 u_char *dst_addr;
 u_char *src_addr;
```

**buf_ptr**  Pointer to the beginning of the Data Link information. Should not include MAC info. The routine does not currently use this parameter, for future extensions.

**len**  Length of the buffer (excluding MAC header).

**dst_addr**  Pointer to the 6-byte destination MAC address (extracted by local methods from the MAC header).

**src_addr**  Pointer to the 6-byte source MAC address (extracted by local methods from the MAC header).

**set_up_ip() Routine Description**  This routine walks through the buffer and copies protocol header information to the appropriate global
variables (that it sets up) for use by the filter function and the three formatting functions.

**NOTE**

Link products should not use this routine. Call this routine only when no link information is available for output formatting (for example, NS_LOOPBACK).

```c
set_up_ip(buf_ptr, len)
    u_char *buf_ptr;
    int len;

buf_ptr  Pointer to beginning of the 802.2 information, which should not include MAC information.
len      Length of the buffer, excluding MAC header.
```

**set_up_ether() Routine Description**

This routine sets up global information used by both the filter function and the three formatting functions.

This routine should be called for each PDU_IN and PDU_OUT trace event that contains Ethernet packets.

```c
buffer  Pointer to the beginning of the Ethernet data. It should not include the destination address, source address, or Ethernet type information. This routine will then walk through the buffer and copy protocol header information to appropriate global variables used by the filter and formatter.
len      Length of the buffer, excluding destination, source, and Ethernet type.
dst_addr Pointer to the 6-byte destination MAC address, extracted by local methods from the MAC header.
src_addr Pointer to the 6-byte source MAC address, extracted by local methods from the MAC header.
ether_type Ethernet-type field from the MAC header.
```

**filter_packet() Routine Description**

filter_packet() examines the globals set up by one of the preceding set_up_xxx() routines and returns 0 if the packet should not be displayed, and the subformatter should return without producing any output.
Designing a Subformatter

If the packet meets the filter criteria, a non-zero value is returned.

**format_link_nice() Routine Description** This routine formats a packet using nice formatting to display upper layer information.

```c
format_link_nice(tl_msg_hdr_type *hdr,
                u_char *buffer,
                int len,
                char *linktype,
                char *linel,
                char *addlinfo,
                char *upperinfo)
```

- **hdr**: Pointer to the standard nettl message header.
- **buffer**: Pointer to data beginning at the 802.2 level. The upper layer routines typically will not format data straight from this buffer, but the uppermost layers may display data at an appropriate offset into the buffer.
- **len**: Length of the buffer (including 802.2, excluding any lower layer data).
- **linktype**: String describing the type of link this information is carried over (for example, FDDI, 802.5, ETHER; 802.3 in the following example).
- **linel**: Short string (less than 23 bytes) giving more information to be displayed on the same line as the source address, for example, “TYPE: 0x800” for Ethernet packets (NOT SNAP), or “LENGTH: 26” for 802.3 packets (as in the following example); may be left blank by passing “”.
- **addlinfo**: Additional lines of information pertaining to data in the MAC header. (Blank for 802.3 and Ethernet, but could include formatted flags or other information in the MAC header for other link types). Should be terminated with a newline (\n).
- **upperinfo**: Other lines of information pertaining to data beyond the MAC header. Will be displayed only if the packet does not have 802.2 or Ethernet information present, that is, as in conjunction with set_up_link()). Ordinarily should be left blank. If present, you may wish to include a separator (for example, -).
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

Return Value
If any part of the formatting encounters a problem (like packet truncated or an unexpected value in a protocol header), a value of 0 is returned. If the formatting is successful a nonzero value is returned.

The following shows an example output:

[The linktype parameter goes here]-vvvvv
====================== 802.3 ============================
[The line1 parameter goes here]-vvvvvvvvvvvvvv
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-FORM AT]
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

format_link_terse() Routine Description
This routine formats a packet using terse formatting to display upper layer information in a single line.

format_link_terse(tl_msg_hdr_type *hdr,
                u_char *buffer,
                int len,
                char *linktype,
                char *addlinfo)

hdr  Pointer to the standard nettl message header.
buffer  Pointer to data beginning at the 802.2 level. The upper layer routines typically will not format data straight from this buffer.
len  Length of the buffer (including 802.2, excluding any lower layer data).
linktype  String describing the type of link this information is carried over, plus any other MAC layer information.
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

appropriate for terse mode. For 802.3 the linktype is simply “8”; for Ethernet (not SNAP) it is “E”.

addlinfo String giving other MAC or upper layer information to be displayed (blank for 802.3 and Ethernet).

The following shows and explains an example output:

8m probe vna request for: 15.13.106.63 from: 00-00-0c-00-06-31 seq: 6dcl
| |
| +-- Any addlinfo string would appear beginning here.
+ The linktype is placed here (the second character is placed by the function and describes what type of MAC address is used,
  (m)ulticast, (b)roadcast, (l)oopback, (i)ndividual.

format_link_raw() Routine Description This routine formats a packet using raw formatting to display upper layer information as hex/ASCII data.

format_link_raw(tl_msg_hdr_type *hdr,
  u_char  *buffer,
  int    len,
  int    offset,
  char   *linktype,
  char   *interface,
  char   *line3,
  char   *addlinfo)

hdr Pointer to the standard nettl message header.
buffer Pointer to entire traced packet (including MAC) use the “offset” parameter to control where the data actually begins printing.
len Length of the entire buffer.
offset Offset to actually begin displaying the data; that is, if the MAC information is not to be shown. 802.3 and Ethernet do not display until the beginning of the 802.2 information or the Ethernet data (because the Source and Dest information are formatted out).
linktype String describing the type of link this information is carried over, such as FDDI, 802.3, 802.5, or Ethernet.
interface String appended to the device ID and printed out in the
Designing a Subformatter


line3

Short string, less than 14 bytes, giving information to be displayed on the same line as the addresses: "Type: 0x800" for Ethernet packets (NOT SNAP) or "Length: 00-1a" for 802.3 packets; may be left blank by passing "".

addlinfo

(Blank for 802.3 and Ethernet, but may include formatted flags or other information in the MAC header for other link types). Terminated with a newline (\n).

The following shows an example output:

vvvvv[The linktype parameter goes here]
Received 60 bytes via 802.3 Mon Dec 02 09:22:04:33390 PST 1991
vvvvv[The interface parameter goes here]
    pid=[ICS]   interface=[lan0]
    [The line3 parameter goes here]-vvvvvvvvvvvvv
Dest: 09-00-09-00-00-01 Source: 00-00-0c-00-06-31 Length: 00-1a
< [The addlinfo parameter info goes here]
14: fc fc 03 00 00 05 03 03 05 03 00 11 00 10 6d c1........m.
30: 00 08 00 06 00 01 0f 0d 6a 3f d8 68 fd f1 0c 20..j?.h...
46: e3 ff 07 50 18 80 00 00 00 00 0c 02...P........

Example: Using HP-UX Formatting Library Routines
The code sample below shows HP-UX formatting library routines including complete description of each line of code.

my_formatter(....)

/* Call for PDU_IN and PDU_OUT trace kinds ONLY */
{
  struct local_hdr_type local_mac_hdr, *my_hdr;
  tl_msg_hdr_type *hdr;
  char temp_buf[80];
  char *buffer, orig_buffer;
  int size, orig_size;
  int ret;

  /* extract TL message header and data */
  /* set buffer and orig_buffer to point to data and */
  /* size and orig_size to the length of traced info */
Designing a Subformatter

/* now extract MAC specific information from trace info */
memcpy(&local_mac_hdr, buffer, sizeof(struct local_hdr_type));
my_hdr = &local_mac_hdr;

/* now bump buffer and size to reflect beginning of 802.2 */
/* information... */
buffer += LOCAL_HDR_SIZE;
size -= LOCAL_HDR_SIZE;

/* call setup routine to set up structures reflecting the */
/* 802.2 and above level headers. Handles SNAP as well */
set_up_8022(buffer, size, my_hdr->dst_addr, my_hdr->src_addr);

/* the routine filter packet will indicate whether the */
/* current packet meets the user specified filter criteria */
/* filter uses the global info setup by set_up_8022. */
/* (i.e. IP address, 802.2 SAP, Ether type, TCP port... ) */
if (!filter_packet())
  /* display no info if filter fails */
  return;

/* call the terse formatter if flag set */
if (terse_fmt) {
  format_link_terse(hdr, buffer, size, "Z", "");
}

/* always return after terse, the caller only wants 1 line */
/* of information, so never fall through to format_link_raw */
return;

/* set up "temp_buf" with any short link specific info.*/
/* If we had longer info to pass about the link hdr, */
/* we pass it as the last "addl_info" parameter to */
/* fmt_link_nice() or fmt_link_raw(), in this case we */
/* just pass "", blank. */
sprintf(temp_buf, "FLAGS: %4x", my_hdr->flags);

/* otherwise call the nice formatter if nice flag set */
if (nice_fmt){

/* always return after terse, the caller only wants 1 line */
/* of information, so never fall through to format_link_raw */
return;

/* set up "temp_buf" with any short link specific info.*/
/* If we had longer info to pass about the link hdr, */
/* we pass it as the last "addl_info" parameter to */
/* fmt_link_nice() or fmt_link_raw(), in this case we */
/* just pass "", blank. */
sprintf(temp_buf, "FLAGS: %4x", my_hdr->flags);

/* otherwise call the nice formatter if nice flag set */
if (nice_fmt){
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

```c
ret = format_link_nice(hdr, buffer, size, "802.Z", temp_buf, ",", ");

/* if the nice formatting failed, fall through to */
/* raw formatting. Otherwise return. */
if (ret) return;
}
format_link_raw(hdr, orig_buffer, orig_size, orig_size-size,
"802.Z", "zan", temp_buf, ");
return;
```

HP-UX Subsystem Formatter Calls

In case the HP-UX formatting library routines are inadequate for the formatter developer (infrequent), HP-UX also provides a full library of low-level formatting calls for developing a formatter “from the ground up.” This section details these calls.

A formatter developer wishing to use the HP-UX network trace and log data formatting calls to develop a subsystem formatter must include the following in the source code.

```c
#include <fmt.h>
This file contains the necessary data structure for the format support calls.
#include <ntl.h>
This file contains the necessary data structure for the trace and log data.
#include <subsys_id.h>
This file contains subsystem identification information and definitions for log classes and trace kinds.
The following function calls are provided to subsystems for formatting trace and log data and are provided to subsystem formatters in the format library libfmt.sl.
```

**NOTE**

Subsystems should not link with the `libfmt.sl` library. All externals are resolved during dynamic loading at run time.
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

tl_header_format1() This routine is called to format a single trace or log header.

The format of the output will conform to the standard HP-UX network tracing and logging recommendations. The formatted header will be written to the output file referenced by output_file[0].fd. The tl_header_format1() routine must be called by every subformatter. At a minimum, this may be the only output generated by the subformatter. If an error occurs inside the tl_header_format1() routine, −1 is returned. Otherwise, if no errors occurred, 0 is returned. Fatal errors are reported through the return value and the status_ptr parameter. All error messages are written to the file pointed to by the error_fd parameter. The error codes are:

FMTERR_INV_HDR_PTR Trace/log header pointer is invalid.
FMTERR_INV_HDR Trace/log header is invalid (corrupt).
FMTERR_INV_OUT_FD Output file descriptor is invalid.
FMTERR_INV_MC_FD Message catalog descriptor is invalid.
FMTERR_SYS_ERROR An error was returned from a system call within tl_header_format1().

The tl_header_format1() routine is defined as:

```c
int tl_header_format1(char *header_ptr,
                      int error_fd,
                      ss_N_fmt_flag_type flags,
                      char *kind_str,
                      char banner_char,
                      int output_file_count,
                      fd_result output_files[];
                      char *time_buffer,
                      int time_buffer_length,
                      int print_op,
                      int user_count,
                      user_acct_result users[],
                      int location, err_num *status_ptr)
```

header_ptr Points to a buffer that contains the header of the trace/log message to be formatted.

error_fd File descriptor that refers to the file that will receive error messages.

flags Type of flag is defined as:
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

typedef struct {
    unsigned verbosity_bit: 1;
    unsigned console_logging: 1;
    unsigned highlight_bit: 1;
    unsigned nice_mode_bit: 1;
    unsigned terse_mode_bit: 1;
    unsigned terse_link_mode_bit: 1;
    unsigned terse_time_mode_bit: 1;
    unsigned map_to_names_bit: 1;
    unsigned reserved: 24;
} ss_N_fmt_flag_type;

This structure is defined in /usr/include/fmt.h.

kind_str

May indicate a text message (typically the result of the tl_log_class or tl_trace_kind function) to be displayed for the kind field from the trace/log header. This string must be null-terminated. The kind message is truncated to 16 characters. If kind_str is NULL, the kind field from the header is displayed as a decimal value.

banner_char

Character to use in the banner header line (typically the result of the tl_banner_check function). The subformatter may use this character to indicate differences in messages, such as inbound or outbound messages. For example, inbound messages could use the character “v”, while outbound messages could use the character “^”.

output_file_count

Number of output files to receive the formatted trace/log header output. For HP-UX only one output file is used, and so this value is always 1.

output_files

Array of structures consisting of a file descriptor and result variable for each file to receive the formatted trace/log header output. For HP-UX, only one output file is used: output_file[0].fd.

time_buffer

Contains a string depicting the formatted time stamp from the trace/log header.

time_buffer_length

Contains the length of time_buffer not counting the null terminator byte.
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

print_op  For HP-UX, this parameter must have a value of 0.
user_count  For HP-UX, this parameter must have a value of 0
users  For HP-UX, this parameter must have a value of NULL.
location  Value which can be used to locate the source of the message in the code. This parameter is set by the subsystem and may be used to represent any information the subsystem desires.
status_ptr  Contains the error value if the routine returns a -1.

tl_format_write()  This routine is called to write the decoded buffer to stdout.

The tl_format_write() routine prints a buffer pointed to by output_file[0].fd. The buffer may be created by one or more calls to the sprintf() C library function. If an error occurs inside the tl_format_write() routine, -1 is returned. Otherwise (no error occurred), 0 is returned. Fatal errors are reported through the return value and status_ptr parameter. All error messages are written to the file pointed to by error_fd. The error codes are:

FMTERR_FORMAT_WRITE   An error has occurred in writing to the output files.
FMTERR_INV_OUT_FD  Invalid output file descriptor.
FMTERR_INV_L_STR   Invalid line pointer string.
FMTERR_SYS_ERROR   An error has been returned from a system call within tl_format_write() routine.

The tl_format_write() routine is defined as:

int tl_format_write(u_char *input_line_ptr,
                   int input_line_byte_count,
                   int error_fd,
                   fmt_wrt_flag_type flags,
                   int output_file_count,
                   fd_result output_files[],
                   int print_op,
                   int user_count,
                   user_acct_result users[],
                   err_num *status_ptr)
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

input_line_ptr  Character string that contains the message to be printed on the outfile files. input_line_ptr need not be null-terminated or end with a newline.

input_line_byte_count  Byte-count of input_line_ptr message string.

error_fd  File descriptor pointing to a file to receive error messages from tl_format_write() routine.

flags  Controls output behavior of the tl_format_write() routine. The value must be set before calling tl_format_write().

typedef struct
{
  unsigned highlight: 1;
  unsigned wait_to_write: 1;
  unsigned reserved: 30;
} fmt_wrt_flag_type;

highlight  Write the input_line_ptr data in inverse video.

wait_to_write  Reserved for future use.

output_file_count  Number of output files to receive the formatted trace/log header output. For HP-UX only one output file is used, and the value is always 1.

output_files  Array of structures consisting of a file descriptor and result variable for each file to receive the formatted trace/log header output. For HP-UX only one output file is used; output_file[0].fd refers to the file receiving the formatter output.

print_op  For HP-UX, this parameter must have a value of 0.

user_count  For HP-UX, this parameter must have a value of 0.

users  For HP-UX, this parameter must have a value of NULL.

status_ptr  Contains the error value if the routine returns a -1.

tl_format_fprintf()  This routine is called to convert, format, and print its arguments under control of the format.

It prints the formatted buffer to stdout and is defined as:
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

```c
int tl_format_fprintf(FILE *stream,
    fmt_wrt_flag_type flags,
    error_num *status_ptr,
    char *format, /* [,,arg] */ ...) {
    stream One of the FILE streams contained in the
    ss_N_fmt_parms_type structure returned by
    tl_get_parms.
    flags Controls the output behavior of the
    tl_format_fprintf() routine. The value must be
    set before calling tl_format_fprintf().
    typedef struct {
        unsigned highlight : 1;
        unsigned wait_to_write : 1;
        unsigned reserved : 30;
    } fmt_wrt_flag_type;
    highlight Write the format data in inverse video.
    wait_to_write Reserved for future use.
    status_ptr Contains the error value if the routine returns a
        -1.
    format The format character string contains two types of
        objects: plain characters that are copied to the output
        stream, and conversion specifications. Each string
        results in fetching 0 or more arguments. The results
        are undefined if there are insufficient args for the
        format. If the format is exhausted while arguments
        remain, the excess arguments are ignored.

    This routine behaves like printf(). For detail see
    tl_format_fprintf(NET3).

tl_raw_format() This routine is called to format a trace or log
        message into both hexadecimal and printable ASCII characters.

    The raw formatted output will appear as follows:
    0 : 73 61 6d 70 6c 65 5f 6c 6f 67 5f 64 61 74 61 2e sample_log_
        data
    16: 20 6d 6f 72 65 5f 64 61 74 61 20 61 73 64 66 6a more_data
        asdfj

    The left-most column gives the decimal byte offset. The center area is the
    hexadecimal display of the data. The right-most column is the printable
    ASCII display of the data. A period will be displayed for any nonprinting
```
character. If an error occurs inside the `tl_raw_format()` routine, a -1 is returned. Otherwise, if no errors occurred, 0 is returned. Fatal errors are reported through the return value and `status_ptr` parameter. All error messages are written to the file pointed to by `error_fd`. The `tl_raw_format()` routine is defined as:

```c
int tl_raw_format(unsigned char *data_ptr,
    int num_bytes,
    int start,
    int error_fd,
    raw_fmt_flag_type flags,
    int output_file_count,
    fd_result output_files[],
    int print_op,
    int user_count,
    user_acct_result users[],
    err_num *status_ptr)
```

- `data_ptr` Pointer to the buffer that contains the data to be dumped in hexadecimal form.
- `num_bytes` Number of bytes to dump from the buffer pointed to by `data_ptr`. There is no check to ensure that the number of bytes given does not exceed the actual buffer length. If `num_bytes` is zero, then no data will be dumped.
- `start` Offset into the buffer pointed to by `data_ptr` indicating where the dump should begin. If `start` is zero, the dump will begin at the byte pointed at by `data_ptr`.
- `error_fd` File descriptor that will receive error messages.
- `flags` Reserved for future use. Value should be set to 0 by the caller.
- `output_file_count` Number of output files to receive the raw dump. For HP-UX, this parameter must have a value of 1.
- `output_files` Array of structures, each of which contains a file descriptor and a result code for the last operation on the file. For HP-UX, only one output file is used; `output_file[0].fd` refers to the file receiving the formatter output.
- `print_op` For HP-UX, this parameter must have a value of 0.
- `user_count` For HP-UX, this parameter must have a value of 0.
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

users For HP-UX, this parameter must have a value of NULL.

status_ptr Contains the error value if the routine returns a -1.

tl_get_parms() This routine returns to the caller a pointer to an ss_N_fmt_parms_type data structure containing parameters that a subsystem subformatter needs in order to operate.

The core formatter builds and initializes this data structure before calling subsys_N_format().

The tl_getParms() routine is defined as:

ss_N_fmt_parms_type *tl_get_parms()

The ss_N_fmt_parms_type type is defined as:

typedef struct {
    int *ss_status_ptr;
    FILE *ss_output_strm;
    int ss_output_fd;
    FILE *ss_error_strm;
    int ss_error_fd;
    nl_catd ss_msg_cat;
    char *ss_name;
    char *ss_binary_msg_ptr;
    char *ss_options_ptr;
    ss_N_fmt_flag_type ss_n_fmt_flags;
    char *time_buffer;
    int time_buffer_length;
    int output_file_count;
    fd_result output_files[1];
    int print_op;
    int user_count;
    user_acct_result *users;
    int inited_flag;
    int nettl_version;
} ss_N_fmt_parms_type;

This data structure is defined in /usr/include/fmt.h, and the parameters are as follows:

ss_status_ptr Used by a subformatter to store an error code if it fails.

ss_output_strm FILE pointer that will receive the formatted trace/log message. This field must be initialized before calling the tl_get_parms() routine.
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

ss_output_fd File descriptor that will receive the formatted trace/log messages.

ss_error_strm FILE pointer that will receive any fatal or nonfatal error messages.

ss_error_fd File descriptor that will receive any fatal or non fatal error messages.

ss_msg_cat Message catalog descriptor to be used in catgets.

ss_name Pointer to the subsystem name.

ss_binary_msg_ptr Pointer to a buffer containing log/trace messages to be formatted.

ss_options_ptr Pointer to a buffer containing information to be passed between the subsys_N_format() routine. See OptionsPtr in “subsys_N_format()” on page 518.

ss_n_fmt_flag Options flags: the type is defined as:

```c
typedef struct {
    unsigned verbosity_bit: 1;
    unsigned console_logging: 1;
    unsigned highlight_bit: 1;
    unsigned nice_mode_bit: 1;
    unsigned terse_mode_bit: 1;
    unsigned terse_link_mode_bit: 1;
    unsigned terse_time_mode_bit: 1;
    unsigned map_to_names_bit: 1;
    unsigned reserved: 24;
} ss_N_fmt_flag_type;
```

See TimeBuffer in the subsys_N_format function call.

tl_check_cat_version() This routine checks that the subsystem message catalog has a compatible version with the subsystem formatter library.

It returns 0 if the versions match, and -1 if they don't, or the file descriptor of the message catalog is invalid.

The tl_check_cat_version() routine is defined as:

```c
int tl_check_cat_version(int MsgCatFd,
                        int SetNum,
                        int MsgNum,
                        char *ExpectedVersion,
```

Appendix C 541
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

```c
FILE *ErrStrm)
MsgCatFd    File descriptor of the message catalog which contains
            the version string.
SetNum      Set number in the message catalog.
MsgNum      Message number in the message catalog.
ExpectedVersion
            Version string that the message catalog is expected to
            contain.
ErrStrm     FILE pointer to a stream that will receive error
            messages.
```

tl_banner_char()  This routine obtains the character to be used when
                  printing a header banner with the tl_header_format1() function.

The character is based on the type of log class or trace kind. This
function helps to ensure that banners are consistent for all subsystems.

```c
char tl_banner_char(unsigned int kind_class)
kind_class    Trace kind or log class of the message.
```

tl_trace_kind()  This routine returns a text interpretation of a trace
                 kind.

The trace kind is stored as an integer. This function converts that
number into a string that can be used in the formatted output. For
example, passing in a trace kind of 0x80000000 causes the return value
to be HDR_IN_TRACE. The result of tl_trace_kind() is typically used
as a parameter to tl_header_format1() when printing a header.

```c
char *tl_trace_kind(unsigned int kind)
kind        Trace kind of the message.
```

tl_log_class()  This routine returns a text interpretation of a log
                 class.

The log class is stored as an integer. This function converts that
number into a string that can be used in the formatted output. For example,
passing in a log class of 8 causes the return value to be DISASTER. The
result of tl_log_class() is typically used as a parameter to
tl_header_format1() when printing a header.

```c
char *tl_log_class(unsigned int class)
```
**Subformatter Option**

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.

Options processing is performed as that of the formatter itself; that is, when it recognizes a subsystem mnemonic it passes that line to the subsystem options function. The subsystem options function is responsible for parsing and determining the contents of the line. By the time the options function receives the line, the mnemonic has been stripped off and all strings have been converted to lower case. The only restriction on the contents of the line are that it cannot exceed 2048 bytes and must contain only printable characters. The `tl_get_line()` function (see the HP-UX Driver Development Reference) must always be used to read options lines from the options file.

Subsystems may adopt this technique to alter the level of information (beyond terse and verbose), to include extra kinds of data, to provide extra filtering (events or certain trace or log messages for data not covered by the global filtering functions), and so on.

`tl_get_line()` This routine obtains a line from a filter command file according to the following steps:

1. The core formatter reads the filter command file, collects the lines specific to a subsystem and then edits and stores them into a temporary file.
2. It then calls the `subsys_N_get_options()` routine with parameter set as a pointer to this temporary file.
3. the `subsys_N_get_options()` routine can call `tl_get_line()` routine to extract one line at a time from this temporary file for processing.
4. The `tl_get_line()` routine returns 0 for EOF, a negative number for failure, and 2 for success.

It is defined as:

```c
int tl_get_line(
    FILE *stream,
```
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

```c
char *buf,
int bufsize,
char *org_buf,
int LineNo,
FILE *ErrorStream
);
```

- `Stream`:
  FILE pointer that points to the temporary filter command file containing a single subsystem's filter commands.

- `buf`:
  Stores the “cleaned” filter command line in this buffer.

- `bufsize`:
  Size of Line (no more than 2048).

- `orgbuf`:
  Stores the original filter command line as it appeared in the filter command file in OrigLine.

- `LineNo`:
  Stores the line number of OrigLine in LineNo as it appeared in the filter command file.

- `ErrorStream`:
  FILE pointer to a stream that will receive error messages.

Internationalization and Message Catalog Support

The formatter provides the subformatters the capability to use the National or Native Language Support (NLS) facilities in HP-UX. When registering the NLS subsystem with the tracing and logging system at installation, the name of the message catalog to be used by the subformatter must also be provided.

The message catalog is called as follows:

1. **netfmt** opens and closes the message catalog by using the **catopen()** and **catclose()** calls.
2. The file descriptor returned by the **catopen()** is passed to the subformatter.
3. If no message catalog is registered, or if the message catalog cannot be opened, a special file descriptor of −1 meaning “no file” is passed in.

The subformatter should perform the appropriate **catgets()** calls to retrieve their messages from the message catalog. Subsystems should not open their own message catalogs or use multiple message catalogs.

The commonly accepted method of using message catalogs is to use the **catgets()** call, providing the English language string as the default to
the call if the message catalog read fails. This should be the same string
the call would retrieve from the default message catalog, located
typically in /usr/lib/nls/msg/C/name.cat, where name is the name
registered with netfmt.

One recommendation for using message catalogs effectively is to have
each logging event correspond to a message number, which makes
processing and retrieval simpler. Different message sets or an offset can
be used for terse (console) and verbose messages.

Because the message catalogs can be altered for a given location, the
subformatter should also put some kind of identifying tag (such as
“FTAM 489”) on the message that is not localized. Support personnel in a
different location will then be able to understand what is being logged
without trying to translate the text of the message.

Due to the subformatter’s dependency on message catalogs to provide the
correct text for a log event, the version of the catalog is highly dependent
on the version of the subsystem. The tl_check_cat_version() function
(see tl_check_cat_version(NET3)) is provided to facilitate checking of
message catalog versions.

Configuring Developed Subsystems into the System

The process for getting the tracing and logging facility to know about
developed subsystems is somewhat complex. Subsystems must inform
the tracing and logging facility of their existence at install/update time.

Each fileset is required to have an SDU configure script. Tracing and
logging take advantage of this independence to facilitate the
configuration of subsystems into the nettl and netfmt commands.

The nettlconf script has the capability to configure the subsystem (see
the nettlconf(1M) manpage). nettlconf should be called from within the
configure script during an SDU update or installation. nettlconf
configures the subsystem information and puts it into the
/etc/nettlgen.conf data base file.

The nettl and netfmt commands use the information in the
/etc/nettlgen.conf data base file to configure themselves at run time.
For the netfmt command, the subsystem’s subformatters (in shared
libraries) are dynamically loaded so all symbols can be resolved. The
nettl command uses the subsystem names and initial log classes to
build the tables necessary to control subsystem operations.

The information that the subsystems need to configure include:
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

- Subsystem ID: Assigned to your subsystem by Hewlett-Packard (see “Assign Subsystem ID” on page 258, Chapter 8).

- Subsystem Mnemonic: This is the name by which the subsystem will be identified in nettl and in the formatted header printed by netfmt. It is a string that may consist of alphanumerics (beginning with a letter) and may contain underscores. Blanks are not allowed.

- Default Logging Class: This is a mask containing the level of logging to be enabled when the logging facility starts up. This level may be changed by subsequent calls to nettl.

- Subsystem Space Type: This is a flag that identifies user-space subsystems and kernel-space subsystems. The two types of subsystems are handled differently within the nettl command.

- Subsystem Formatting Function: This is the C function name used to call the function that supports formatting for the subsystem. This function must be contained in the subsystem formatter shared library.

- Subsystem Options Function: This is the C function that is called to process options specified by the user in the netfmt options file. Only the OTS, LAN, ARPA, and X.25 subsystems use this feature. The formatter uses this function to set up global filtering and formatting information as well. This function must be contained in the subsystem formatter shared library.

- Subsystem Group Name: Each subsystem belongs to some logical group, usually a product. This group name is included on the banner printed during formatting. Although this group name can be any ASCII string, it should definitely contain the subsystem product name. For example, all X.25 subsystems use the group name “X.25/9000 Networking”.

- Subsystem Formatter Message Catalog: This is the name of the message catalog used by the subsystem formatter functions. This is typically an unqualified name, that is, the base name of the catalog with no path or .cat extensions. For example, the default message catalog for the formatter is netfmt.cat, and it resides in the default NLS directory, /usr/lib/nls/msg/C. This could be specified simply as netfmt. However, if the message catalog does not reside in the default directory, the message catalog name must contain NLSPATH path constructors described in the environ(5) manpage. For example, for product xyz, the abc message catalog, /opt/xyz/lib/nls/msg/C/abc.cat, would be specified as

---

Appendix C


How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

/opt/xyz/lib/nls/%L/abc.cat. This is so the end user of the formatter can use other message catalogs and control them with the LANG and NLSPATH environment variables. This restriction requires subsystems to load their standard English catalog into the C directory under their nls paths (this is the standard place for the shipped message catalogs).

Sample Subformatter Configure Script

The fileset configure script should perform the configuration of all subsystems contained in the fileset. The following fragment is from an SDU control script to perform the configuration:

```bash
#!/usr/bin/posix/sh
#
# Product:
# Fileset: NETTL-MIN
# configure
# @(#) $Revision: 1.2 $
#
# (c) Copyright Hewlett-Packard Company 1993
#
#
set -a # Export all vars
exitval=0 # Anticipate success

: ${UTILS:="/usr/lbin/sw/control_utils"}
if [ ! -f $UTILS ]
then
    echo "ERROR: Cannot find $UTILS"
    exit 1
fi

: ${FILESET:="NETTL-MIN"}
: ${NETTLCONF:="$(SW_ROOT_DIRECTORY)usr/sbin/nettlcomf"}
: ${NETFMT:="$(SW_ROOT_DIRECTORY)usr/sbin/netfmt"}
if [ ! -x "$NETTLCONF" ] ; then
    echo "ERROR: Cannot find $NETTLCONF"
    exit 1
fi
```

Appendix C
Designing a Subformatter

# Subsystem A
$NETTLCONF -S -id 0 -name SUBSYSTEM_A -class 12 -kernel\ 
  -lib libsubsystem_A.sl -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.sl -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.sl -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`

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 0
Network Trace/Log Subsystem Installation Testing

Subsystem developers must perform complete installation testing on their subsystems. As described in the previous section, the network trace/log facility is configured at installation time by a registration process that occurs in the subsystems configure script. This process tells the `netfmt` and `nettl` commands the IDs of the subsystems that exist on the system and gives information about how the subsystems are to be controlled and formatted. Only those subsystems that are registered are allowed to be turned on for logging and tracing or have their records formatted appropriately.

The `nettlconf` command does not check the parameters that are passed to it. The subsystem must check that the information to be stored in the configuration database is correct. Subsystems must test their installation for all possible environments, including multi-user systems, workstations, and diskless clusters.

NOTE

The registration scheme has the potential to break tracing and logging for all subsystems if the configuration becomes corrupt or if the information that is given is invalid. Subsystems should test and review the procedures used to configure their subsystems into the network trace/log facility.

The problems described in the following list are very common and can cause the configuration file to be unusable. All of these problems are preventable with proper understanding and testing of the subsystem configuration process:

- The subsystem subformatter library or message catalog cannot be found or opened except by superuser.
- A field in the `nettlgen.conf` subsystem configuration database file is corrupted.
- Symbols in the subformatter library conflict with symbols exported from other subformatter libraries of other subsystems. (This situation cannot occur if the `-e` option to ld is used when creating the subformatter library.)
- Symbols remain unresolved after `netfmt` has loaded the subformatter libraries of all configured subsystems.
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

- The function name of the `subsys_N_format` or `subsys_N_get_options` functions cannot be found in the specified subformatter library.

- The subsystem name or ID is in use by another subsystem. (This cannot happen if subsystems use the subsystem names and ID numbers assigned by the OpenConnect Team as described in “Assign Subsystem ID” on page 258.)
D

Shared Library Examples for the driveradmin and lanscan Commands
Shared Library Examples for the driveradmin and lanscan Commands

This appendix contains examples of the `lanscan` and `driveradmin` shared library code and the `driveradmin` sample code.
The lanscan’s shared library sample code

/****************************************************************
** $Revision: 1.1.119.2 $**
** $Date: 99/05/24 15:45:05 $**
****************************************************************

(C) COPYRIGHT HEWLETT-PACKARD COMPANY 1999. ALL RIGHTS
RESERVED. NO PART OF THIS PROGRAM MAY BE PHOTOCOPIED,
REPRODUCED, OR TRANSLATED TO ANOTHER PROGRAM
LANGUAGE WITHOUT THE PRIOR WRITTEN CONSENT OF
HEWLETT-PACKARD COMPANY.

*****************************************************************
*****************************************************************

This file implements the common function peenet which each of the
pe<driver name> functions calls when lanscan -v is executed for a
enetlink.

***************************************************************************/
#endif
#endif

#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stdsysm.h>
#include <sys/types.h>
#include <errno.h>
#include <nlist.h>
#include <netio.h>/* Direct Access header definition */
#include <nl_types.h>

#include <machine/param.h>
#include <machine/pde.h>
#include <sys/libIO.h>
#include <sio/llio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <net/if.h>
#include <netinet/if_ether.h>
#include <memory.h>

#ifndef LOCALINC
#include <sio/lan_dlpikrn.h>
#else
#include "lan_dlpikrn.h"
#endif

nl_catd catfd;
struct {
    char *s;
    int nlix;
} encaps[] = {
    "IEEE", 0/* llc_flags 0x0001 */
    "HPEXTIEEE", 1/* llc_flags 0x0002 */
    "SNAP", 2/* llc_flags 0x0004 */
}
The lanscan's shared library sample code

```c

char sp = ' '; int msg_base=80;

peenet(hwift)
    hw_ift_t *hwift;
    {
        int j, count;
        int ix = 1;

        catfd = catopen("peenet", 0);
        for (j = 0, ix = 1; j < 6; j++, ix <<= 1) {
            if (hwift->llc_flags & ix) {
                printf ((catgets (catfd, 1, msg_base + encaps[j].nlix,
                                encaps[j].s)));
                printf ((catgets (catfd, 1, 19, "%c"), sp));
            }
        }
    } /* end peenet() */

```
The driveradmin’s shared library sample code

******************************************************************
** $Revision: 1.1.119.2 $**
** $Date: 99/05/30 15:40:49 $**
******************************************************************
******************************************************************
** (C) COPYRIGHT HEWLETT-PACKARD COMPANY 1996. ALL**
** RIGHTS RESERVED.**
** NO PART OF THIS PROGRAM MAY BE PHOTOCOPIED,**
** REPRODUCED, OR TRANSLATED**
** TO ANOTHER PROGRAM LANGUAGE WITHOUT THE PRIOR**
** WRITTEN CONSENT OF HEWLETT-PACKARD COMPANY.**
******************************************************************
***
***
*** dsenet.c
******************************************************************
*
* This file implements the shared library using by the enet driver.
*
*
* Modification History
*
* __________  ________________________________
*
*
\******************************************************************/
#f defined(MODULEID) &\& !defined(lint)
static char rcsid[]="@(#) $Header: dsenet.c,v 1.1.119.2 99/05/30 15:40:49
lwen Exp $ $Revision: 1.1.119.2 $ $Date: 99/05/30 15:40:49 $";
#endif

#ifndef lint
static char HPPROD_ID[]="@(#)libdenet.sl: Version: B.11.00 $Date:
99/05/30 15:40:49 $";
#endif

#include <stdio.h>
#include <errno.h>
#include <nl_types.h>
#include <sio/llio.h>
#include <sys/stropts.h>
#include <sys/mib.h>
#include <sys/dlpi.h>
#include <sys/dlpi_ext.h>
#include <netio.h>

#include "enetadmin.h" /* message header file definition */

extern int errno; /* error indication for system calls */

nl_catd catfd;
char*driver_name ;
terrickbase;
charerrbuf[128] ;
Shared Library Examples for the driveradmin and lanscan Commands
The driveradmin’s shared library sample code

```c
struct {
    char  *s;
    int   nlix;
} iftypes[] = {
    "INVALID(%d)\ n",4100,
    "other(%d)\ n",4101,
    "regular1822(%d)\ n",4102,
    "hdh1822(%d)\ n",4103,
    "ddn-x25(%d)\ n",4104,
    "rfc877-x25(%d)\ n",4105,
    "ethernet-csmacd(%d)\ n",4106,
    "iso88023-csmacd(%d)\ n",4107,
    "iso88024-tokenBus(%d)\ n",4108,
    "iso88025-tokenRing(%d)\ n",4109,
    "iso88026-man(%d)\ n",4110,
    "starLan(%d)\ n",4111,
    "proteon-10MBit(%d)\ n",4112,
    "proteon-80MBit(%d)\ n",4113,
    "hyperchannel(%d)\ n",4114,
    "fddi(%d)\ n",4115,
    "iapb(%d)\ n",4116,
    "sdlc(%d)\ n",4117,
    "t1-carrier(%d)\ n",4118,
    "cept(%d)\ n",4119,
    "basic-sdn(%d)\ n",4120,
    "primary-sdn(%d)\ n",4121,
    "prop-PointToPointSerial(%d)\ n",4122,
};
```
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin’s shared library sample code

```c
#include <stdio.h>

int invalid(int n)
{
    printf("Invalid(%d)\n", n);
    return 0;
}

int up(int n)
{
    printf("Up(%d)\n", n);
    return 0;
}

int down(int n)
{
    printf("Down(%d)\n", n);
    return 0;
}

int testing(int n)
{
    printf("Testing(%d)\n", n);
    return 0;
}

int main()
{
    struct {
        char *s;
        int nlix;
    } adopstatus[] = {
        invalid(100),
        invalid(200),
        up(101),
        down(102),
        testing(103)
    };

    // Global areas for sending and receiving stream messages
    #define AREA_SIZE 5000 /* bytes; big enough for largest possible msg */
    #define LONG_AREA_SIZE (AREA_SIZE / sizeof(u_long)) /* AREA_SIZE/4 */

    u_longctrl_area[LONG_AREA_SIZE]; /* for control messages */
    u_longdata_area[LONG_AREA_SIZE]; /* for data messages */

    struct strbuf ctrl_buf = {
        AREA_SIZE, /* maxlen = AREA_SIZE */
        0, /* len gets filled in for each message */
        ctrl_area /* buf = control area */
    };
```

Appendix D 559
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin's shared library sample code

```c
struct strbuf data_buf = {
    AREA_SIZE, /* maxlen = AREA_SIZE */
    0, /* len gets filled in for each message */
    data_area/* buf = data area */
};

dsenet (fd, cur_ppa, termlines)
int fd;
int cur_ppa;
termlines;
{
    dl_get_statistics_req_t *get_statistics_req =
        (dl_get_statistics_req_t *) ctrl_area;
    dl_get_statistics_ack_t *get_statistics_ack =
        (dl_get_statistics_ack_t *) ctrl_area;
    mib_ifEntry*mib1p;
    mib_Dot3StatsEntry*mib2p;
    mib_Dot3CollEntry*mib3p;
    int getstats_failed = 0;
    u_int*addrintp;
    u_short*addrshortp;

catfd = catopen ("driver_name", 0);

    /* Get statistics on LAN Interface device using DLPI */
    /* send the DL_GET_STATISTICS_REQ and wait for the OK_ACK */
    get_statistics_req->dl_primitive = DL_GET_STATISTICS_REQ;
```
if (put_ctrl(fd, sizeof(dl_get_statistics_req_t), 0) == -1)
getstats_failed = TRUE;
else if (get_msg(fd) == -1)
getstats_failed = TRUE;
else if (check_ctrl(DL_GET_STATISTICS_ACK) == -1)
getstats_failed = TRUE;

if (getstats_failed) {
    sprintf (errbuf, "\%s: unable to get statistics\n", driver_name);
    fprintf(stderr, (catgets (catfd, 1, errbase + 1, errbuf)));
    catclose(catfd);
    return;
}

mib1p = (u_char *) ctrl_area + get_statistics_ack->dl_stat_offset;
printf((catgets(catfd,1,105,
    "Description               = \%s \n"), mib1p->ifDescr);
printf((catgets(catfd,1,106,
    "Type (value)              = \
")), mib1p->ifType);
printf((catgets(catfd,1,107,
    "MTU Size                  = \%u \n")), mib1p->ifMtu);
printf((catgets(catfd,1,108,
    "Speed                     = \%u \n")), mib1p->ifSpeed);
addrintp = mib1p->ifPhysAddress.o_bytes;
addrshortp = &((mib1p->ifPhysAddress.o_bytes[4]);
printf((catgets(catfd,1,109,
The driveradmin’s shared library sample code

```
"Station Address            = 0x%x%x\04x\ n"), *addrintp, *addrshortp);
printf((catgets(catfd,1,110, "Administration Status (value)  = " ));
printf((catgets(catfd,1,adopstatus[mib1p->ifAdmin].nlix, adopstatus[mib1p->ifAdmin].s)), mib1p->ifAdmin);
printf((catgets(catfd,1,111, "Operation Status (value)       = ")), mib1p->ifOper);
printf((catgets(catfd,1,112, "Last Change                     = %u\ n")), mib1p->ifLastChange);
printf((catgets(catfd,1,113, "Inbound Octets                  = %u\ n")), mib1p->ifInOctets);
printf((catgets(catfd,1,114, "Inbound Unicast Packets         = %u\ n")), mib1p->ifInUcastPkts);
printf((catgets(catfd,1,115, "Inbound Non-Unicast Packets     = %u\ n")), mib1p->ifInNUcastPkts);
printf((catgets(catfd,1,116, "Inbound Discards                = %u\ n")), mib1p->ifInDiscards);
printf((catgets(catfd,1,117, "Inbound Errors                  = %u\ n")), mib1p->ifInErrors);
printf((catgets(catfd,1,118, "Inbound Unknown Protocols       = %u\ n")), mib1p->ifInUnknownProtos);
printf((catgets(catfd,1,119, "Outbound Octets                 = %u\ n")), mib1p->ifOutOctets);
```
"Outbound Unicast Packets = %u\ n"),
mib1p->ifOutUcastPkts);
printf((catgets(catfd,1,121,
"Outbound Non-Unicast Packets = %u\ n"),
mib1p->ifOutNUcastPkts);
printf((catgets(catfd,1,122,
"Outbound Discards = %u\ n"), mib1p->ifOutDiscards);
printf((catgets(catfd,1,123,
"Outbound Errors = %u\ n"), mib1p->ifOutErrors);
printf((catgets(catfd,1,124,
"Outbound Queue Length = %u\ n"), mib1p->ifOutQlen);
printf((catgets(catfd,1,125,
"Specific = %u\ n"), mib1p->ifSpecific);

/* Are there Extended MIB statistics to be displayed? */
if (!mib1p->ifSpecific)
    /* No */
    {
        catclose(catfd);
        return;
    }

/* Display "Continue" message and wait for user input */
disp_continue();

mib2p = (int)mib1p + sizeof(mib_ifEntry);
printf((catgets(catfd,1,150,
    "Ethernet-like Statistics Group\ n\ n")),
mib2p->dot3StatsIndex);
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin’s shared library sample code

```c
    printf((catgets(catfd,1,151,
               "Index                           = %u \n")), mib2p->dot3StatsIndex);
    printf((catgets(catfd,1,152,
               "Alignment Errors                = %u \n")), mib2p->dot3StatsAlignmentErrors);
    printf((catgets(catfd,1,153,
               "FCS Errors                      = %u \n")), mib2p->dot3StatsFCSErrors);
    printf((catgets(catfd,1,154,
               "Single Collision Frames         = %u \n")), mib2p->dot3StatsSingleCollisionFrames);
    printf((catgets(catfd,1,155,
               "Multiple Collision Frames       = %u \n")), mib2p->dot3StatsMultipleCollisionFrames);
    printf((catgets(catfd,1,156,
               "Deferred Transmissions          = %u \n")), mib2p->dot3StatsDeferredTransmissions);
    printf((catgets(catfd,1,157,
               "Late Collisions                 = %u \n")), mib2p->dot3StatsLateCollisions);
    printf((catgets(catfd,1,158,
               "Excessive Collisions            = %u \n")), mib2p->dot3StatsExcessiveCollisions);
    printf((catgets(catfd,1,159,
               "Internal MAC Transmit Errors    = %u \n")), mib2p->dot3StatsInternalMacTransmitErrors);
    printf((catgets(catfd,1,160,
               "Carrier Sense Errors            = %u \n")), mib2p->dot3StatsCarrierSenseErrors);
    printf((catgets(catfd,1,161,
               "Frames Too Long                 = %u \n")), mib2p->dot3StatsFrameTooLongs);
```

564  Appendix D
printf((catgets(catfd,1,162,
    "Internal MAC Receive Errors   = %u\n")),
   mib2p->dot3StatsInternalMacReceiveErrors);
   printf((catgets(catfd,1,600,"\n")));

#ifndef UNSUPPORTED /* The enet driver and card don't maintain the
following */
   mib3p = (int)mib2p + sizeof(mib_Dot3StatsEntry); /* JR 3/23/94 */
   printf((catgets(catfd,1,180,
    "\nEthernet-like Collision Statistics Group\n\n")),
   mib2p->dot3StatsIndex);
   printf((catgets(catfd,1,181,
    "Collision Index             = %u\n")), mib3p->dot3CollIndex);
   printf((catgets(catfd,1,182,
    "Collision Count             = %u\n")), mib3p->dot3CollCount);
   printf((catgets(catfd,1,183,
    "Collision Frequency         = %u\n")),
   mib3p->dot3CollFrequencies);
   printf((catgets(catfd,1,600,"\n")));
#endif /* end UNSUPPORTED */

catclose(catfd);

} /* end dsenet() */

disp_continue()
{
  u_char c;
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin’s shared library sample code

```c
fflush(stdout);

printf((catgets(catfd,1,600,"n")));

/* ask for depressing enter key */
printf((catgets(catfd,1,601,"Press <Return> to continue ")));

/* flush the input stream until the first field character */
/* or a field separator ( end of line ) */
while (((c = getchar()) != EOF) && (c != \n));

/* EOF encountered ? */
if (feof(stdin) != FALSE) { /* EOF while reading the file */
printf((catgets(catfd,1,602,
"nAdministration terminated by EOF on input.n")));
catclose(catfd);
exit(0); /* good exit status */
}

printf((catgets(catfd,1,600,"n")));

} /* end disp_continue() */

******************************************************************************
get the next message from a stream
******************************************************************************
```
int get_msg(fd)
int fd;/* file descriptor */
{
  int flags = 0;/* 0 ---> get any available message */

  /*
   * zero first byte of control area so the caller can call check_ctrl
   * without checking the get_msg return value; if there was only data
   * in the message and the user was expecting control or control
   * data then when he calls check_ctrl it will compare the expected
   * primitive zero and print information about the primitive that it
   * got.
   */
  ctrl_area[0] = 0;

  /* call getmsg and check for an error */
  if (getmsg(fd, &ctrl_buf, &data_buf, &flags) < 0) {
    sprintf (errbuf, "%s: getmsg failed, errno = %d\n", driver_name,
              errno);
    fprintf(stderr, (catgets (catfd, 1, errbase + 2, errbuf)), errno);
    return(-1);
  }

  return(1);
} /* end get_msg() */
put a message consisting of only a control part on a stream
***************************************************************************/
int
put_ctrl(fd, length, pri)
intfd;/* file descriptor */
intlength;/* length of control message */
intpri;/* priority of message: either 0 or RS_HIPRI */
{
/* set the len field in the strbuf structure */
ctrl_buf.len = length;

/* call putmsg and check for an error */
if (putmsg(fd, &ctrl_buf, 0, pri) < 0) {
    sprintf (errbuf, "%s: putmsg failed, errno = %d\n", 
        driver_name, errno);
    fprintf(stderr, (catgets (catfd, 1, errbase + 3, errbuf)), errno);
    return(-1);
}
} /* end put_ctrl() */

 רוצה למדחף גבוה? (N) ישראל (Y) נורווגיה (P) אוסטריה (U) איטליה (L) ספרד

***************************************************************************/
check that control message is the expected message
***************************************************************************/
int
check_ctrl(ex_prim)
intex_prim;/* the expected primitive */
{
    dl_error_ack_t*err_ack = (dl_error_ack_t*)ctrl_area;
/* did we get the expected primitive? */
if (err_ack->dl_primitive != ex_prim) {
    /* did we get a control part */
    if (ctrl_buf.len) {
        /* yup; is it an ERROR_ACK? */
        if (err_ack->dl_primitive == DL_ERROR_ACK) {
            /* yup; format the ERROR_ACK info */
            sprintf (errbuf, "%s: expected primitive 0x%02x, got
DL_ERROR_ACK\n",
                driver_name, ex_prim);
            fprintf(stderr, (catgets (catfd, 1, errbase + 4,
                errbuf)),
                ex_prim);
            fprintf(stderr, (catgets(catfd, 1, errbase + 5,
                " dl_error_primitive = 0x%02x\n")),
                err_ack->dl_error_primitive);
            fprintf(stderr, (catgets(catfd, 1, errbase + 6,
                " dl_errno = 0x%02x\n")),
                err_ack->dl_errno);
            fprintf(stderr, (catgets(catfd, 1, errbase + 7,
                " dl_unix_errno = %d\n")),
                err_ack->dl_unix_errno);
            return(-1);
        } else {
            /* didn't get an ERROR_ACK either; print whatever
primitive we did get */
            fprintf(stderr, (catgets(catfd, 1, errbase + 4,
                " dl_primitive = 0x%02x\n")),
                err_ack->dl_primitive);
        }
    } else {
    } else {
*/
'dsenet: expected primitive 0x%02x, got DL_ERROR_ACK\ n*),ex_prim);
return(-1);
}
} else {
/* no control; did we get data */
if (data_buf.len) {
/* tell user we only got data */
sprintf (errbuf, "%s: check_ctrl found only data\ n",
       driver_name) ;
fprintf(stderr, (catgets (catfd, 1, errbase + 8,
                        errbuf))) ;
return(-1);
} else {
/* message???; well, it was probably an
 * interrupted system call */
sprintf (errbuf, "%s: check_ctrl found no message\ n",
       driver_name) ;
fprintf(stderr, (catgets (catfd, 1, errbase + 9,
                        errbuf))) ;
return(-1);
}/* end else */
}/* end else */
}/* end if */
}/* end check_ctrl() */
mib_ifEntry * dsenet_get_statistics(fd)
    int fd;
{
    dl_get_statistics_req_t *get_statistics_req =
        (dl_get_statistics_req_t *) ctrl_area;
    dl_get_statistics_ack_t *get_statistics_ack =
        (dl_get_statistics_ack_t *) ctrl_area;
    mib_ifEntry *mib_ptr;
    int getstats_failed = 0;

    /*
    * Change the statistics on LAN Interface device using DLPI
    * to send the DL_GET_STATISTICS_REQ and wait for the
    * DL_GET_STATISTICS_ACK
    */
    get_statistics_req->dl_primitive = DL_GET_STATISTICS_REQ;
    if (put_ctrl(fd, sizeof(dl_get_statistics_req_t), 0) == -1)
        getstats_failed = TRUE;
    else if (get_msg(fd) == -1)
        getstats_failed = TRUE;
    else if (check_ctrl(DL_GET_STATISTICS_ACK) == -1)
        getstats_failed = TRUE;

    if (getstats_failed) {
        fprintf(stderr, (catgets(fd,2,299,
            "get_stats: unable to get statistics\n")));
    }
exit(-1);
}

mib_ptr = (u_char *) ctrl_area + get_statistics_ack->dl_stat_offset;
return(mib_ptr);
}

struct settingtoken {
    char *name[4];
    int type;
};

struct settingtoken parsetab[] = {
    { "10HD", "10Hd", "10hD", "10hd", 11 },
    { "10FD", "10Fd", "10fD", "10fd", 12 },
    { "100HD", "100Hd", "100hD", "100hd", 13 },
    { "100FD", "100Fd", "100fD", "100fd", 14 },
    { "AUTO_ON", "AUTO_on", "auto_ON", "auto_on", 15 },
    { 0, 0, 0, 0, 0 }, /* terminator */
};

int dsenet_parse(arg)
    char*arg;
{
    char*s;
    int i, j;
/*
 * i: row of parsetab
 * j: column of parsetab
 */
i = 0;
while (parsetab[i].name[0]) {
    for (j = 0; j < 4; j++) {
        if (strcmp(arg, parsetab[i].name[j]) == 0) {
            return(parsetab[i].type);
        }
    }
    i++;
}

return(-1);
}

#define SET_SPEED 994

int dsenet_set_special(fd, ppa, arg)
    int fd;
    int ppa;
    char *arg;
{

Appendix D
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin's shared library sample code

```c
struct strioctl str;
struct fis larg;
char prompt;
int setting_type;

/*
 * Must be super-user
 */
if (getuid() != 0) {
    fprintf(stderr, "Must be super-user\n");
    exit(-1);
}

if ((setting_type = dsenet_parse(arg)) == -1) {
    fprintf(stderr, "enet: ERROR: Setting %s is unsupported\n", arg);
    fprintf(stderr, "Valid types: 10HD, 10FD, 100HD, 100FD, AUTO_ON\n");
    exit(-1);
}

friendly_reminding();

/*
 * fill up driver specific
 */
str.ic_cmd = DLPI_LINK_SPEED;
str.ic_timeout = 1;
str.ic_dp = &larg;
str.ic_len = sizeof(larg);
```

Appendix D
larg.reqtype = SET_SPEED;
larg.vtype = INTEGERTYPE;

/*
 * The following magic number is chose for backward compatibility
 * 11:10HD
 * 12:10FD
 * 13:100HD
 * 14:100FD
 * 15:autoneg_ON
 */
larg.value.i = setting_type;

if (ioctl(fd, I_STR, &str) < 0) {
    fprintf(stderr, "ERROR Setting %s is unsupported\n", arg);
    exit(-1);
}

fprintf(stderr, "Driver is attempting to set the new speed\n\n" );
fprintf(stderr, "Reset will take approximately 11 seconds\n\n" );
return(0);
}

friendly_reminding()
{
    fprintf(stderr, "\n" );
    fprintf(stderr,"WARNING: an incorrect setting could cause ");
    fprintf(stderr,"serious network problems!!!\n\n" );
int dsenet_get_special (fd, ppa, arg, ret)
int fd;
int ppa;
char *arg;
char **ret;
{
    mib_ifEntry * mib_ptr;
    int catfd;
    int duplex = 0; /* HD=0 FD=1 */

    catfd = catopen("dsenet", 0);

    mib_ptr = dsenet_get_statistics(fd);

    /* Use the description field to determine if we are in half or full
     * duplex. The string should be short so we can be inefficient
     */
    {
        int i;
        char *str = mib_ptr->ifDescr;

        for (i = 0; str[i]; i++) {
            if (str[i] == 'F' && str[i+1] == 'u' &&
                str[i+2] == 'l' && str[i+3] == 'l') {
                duplex = 1;
                break;
            }
        }
    }
}
printf((catgets(catfd,2,201,"Current Speed = %u %s\n")), mib_ptr->ifSpeed,(duplex==1)?"Full-Duplex":"Half-Duplex");

catclose (catfd);

return -1; 

}
The driveradmin sample code

/***************************************************************************/
/*
* enetadmin
*
***************************************************************************/

/***************************************************************************/
/** main **/
/****/
/** DESCRIPTION: The main Code file contains the **/
/** Main Command and the Break key functions. **/
/** Main command is the functionthat is given **/
/** control by the shell at Administration **/
/** execution time. Break key is the function **/
/** that processes the INTR signal. **/
/**** /
/**** /
/**** /
***************************************************************************/

#define MODULEID

#if defined(MODULEID) && !defined(lint)
static char rcsid[]="@(#) enetadmin.c 99/05/30";
#endif

#include <stdio.h>

#include <stdio.h>
#include <fcntl.h>
#include <curses.h>
#include <errno.h>
#include <nl_types.h>
#include <locale.h>
#include <term.h>
#include <sio/llio.h>
#include <sys/mib.h>
#include <dl.h>
#include <netio.h>
#include <sys/dlpi.h>
#include <sys/dlpi_ext.h>
#include <sys/stropts.h>
#include "enetadmin.h"

/* Local Macro and constants definition */
#define VERSION "1.0"
#define NATIVE_COMPUTER "C" /* default NLS language */

/* External variables definition */

boolean terse=FALSE, /* terse mode */
echocmd=FALSE, /* = TRUE, echo commands on stdout */
break_hit=FALSE; /* set break_hit on break key depressed */

byte msg_buf_1[MAX_MSG_LENGTH+1]; /* pool of message buffers */
byte msg_buf_2[MAX_MSG_LENGTH+1];
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c
byte msg_buf_3[MAX_MSG_LENGTH+1];

int   termlines;     /* holds the screen line numbers of the*/
/* external variable declaration */

/* external variable declaration */
extern int errno;

/* NLS defines */
nl_catd nls_lanadmin;
char *catgets();
extern char *setlocale();
extern char *getenv();
extern char *admin_msgtable[]; /* table containing enetadmin messages */
extern char *share_msgtable[]; /* table containing day and month */

/* Command line options variables and macros */
extern char *optarg;
extern int optind;
extern int optopt;
extern int opterr;
u_int options = 0;
u_int third_p_options = 0;
#define NO_OPTION 0x000
#define DISPLAY_ADDR 0x001
#define CHANGE_ADDR 0x002
#define DISPLAY_MTUSIZE 0x004
#define CHANGE_MTUSIZE 0x008
```
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

#define RESET_MTUSIZE 0x010
#define DISPLAY_SPEED 0x020
#define CHANGE_SPEED 0x040
#define DOWNLOAD 0x080
#define UNLOAD 0x100
#define DISPLAY_RIF 0x200
#define CHANGE_RIF 0x400
#define SET_DRIVER_SPECIAL 0x800
#define GET_DRIVER_SPECIAL 0x1000

/* for third parties */
#define THIRD_PARTY 0x1200

#define SET_MTU_SIZE 980/* As defined in token1.h */
#define RESET_MTU_SIZE 981/* As defined in token1.h */
#define SET_SPEED 994/* As defined in token1.h */

#define STR_ON "on"
#define STR_OFF "off"
#define TURN_ON 1
#define TURN_OFF 0

char cmdstr[80];
char new_phys_addr[80];
char new_rif[20];
char new_mtusize[20];
char new_speed[30];
char download_file[256];
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c
char driver_special_arg[256];
char tempstr[] = "\0\0\0";
int ppa;
int fd; /* file descriptor */

u_char* get_phys_addr();
mib_ifEntry* get_stats();
void lcs_admin();
void lcs_clear();
void lcs_display();
void lcs_reset();
void admin_screen_size();

**************************************************************************
global areas for sending and receiving streams messages
**************************************************************************
#define AREA_SIZE 25232 /* bytes; big enough for largest possible msg */
#define LONG_AREA_SIZE (AREA_SIZE / sizeof(u_long)) /* AREA_SIZE / 4 */

u_long ctrl_area[LONG_AREA_SIZE]; /* for control messages */
u_long data_area[LONG_AREA_SIZE]; /* for data messages */
u_long ppa_area[LONG_AREA_SIZE]; /* for saving ppa area */

struct strbuf ctrl_buf = {
    AREA_SIZE,
    0,
    ctrl_area
};
```
struct strbuf data_buf = {
    AREA_SIZE,
    0,
    data_area
};

int cur_ppa;

dl_hp_ppa_info_t ppa_info;

#define MAX_PPA_STR_LEN 4
char ppa_str[MAX_PPA_STR_LEN];

#define MAX_DEVICE_FILE_STR_LEN 40
/* third parties path and special device file name */
char third_p_dev_file[MAX_DEVICE_FILE_STR_LEN];

#define TEMP_SIZE 40
char shlib_filename[TEMP_SIZE];
char shlib_funcname[TEMP_SIZE];
shlib_tlib;

nls_start()
{
    char *lang;
    char buf[20];
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c
lang = getenv("LANG");
  do {
    /* Make sure LANG is set to a valid language */
    if (!lang && *lang) {
      lang = NATIVE_COMPUTER;
      sprintf(buf, "LANG=%s", lang);
      putenv(buf);
    }
    /* lang is always set at this point */
    setlocale(LC_ALL, "");
    if(strcmp(lang, NATIVE_COMPUTER) == 0) {
      break;
    } else { /* lang wasn't C */
      fprintf(stderr, "Language %s will used.\n", NATIVE_COMPUTER);
    }
    /* set default so we use C on next pass */
    lang = NULL;
  }
  while (nls_lanadmin < 0);
return(0);
} /* end nls_start() */

void
```
main(argc, argv)
int argc;
char* argv[];
{
    word command; /* command code returned by utils_get_cmd()*/
    char buf[20];
    u_int opt;
    int i;

    /* trap the SIGINT signal ( BREAK key interruption ) */
    if (signal(SIGINT,SIG_IGN) != SIG_IGN) /* set up trap unless already */
        signal(SIGINT, break_key); /* ignoring signals */

    nls_start();

    strcpy(cmdstr, argv[0]);

driver_special_arg[0] = 0;
    optarg = FALSE;
    while ((opt = getopt(argc, argv, ":etaA:bB:mM:RsS:ud:Xx:3:")) != -1) {
        switch (opt) {
        case 'e':
            echocmd = TRUE;
            break;
        case 't':
            terse = TRUE;
            break;
        case 'a':
            break;
        case 'b':
            break;
        case 'm':
            break;
        case 's':
            break;
        case 'u':
            break;
        case 'd':
            break;
        case 'X':
            break;
        case 'x':
            break;
        case '3':
            break;
        default:
            break;
        }
    }
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c
options |= DISPLAY_ADDR;
break;

case 'A':
    options |= CHANGE_ADDR;
    strcpy(new_phys_addr, optarg);
    break;

case 'b':
    options |= DISPLAY_RIF;
    break;

case 'B':
    options |= CHANGE_RIF;
    strcpy(new_rif, optarg);
    break;

case 'm':
    options |= DISPLAY_MTUSIZE;
    break;

case 'M':
    options |= CHANGE_MTUSIZE;
    strcpy(new_mtusize, optarg);
    break;

case 'R':
    options |= RESET_MTUSIZE;
    break;

case 's':
    options |= DISPLAY_SPEED;
    break;

case 'S':
    options |= CHANGE_SPEED;
```
strcpy(new_speed, optarg);
break;

case 'd':
    options |= DOWNLOAD;
    strcpy(download_file, optarg);
    break;

case 'u':
    options |= UNLOAD;
    break;

case 'X':
    options |= SET_DRIVER_SPECIAL;
    for (i=optind; i<argc-1; i++)
    {
        strcat(driver_special_arg, argv[i]);
        if (i < argc-2)
            strcat(driver_special_arg, " ");
    }
    if (optind < argc)
        optind = argc - 1;
    break;

case 'x':
    options |= GET_DRIVER_SPECIAL;
    for (i=optind; i<argc-1; i++)
    {
        strcat(driver_special_arg, argv[i]);
        if (i < argc-2)
            strcat(driver_special_arg, " ");
    }
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c
if (optind < argc)
    optind = argc - 1;
break;
    /* for third parties */
case '3':
    third_p_options = THIRD_PARTY;
    strcpy(third_p_dev_file, optarg);
    break;
case '3':
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5000,
        "\ nOption -%c requires an argument\ n")), optopt);
    usage_error();
    break;
case '?':
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5001,
        "\ nUnrecognized option: -%c\ n")), optopt); /*
    catgets 5001 */
    usage_error();
    break;
default:
    usage_error();
    break;
} /* end switch */
} /* end while */

if (opterr)
    usage_error();
```
if (options) {
    process_options(argc, argv);
    exit(0);
}

/* get the number of lines of the screen of the terminal from which 
 * the diagnostics started*/
admin_screen_size();

fprintf(stderr, "\n\n%s\n\n", utils_get_message(M_WAKE_UP), VERSION);

/* get time in seconds, print time */
utils_d_datetime(stderr);

fprintf(stderr, "%s\n", utils_get_message(M_COPYRIGHT_1));
fprintf(stderr, "%s\n", utils_get_message(M_COPYRIGHT_2));

/* main command loop */

while (TRUE) {
    fprintf(stderr, "\n\n%", utils_get_message(M_TEST_SELECTION_MODE));

    /* displays menu and get operator command */
    command = utils_get_cmd(MAIN_COMMAND_MENU);
    switch(command) {
    case MAIN_LAN_CMD:
        lcs_admin(); /* select LAN Interface status */
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c
break;

case MAIN_MENU_CMD: /* print out menu if terse=true */
    if (terse == TRUE)
        utils_d_menu(MAIN_COMMAND_MENU);
    break;

case MAIN_QUIT_CMD:
    utils_quit_cmd(); /* bye bye! */
    break;

case MAIN_TERSE_CMD:
    terse=TRUE;
    fprintf(stderr, "%s\n",
        utils_get_message(M_DO_NOT_DISPLAY_COMMAND_MENU));
    break;

case MAIN_VERBOSE_CMD:
    terse=FALSE;
    fprintf(stderr, "%s\n", utils_get_message
        (M_DISPLAY_COMMAND_MENU));
    break;

case LCS_THIRD_PARTY_MENU_CMD: /* for third
    parties */
    third_p_options = THIRD_PARTY;
    fprintf(stderr, "%s", "Enter 3rd parties device filename: ");
    lcs_3rd_parties();
    third_p_options = 0;
    break;
```
process_options(argc, argv)
int argc;
char *argv[];
{
  u_int opt_mask;
  int ix;
  mib_ifEntry *mib_ptr;
  u_char *p;
  int len;

  /* Make sure an PPA was also selected. */
  if (optind+1 == argc) {
    /* Convert PPA to an integer */
    ppa = atoi(argv[argc-1]);

    /* check for third party option */
    if (third_p_options == THIRD_PARTY ) {
      /* Open the third party's device file */
      if ((fd = open(third_p_dev_file, O_RDWR)) == -1) {
        fprintf(stderr, "%s\n",
              utils_get_message(L_UNABLE_TO_OPEN_DEVICE_FILE),third_p_dev_file);
        }
exit(-1);

} else /* no third party option */
    /* Open the DLPI device file, /dev/dlpi. */
    if ((fd = open("/dev/dlpi", O_RDWR)) == -1) {
        fprintf(stderr, "%s%s
",
                utils_get_message(L_UNABLE_TO_OPEN_DEVICE_FILE),
                " /dev/dlpi");
        exit(-1);
    }
}

/* Get the list of PPAs */
if (get_ppa_list() == -1)
    exit(-1);

/* Attach to the new ppa (ppa) */
if (attach_ppa(ppa) == -1)
    exit(-1);
cur_ppa = ppa;
} else {
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5002, "nError -
                      must enter a PPA Number\n"))); /* catgets 5002 */
    usage_error();
}

for (opt_mask = 1; options; opt_mask <<= 1) {
    switch (options & opt_mask) {
    case DISPLAY_ADDR:
        p = get_phys_addr(&len, DL_CURR_PHYS_ADDR);
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

display_phys_addr(p, len);
break;
case CHANGE_ADDR:
  change_phys_addr();
  break;
case DISPLAY_RIF:
  display_rif_flag ();
  break;
case CHANGE_RIF:
  change_rif_flag ();
  break;
case DISPLAY_MTUSIZE:
  mib_ptr = get_stats();
  printf((catgets(nls_lanadmin,NL_SETN,6000,
      "MTU Size = %u\n")), /* catgets 6000 */
      mib_ptr->ifMtu);
  break;
case CHANGE_MTUSIZE:
  change_mtusize();
  break;
case RESET_MTUSIZE:
  reset_mtusize();
  break;
case DISPLAY_SPEED:
  mib_ptr = get_stats();
  printf((catgets(nls_lanadmin,NL_SETN,6001,
      "Speed = %u\n")), /* catgets 6001 */
      mib_ptr->ifSpeed);

Appendix D 593
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c
break;
case CHANGE_SPEED:
    change_speed();
    break;
case DOWNLOAD:
    download(1);
    break;
case UNLOAD:
    download(0);
    break;
case SET_DRIVER_SPECIAL:
    set_driver_special();
    break;
case GET_DRIVER_SPECIAL:
    get_driver_special();
    break;
default:
    break;
} /* end switch */

/* Reset the option bit we just processed. */
options &= ~opt_mask;
    } /* end for */
} /* end process_options */

/******************************/

mib_ifEntry *
```
get_stats()
{
dl_get_statistics_req_t *get_statistics_req = (dl_get_statistics_req_t *)
ctrl_area;
dl_get_statistics_ack_t *get_statistics_ack = (dl_get_statistics_ack_t *)
ctrl_area;
mib_ifEntry*mib_ptr;
int getstats_failed = 0;

/* Change the statistics on LAN Interface device using DLPI
 * to send the DL_GET_STATISTICS_REQ and wait for the
 * DL_GET_STATISTICS_ACK */
get_statistics_req->dl_primitive = DL_GET_STATISTICS_REQ;
if (put_ctrl(fd, sizeof(dl_get_statistics_req_t), 0) == -1)
getstats_failed = TRUE;
else if (get_msg(fd) == -1)
getstats_failed = TRUE;
else if (check_ctrl(DL_GET_STATISTICS_ACK) == -1)
getstats_failed = TRUE;
if (getstats_failed) {
fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5003,
"get_stats: unable to get statistics\ n"))); /* catgets 5003 */
exit(-1);
}

mib_ptr = (u_char *) ctrl_area + get_statistics_ack->dl_stat_offset;
return(mib_ptr);

} /* end get_stats() */
**driveradmin sample code**

```c
u_char *
get_phys_addr(len, type)
int *len;
int type;
{
    dl_phys_addr_req_t *phys_addr_req = (dl_phys_addr_req_t *) ctrl_area;
    dl_phys_addr_ack_t *phys_addr_ack = (dl_phys_addr_ack_t *) ctrl_area;
    int physaddr_failed = 0;

    /* Get the physical address on LAN Interface device using DLPI */
    * to send the DL_PHYS_ADDR_REQ and wait for the
    DL_PHYS_ADDR_ACK */
    phys_addr_req->dl_primitive = DL_PHYS_ADDR_REQ;
    phys_addr_req->dl_addr_type = type;
    if (put_ctrl(fd, sizeof(dl_phys_addr_req_t), 0) == -1)
        physaddr_failed = TRUE;
    else if (get_msg(fd) == -1)
        physaddr_failed = TRUE;
    else if (check_ctrl(DL_PHYS_ADDR_ACK) == -1)
        physaddr_failed = TRUE;

    if (physaddr_failed) {
        fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5004,
            "get_phys_addr: unable to get physical address\n")));
        exit(-1);
    }
    *len = phys_addr_ack->dl_addr_length;
}
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c
return((char *) ctrl_area + phys_addr_ack->dl_addr_offset);
} /* end get_phys_addr() */

/*------------------------*/
display_phys_addr(p, len)
  u_char*p;
  int len;
  {
    intix;

    printf((catgets(nls_lanadmin,NL_SETN,6002,"Station Address =
            0x")));
    for (ix = 0; ix <len; ix++)
      printf((catgets(nls_lanadmin,NL_SETN,6003,"%02x")), *(p+ix));
    printf((catgets(nls_lanadmin,NL_SETN,6004,"\n")));
  } /* end display_phys_addr() */

/*------------------------*/

change_phys_addr()
{
  dl_set_phys_addr_req_t *set_phys_addr_req = (dl_set_phys_addr_req_t *) ctrl_area;
  int  setphysaddr_failed = 0;
  u_char  *p;
  u_char  phaddr[40];
```
int len;
int ix;
int (*reset)();
int *libptr, flag=0;

/* Must be super-user */
if (getuid() != 0) {
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5005, 
"Must be super-user\n"))); /* catgets 5005 */
    exit(-1);
}

/* Get and display the current physical address */
p = get_phys_addr(&len, DL_CURR_PHYS_ADDR);
memcpy(phaddr, p, len);
printf((catgets(nls_lanadmin,NL_SETN,6005, "Old ")));
display_phys_addr(phaddr, len);

/* Convert the new physical address to binary */
if (strncmp(new_phys_addr, "DEFAULT", 7) == 0) {
    p = get_phys_addr(&len, DL_FACT_PHYS_ADDR);
    memcpy(phaddr, p, len);
} else {
    net_aton(phaddr, new_phys_addr, len);
}
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

/* Change the physical address on LAN Interface device using DLPI */
set_phys_addr_req->dl_primitive = DL_SET_PHYS_ADDR_REQ;
set_phys_addr_req->dl_addr_length = len;
set_phys_addr_req->dl_addr_offset = sizeof(dl_set_phys_addr_req_t);
memcpy((char *)ctrl_area+set_phys_addr_req->dl_addr_offset, phaddr, len);
if (put_ctrl(fd, sizeof(dl_set_phys_addr_req_t)+len, 0) == -1)
setphysaddr_failed = TRUE;
else if (get_msg(fd) == -1)
setphysaddr_failed = TRUE;
else if (check_ctrl(DL_OK_ACK) == -1)
setphysaddr_failed = TRUE;

if (setphysaddr_failed) {
fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5006,
"change_phys_addr: unable to change physical address\n")));
exit(-1);
}

/* Some device interface requires reset here, check it for the device specific reset function, if any! */
sprintf(shlib_filename, "/usr/lib/lanadmin/libds%s.sl", ppa_info.dl_name);
/* Load the shared library */
if ((lib = shl_load(shlib_filename, BIND_IMMEDIATE, 0)) == 0) {

Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c
fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5100,
        "Cannot open shared library %s\n")), shlib_filename);
    return;
}
/* Change of MAC address, needs reset/download to be performed on
the card */
    sprintf(shlib_funcname, "ds%s_reset", ppa_info.dl_name);

/* Get the address of the shared library function, if exists */
    libptr =&lib;
    if (shl_findsym(libptr, shlib_funcname, TYPE_PROCEDURE, &reset)
!= -1) {
        /* Reset the device/interface */
        (*(reset))(fd, cur_ppa, flag);
        shl_unload(lib);
    }

/* Display the new physical address */
    printf((catgets(nls_lanadmin,NL_SETN,6006, "New "));
    display_phys_addr(phaddr, len);
} /* end change_phys_addr() */

change_mtusize()
{
    struct strioctl strioctl;
    struct fis arg;
```
mib_ifEntry*mib_ptr;

    /* Must be super-user */
    if (getuid() != 0) {
        fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5005,
            "Must be super-user\n"))); /* catgets 5005 */
        exit(-1);
    }

    printf((catgets(nls_lanadmin,NL_SETN,6005, "Old ")), /* catgets 6005 */
            mib_ptr = get_stats();
    printf((catgets(nls_lanadmin,NL_SETN,6000, 
        "MTU Size = %u\n")), /* catgets 6000 */
            mib_ptr->ifMtu);

    strioctl.ic_cmd = DLPI_MTU_BYTE_SIZE;
    strioctl.ic_timeout = 0;
    strioctl.ic_len = sizeof(arg);
    strioctl.ic_dp = &arg;
    arg.reqtype = SET_MTU_SIZE;
    arg.vtype = INTEGERTYPE;
    arg.value.i = atoi(new Mtusize);
    if (ioctl(fd, I_STR, &strioctl) < 0) {
        fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5008,
            "Unable to change MTU size: errno = %d\n")), errno);
        /* catgets 5008 */
        return;
    }
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c
printf((catgets(nls_lanadmin,NL_SETN,6006, "New ")));
  mib_ptr = get_stats();
  printf((catgets(nls_lanadmin,NL_SETN,6000, 
    "MTU Size = %u\n")), /* catgets 6000 */
    mib_ptr->ifMtu);
} /* end change_mtusize() */

reset_mtusize()
{
  struct strioctl strioctl;
  struct fis arg;
  mib_ifEntry*mib_ptr;

  /* Must be super-user */
  if (getuid() != 0) {
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5005, 
      "Must be super-user\n"))); /* catgets 5005 */
    exit(-1);
  }

  printf((catgets(nls_lanadmin,NL_SETN,6005, "Old ")));
  mib_ptr = get_stats();
  printf((catgets(nls_lanadmin,NL_SETN,6000, 
    "MTU Size = %u\n")), /* catgets 6000 */
    mib_ptr->ifMtu);
```
```c
strioctl.ic_cmd = DLPI_MTU_BYTE_SIZE;
strioctl.ic_timeout = 0;
strioctl.ic_len = sizeof(arg);
strioctl.ic_dp = &arg;
arg.reqtype = RESET_MTU_SIZE;
arg.vtype = INTEGERTYPE;
if (ioctl(fd, I_STR, &strioctl) < 0) {
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5008,
        "Unable to change MTU size: errno = %d\n")), errno);
    /* catgets 5008 */
    return;
}
    printf((catgets(nls_lanadmin,NL_SETN,6006, "New "));
    mib_ptr = get_stats();
    printf((catgets(nls_lanadmin,NL_SETN,6000,
        "MTU Size                        = %u\n")), /* catgets 6000 */
        mib_ptr->ifMtu);
} /* end change_mtusize() */

void
change_speed()
{
    struct strioctl strioctl;
    struct fis arg;
    mib_ifEntry*mib_ptr;

    /* Must be super-user */
    if (getuid() != 0) {
```
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c
fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5005, "Must be super-user\ n")));
exit(-1);
}

printf((catgets(nls_lanadmin,NL_SETN,6005, "Old ")));
mib_ptr = get_stats();
printf((catgets(nls_lanadmin,NL_SETN,6001, "Speed                           = %u\ n")), /* catgets 6001 */
mib_ptr->ifSpeed);

strioctl.ic_cmd = DLPI_LINK_SPEED;
strioctl.ic_timeout = 0;
strioctl.ic_len = sizeof(arg);
strioctl.ic_dp = &arg;
arg.reqtype = SET_SPEED;
arg.vtype = INTEGER TYPE;
arg.value.i = atoi(new_speed);
if (ioctl(fd, I_STR, &strioctl) < 0) {
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5007, "Unable to change speed: errno = %d\ n")), errno);
    /* catgets 5007 */
    return;
}

printf((catgets(nls_lanadmin,NL_SETN,6006, "New ") ));
mib_ptr = get_stats();
printf((catgets(nls_lanadmin,NL_SETN,6001, "Speed                           = %u\ n")), /* catgets 6001 */
mib_ptr->ifSpeed);
```

Appendix D
"Speed = %u\n")", /* catgets 6001 */
mib_ptr->ifSpeed);
} /* end change_speed() */
/
******************************************************************/

download(loadflag)
int loadflag;
{
int(*dload)();
shl_t*libptr;
intfn_found = 0;

  /* Must be super-user */
  if (getuid() != 0) {
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5005,
                     "Must be super-user\n")));
    exit(-1);
  }

  /* The name of the shared library that contains the function to
    * do the firmware download associated with this device is:
    *   "/usr/lib/lanadmin/libdl<driver_name>.sl"
  OR "/usr/lib/lanadmin/libds<driver_name>.sl" */

  sprintf(shlib_filename, "/usr/lib/lanadmin/libdl%s.sl",
          ppa_info.dl_name);

  /* Load the shared library */
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c
if ((lib = shl_load(shlib_filename, BIND_IMMEDIATE, 0)) != -1) {

    /* The name of the shared library function to do the firmware
download
    * associated with this device is:
    *     "dl<driver_name>" */
sprintf(shlib_funcname, "dl%s", ppa_info.dl_name);

    /* Get the address of the shared library function */
    libptr = &lib;
    if (shl_findsym(libptr, shlib_funcname, TYPE_PROCEDURE, &dload) < 0)
        shl_unload(lib);
    else      fn_found = 1;
}

if (!fn_found) {
    sprintf(shlib_filename, "/usr/lib/lanadmin/libds%s.sl", ppa_info.dl_name);
    /* Load the alternate shared library */
    if ((lib = shl_load(shlib_filename, BIND_IMMEDIATE, 0)) == 0) {
        fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5100,
            "Cannot open shared library %s\n")), shlib_filename);
        detach_ppa();
        return;
    }
    sprintf(shlib_funcname, "ds%s_dnld", ppa_info.dl_name);

    /* The name of the shared library function to do the firmware
download
    * associated with this device is:
    *     "dn<driver_name>" */
sprintf(shlib_funcname, "dn%s", ppa_info.dl_name);

    /* Get the address of the shared library function */
    libptr = &lib;
    if (shl_findsym(libptr, shlib_funcname, TYPE_PROCEDURE, &dload) < 0)
        shl_unload(lib);
    else      fn_found = 1;
}
```
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

/* Get the address of the shared library function */
libptr = &lib;
if (shl_findsym(libptr, shlib_funcname, TYPE_PROCEDURE, 
&dload) < 0) {
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5101, 
    "Cannot find function %s in shared library %s\n")), 
    shlib_funcname, shlib_filename);
    detach_ppa();
    exit(-1);
}

/* Load/unload firmware for this interface */
(*(dload))(loadflag, fd, download_file);

/* Unload the shared library after we're done with it. */
shl_unload(lib);
} /* end download() */

******************************************************************************

void
break_key()
{
    signal(SIGINT,break_key);
    break_hit=TRUE;
} /* end break_key() */

******************************************************************************
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

usage_error()
{
    fprintf(stderr,(catgets(nls_lanadmin,NL_SETN,6004, "\ n")));  
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,4000,  
    "Usage: %s [-a] [-A station_addr| n"
)), cmdstr);  
    /* catgets 4000 */ 
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,4004,  
        " [-b] [-B on| off| n"
)), cmdstr);  
    /* catgets 4004 */ 
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,4001,  
        " [-m] [-M mtu_size| n"
)), cmdstr);  
    /* catgets 4001 */ 
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,4002,  
        " [-R| n"
)), cmdstr);  
    /* catgets 4002 */ 
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,4003,  
        " [-s] [-S speed| n"
)), cmdstr);  
    /* catgets 4003 */ 
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,4005,  
        " PPA\ n"
)), cmdstr);  
    /* catgets 4005 */ 
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,4006,  
        " [-e\ n"
)), cmdstr);  
    /* catgets 4006 */ 
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,4007,  
        " options| -X options| n"
)), cmdstr);  
    /* catgets 4006 */ 
}
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```
/* catgets 4007 */
fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,4008,
"Default: %s\n")), cmdstr);
/* catgets 4008 */
fprintf(stderr,(catgets(nls_lanadmin,NL_SETN,6004,"
\n")))
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,4009,
"a Display current station address corresponding to
PPA Number\n"))); /* catgets 4009 */
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,4010,
"A Set new station address corresponding to PPA
Number\n"))); /* catgets 4010 */
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,4011,
"station_addr New station address in hex with '0x'
prefix\n"))); /* catgets 4011 */
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,4022,
"b Display source routing flag corresponding to PPA
Number\n"))); /* catgets 4022 */
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,4023,
"B Set source routing flag corresponding to PPA
Number\n"))); /* catgets 4023 */
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,4024,
"on|off Enable or disable 802.5 source routing
protocol\n"))); /* catgets 4024 */
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,4012,
"m Display current MTU size corresponding to PPA
Number\n"))); /* catgets 4012 */
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,4013,
"M Set new MTU size corresponding to PPA
Number\n"))); /* catgets 4013 */
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,4014,
```

Appendix D 609
Shared Library Examples for the driveradmin and lanscan Commands

**The driveradmin sample code**

```
"mtu_size      New MTU size in bytes:\n")); /* catgets 4014 */
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,4015,
    "R         Reset MTU size corresponding to PPA Number to
default.\n"))); /* catgets 4015 */
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,4016,
    "s            Display current speed setting corresponding to PPA
Number.\n")); /* catgets 4016 */
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,4017,
    "S            Set new speed setting corresponding to PPA
Number.\n")); /* catgets 4017 */
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,4018,
    "speed      New speed setting in Mbits/second.\n")); /* catgets 4018 */
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,4026,
    "x            Display driver specific options corresponding to Net
Mgmt ID.\n")));
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,4027,
    "X            Set driver specific options corresponding to Net
Mgmt ID.\n")));
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,6004, "\n")));
    exit(-1);
} /* end usage_error() */
```
lcs_get_dev_file_name()
{
    word len;             /* device name length */
    int temp_fd;          /* temp file descriptor */

    break_hit = FALSE;

    /* loop on the device name input until the device name is ok */
    while (TRUE) {

        /* get the device file name include path (absolute or relative) */
        len = utils_get_field(MAX_DEVICE_FILE_STR_LEN+1,
                               third_p_dev_file);

        if ((break_hit == TRUE) || (len == 0)) {
            if (break_hit == TRUE) {
                fprintf(stderr, "Please enter device special file name
                        Enter 3rd parties' device filename : ");
                continue;
            }

            if ((temp_fd = open(third_p_dev_file, O_RDWR)) == -1) {

                /* See if we can open this special device name */
                if ((temp_fd = open(third_p_dev_file, O_RDWR)) == -1) {

                    fprintf(stderr, "Please enter device special file name
                        Enter 3rd parties' device filename : ");
                    continue;
                }
            }
        }
    }
}
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c
fprintf(stderr, "Invalid device file name. Please check the path.\n")
fprintf(stderr, "Enter 3rd parties' device filename: ");
    continue;
}

/* OK then we close the file */
close(temp_fd);

return;
} /* end while */
} /* end lcs_get_dev_file_name */

lcs_3rd_parties()
{
    word command;

    /* get the device file name for third parties' interface from user */
lcs_get_dev_file_name();

    /* Open the third parties device file. */
    if ((fd = open(third_p_dev_file, O_RDWR)) == -1) {
        fprintf(stderr, "%s\n",
            utils_get_message(L_UNABLE_TO_OPEN_DEVICE_FILE),
            third_p_dev_file);
        return;
    }
```
/* Find the first PPA. */
if (get_ppa_list() == -1)
    return;

/* loop on LAN Interface commands */
while (TRUE) {
    fprintf(stderr, "%s%d
,
    utils_get_message(L_LAN_CARD_TEST_MODE),
    cur_ppa);

    /* display menu and get operator command */
    command = utils_get_cmd(LAN_CARD_TEST_MENU);
    switch(command) {
    case LCS_CLEAR_CMD: /* clears the LAN Interface statistics */
        lcs_clear();
        break;
    case LCS_DISPLAY_CMD: /* displays the LAN Interface info */
        lcs_display();
        break;
    case LCS_END_CMD: /* exit the current mode */
        fprintf(stderr, "%s\n",
            utils_get_message(L_END_OF_LAN_CARD_TEST_MODE));
        return; /* return to the main command loop */
    case LCS_MENU_CMD: /* display the menu if terse = true */
        if (terse == TRUE)
            utils_d_menu(LAN_CARD_TEST_MENU);
        break;
    case LCS_NMID_CMD: /* get the LAN Interface PPA */
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c
lcs_get_nm_id();
break;

case LCS_QUIT_CMD: /* exit the Diagnostic program */
    utils_quit_cmd();
break;

case LCS_RESET_CMD: /* reset the LAN Interface */
    lcs_reset();
    break;

case LCS_DRIVER_MENU_CMD: /* driver specific menu */
    /* menu_driver_special();*/
    fprintf(stderr, "This option is not supported by driver at this time
\n");
    break;

} /* end switch */
} /* end while */
} /* end lcs_3rd_parties() */

/******************************************************************/

tvoid
lcs_admin()
{
    word command;/* returned by utils_get_cmd() */

    /* Open the DLPI device file, /dev/dlpi. */
    if ((fd = open("/dev/dlpi", O_RDWR)) == -1) {
        fprintf(stderr, "%s\n", 
        utils_get_message(L_UNABLE_TO_OPEN_DEVICE_FILE), "
/dev/dlpi");

```
return;
}

/* Find the first PPA. */
if (get_ppa_list() == -1)
return;

/* loop on LAN Interface commands */
while (TRUE) {
    fprintf(stderr, "\n%as%d\n", 
    utils_get_message(L_LAN_CARD_TEST_MODE), cur_ppa);

/* display menu and get operator command */
    command = utils_get_cmd(LAN_CARD_TEST_MENU);
    switch(command) {
    case LCS_CLEAR_CMD: /* clears the LAN Interface statistics */
        lcs_clear();
        break;
    case LCS_DISPLAY_CMD: /* displays the LAN Interface info */
        lcs_display();
        break;
    case LCS_END_CMD: /* exit the current mode */
        fprintf(stderr, "%s\n", 
        utils_get_message(L_END_OF_LAN_CARD_TEST_MODE));
        return; /* return to the main command loop */
    case LCS_MENU_CMD: /* display the menu if terse = true */
        if (terse == TRUE)
            utils_d_menu(LAN_CARD_TEST_MENU);
        break;
    }
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

case LCS_NMID_CMD: /* get the LAN Interface PPA */
    lcs_get_nm_id();
    break;
case LCS_QUIT_CMD: /* exit the Diagnostic program */
    utils_quit_cmd();
    break;
case LCS_RESET_CMD: /* reset the LAN Interface */
    lcs_reset();
    break;
case LCS_DRIVER_MENU_CMD: /* driver specific menu */
    menu_driver_special();
    break;
} /* end switch */
} /* end while */
} /* end lcs_admin() */

/*****************************/

void
lcs_clear()
{

dl_hp_reset_stats_req_t*reset_stats_req
=(dl_hp_reset_stats_req_t*)ctrl_area;
int reset_failed = FALSE;

  /* user allowed to reset the LAN Interface? */
  if (getuid() != 0) {
    fprintf(stderr, "%s\n",
        utils_get_message(L_NOT_AUTHORIZED_TO_CLEAR_STATISTICS));
}
return;

} /* su; go ahead */
/* attach to the current ppa and make sure it is valid */
if (attach_ppa(cur_ppa) != -1) {
    fprintf(stderr, "%s\n",
        utils_get_message(L_CLEARING_LAN_CARD_STATISTICS_REGISTERS));

    /* Clear statistics on LAN Interface device using DLPI */
    /* send the DL_HP_RESET_STATS_REQ and wait for the OK_ACK */
    reset_stats_req->dl_primitive = DL_HP_RESET_STATS_REQ;
    if (put_ctrl(fd, sizeof(dl_hp_reset_stats_req_t), 0) == -1) 
        reset_failed = TRUE;
    else if (get_msg(fd) == -1) 
        reset_failed = TRUE;
    else if (check_ctrl(DL_OK_ACK) == -1) 
        reset_failed = TRUE;

    if (reset_failed)
        fprintf(stderr, "%s\n",
            utils_get_message(L_UNABLE_TO_CLEAR_STATISTICS_REGISTERS));
    detach_ppa();
} else
    fprintf(stderr, "%s %d\n",
        utils_get_message(L_UNABLE_TO_ACCESS_NMID), cur_ppa);
} /* end lcs_clear() */
void
lcs_display()
{

dword line_count;/* used to count the number of lines displayed on the screen*/

inthw_status;

int(*dispstats)();

shl_t*libptr;

printf("\n%5s\n", utils_get_message(L_LAN_CARD_STATUS_DISPLAY));

/* get and print time */

utils_d_datime(stdout);

/* display the Card Instance Number */

printf("%5s\n", utils_get_message(L_NM_ID), cur_ppa);

/* attach to the current ppa and make sure it is valid */

if (attach_ppa(cur_ppa) != -1) {
/* The name of the shared library that contains the function to display the statistics associated with this device is:
 * /usr/lib/lanadmin/libds<driver_name>.sl */

sprintf(shlib_filename, "/usr/lib/lanadmin/libds%s.sl", ppa_info.dl_name);

...
/* Load the shared library */
if ((lib = shl_load(shlib_filename, BIND_IMMEDIATE, 0)) == 0) {
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5100,
        "Cannot open shared library %s\n")), shlib_filename);
    /* catgets 5100 */
    detach_ppa();
    return;
}

/* The name of the shared library function to display the statistics
* associated with this device is:
*    "ds<driver_name>" */
sprintf(shlib_funcname, "ds%s", ppa_info.dl_name);

/* Get the address of the shared library function */
libptr =&lib;
if (shl_findsym(libptr, shlib_funcname,TYPE_PROCEDURE,&dispstats)<0){
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5101,
        "Cannot find function %s in shared library %s\n")),
    /* catgets 5101 */
    shlib_funcname, shlib_filename);
    detach_ppa();
    return;
}

/* Display the statistics for this interface */
(*(dispstats))(fd, cur_ppa, termlines);
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

/* Unload the shared library after we're done with it. */
shl_unload(lib);

detach_ppa();

} else
fprintf(stderr, "%s %d
",
utils_get_message(L_UNABLE_TO_ACCESS_NMID), cur_ppa);
fflush(stdout);
} /* end lcs_display() */

/******************************************************************/

lcs_get_nm_id()
{
int temp_ppa;
word len;

break_hit = FALSE;

/ * loop on the device name input until the device name is ok */
while (TRUE) {
/* ask for a new Card Instance Number */
fprintf(stderr, "%s%d: ",
utils_get_message(L_ENTER_NM_ID_CURRENTLY), cur_ppa);

/* get the new Card Instance Number */
len = utils_get_field(MAX_PPA_STR_LEN+1, ppa_str);

if ((break_hit == TRUE) || (len == 0)) {/* keep old PPA */
if (break_hit == TRUE)
fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,6004, "\n")));
return;
}
/* Convert the string Card Instance Number string to an integer */
temp_ppa = atoi(ppa_str);
/* See if we can attach to this new ppa */
if (attach_ppa(temp_ppa) == -1) {
    fprintf(stderr, "%s", utils_get_message(L_INVALID_NMID_ENTRY));
    continue;
}
/* Detach from the new ppa and return */
detach_ppa();
cur_ppa = temp_ppa;
return;
} /* end while */
}/* end lcs_get_nm_id() */

void
lcs_reset()
{
    dl_hp_hw_reset_req_t*hw_reset_req = (dl_hp_hw_reset_req_t*)ctrl_area;
}
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c
int reset_failed = FALSE;
int(*reset)();
int*libptr, flag=1;

/* user allowed to reset the LAN Interface? */
if (getuid() != 0) {
    fprintf(stderr, "%s
",
        utils_get_message(L_NOT_ALLOWED_TO_RESET_LAN_CARD));
    return;
}
/* su; go ahead*/
/* attach to the current ppa and make sure it is valid*/
if (attach_ppa(cur_ppa) != -1) {
    fprintf(stderr, "%s
",
        utils_get_message(L_RESETTING_LAN_CARD_TO_RUN_SELFTEST)) ;

    /* Check for the device specific reset function, if any ! */
    sprintf(shlib_filename, "/usr/lib/lanadmin/libds%s.sl",
        ppa_info.dl_name);
    /* Load the shared library */
    if ((lib = shl_load(shlib_filename, BIND_IMMEDIATE, 0)) == 0) {
        fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5100,
            "Cannot open shared library %s\n")), shlib_filename);
        return;
    }
    /* Call the corresponding device reset, if any ! */
    sprintf(shlib_funcname, "ds%s_reset", ppa_info.dl_name);
```
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c
/* Get the address of the shared library function */
libptr = &lib;
if (shl_findsym(libptr, shlib_funcname, TYPE_PROCEDURE, &reset) != -1) {
    /* Load firmware for this interface */
    (*reset)(fd, cur_ppa, flag);
    shl_unload(lib);
    detach_ppa();
    return;
}

/* reset LAN Interface using DLPI */
/* send the DL_HP_HW_RESET_REQ and wait for the OK_ACK */
hw_reset_req->dl_primitive = DL_HP_HW_RESET_REQ;
if (put_ctrl(fd, sizeof(dl_hp_hw_reset_req_t), 0) == -1)
    reset_failed = TRUE;
else if (get_msg(fd) == -1)
    reset_failed = TRUE;
else if (check_ctrl(DL_OK_ACK) == -1)
    reset_failed = TRUE;
if (reset_failed)
    fprintf(stderr, "%s\n", utils_get_message(L_UNABLE_TO_RESET_LAN_CARD));
detach_ppa();
} else
    fprintf(stderr, "%s %d\n", utils_get_message(L_UNABLE_TO_ACCESS_NMID), cur_ppa);
} /* end lcs_reset() */
```
/******************************************************************
get the next message from a stream
*******************************************************************/

int get_msg(fd)
int fd;/* file descriptor */
{
    int flags = 0;/* 0 ---> get any available message */

    /* zero first byte of control area so the caller can call check_ctrl
     * without checking the get_msg return value; if there was only data
     * in the message and the user was expecting control or control + data,
     * then when he calls check_ctrl it will compare the expected primitive
     * zero and print information about the primitive that it got.
     */
    ctrl_area[0] = 0;

    /* call getmsg and check for an error */
    if (getmsg(fd, &ctrl_buf, &data_buf, &flags) < 0) {
        fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5102,
                "error: getmsg failed, errno = %d\n")), errno);
        /* catgets 5102 */
        return(-1);
    }
The driveradmin sample code

return(1);
} /* end get_msg() */

/********************************************************************************
check that control message is the expected message
********************************************************************************/
int
check_ctrl(ex_prim)
intex_prim;/* the expected primitive */
{
dl_error_ack_t *err_ack = (dl_error_ack_t *)ctrl_area;

    /* did we get the expected primitive? */
    if (err_ack->dl_primitive != ex_prim) {
    /* did we get a control part */
    if (ctrl_buf.len) {
    /* yup; is it an ERROR_ACK? */
    if (err_ack->dl_primitive == DL_ERROR_ACK) {
    /* yup; format the ERROR_ACK info */
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5106,
    "error: expected primitive 0x%02x, ")),
    ex_prim);
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5107,
    "got DL_ERROR_ACK\ n"))); /* catgets 5107 */
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5108,
    " dl_error_primitive=0x%02x\ n")), err_ack->dl_error_primitive);
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5109,
    " dl_errno=0x%02x\ n")), /* catgets 5109 */
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c
err_ack->dl_errno);
fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5110,
" dl_unix_errno = %d\n")), /* catgets 5110 */
err_ack->dl_unix_errno);
return(-1);
} else {
  /*
   didn't get an ERROR_ACK either; print whatever
   primitive we did get
  */
  fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5106,
"error:  expected primitive 0x%02x, "), ex_prim);
  fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5111,
"got primitive 0x%02x\n")), /* catgets 5111 */
err_ack->dl_primitive);
return(-1);
} else {
  /* no control; did we get data? */
  if (data_buf.len) {
    /* tell user we only got data */
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5112,
"error:  check_ctrl found only data\n")));
    return(-1);
  } else {
    /* no message???; well, it was probably an
     * interrupted system call */
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5113,
"error:  interrupted system call\n")));
    return(-1);
  }
}
```

Appendix D
"error:  check_ctrl found no message\n"));
return(-1);
} /* end else */
} /* end else */
} /* end if */
} /* end check_ctrl() */

/******************************************************************
put a message consisting of only a control part on a stream
******************************************************************/

int
put_ctrl(fd, length, pri)
intfd;
intlength;
intpri;
{
    /* set the len field in the strbuf structure */
    ctrl_buf.len = length;

    /* call putmsg and check for an error */
    if (putmsg(fd, &ctrl_buf, 0, pri) < 0) {
        fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5104,
            "error:  put_ctrl putmsg failed, errno =%d\n")), errno);
        return(-1);
    }
} /* end put_ctrl() */
Get the list of available PPAs and set cur_ppa to the first one.

```
int get_ppa_list()
{
    dl_hp_ppa_req_t *ppa_req = (dl_attach_req_t *)ctrl_area;
    dl_hp_ppa_ack_t *ppa_ack = (dl_hp_ppa_ack_t *)ctrl_area;
    dl_hp_ppa_info_t *ppa_info_temp;

    /* find a PPA to attach to; we assume that the first PPA on the
     * remote is on the same media as the first local PPA
     */

    /* send a PPA_REQ and wait for the PPA_ACK */
    ppa_req->dl_primitive = DL_HP_PPA_REQ;
    if (put_ctrl(fd, sizeof(dl_hp_ppa_req_t), 0) == -1)
        return(-1);
    if (get_msg(fd) == -1)
        return(-1);
    if (check_ctrl(DL_HP_PPA_ACK) == -1)
        return(-1);
    /* make sure we found at least one PPA */
    if (ppa_ack->dl_length == 0) {
        fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5114,
            "error: no PPAs available\n")); /* catgets 5114 */
    } else {
        cur_ppa = ppa_ack;
        ppa_info_temp = ppa_ack;
        ppa_info_temp->dl_primitive = DL_HP_PPA_INFO;
        if (put_ctrl(fd, sizeof(dl_hp_ppa_info_t), 0) == -1)
            return(-1);
        if (get_msg(fd) == -1)
            return(-1);
        if (check_ctrl(DL_HP_PPA_INFO) == -1)
            return(-1);
    }
```
/* Save all the PPA information. */
memcpy((u_char *)ppa_area, (u_char *)ctrl_area+ppa_ack->dl_offset,
ppa_ack->dl_length);

ppa_count = ppa_ack->dl_count;

/* examine the first PPA */
ppa_info_temp = (dl_hp_ppa_info_t *) ppa_area;
cur_ppa = ppa_info_temp->dl_ppa;

return(0);
} /* end get_ppa_list() */

/******************************************************************
attach to the current PPA
******************************************************************/
int
attach_ppa(ppa)
intppa;
{
    dl_attach_req_t*attach_req = (dl_attach_req_t *)ctrl_area;
dl_hp_ppa_info_t *ppa_info_temp;
    int count;
    int found;

    /* See if this ppa is even in list */
    for (count = found = 0, ppa_info_temp = ppa_area; count < ppa_count;
count++, ppa_info_temp++) {


Appendix D

629
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c
if (ppa == ppa_info_temp->dl_ppa /* &&
    strncmp(ppa_info_temp->dl_module_id_1, "lan", 3) */) {
    found = TRUE;
    break;
} /* end if */
} /* end for */

if (!found)
    return(-1);

/*
 * fill in ATTACH_REQ with the PPA we found, send the
 * ATTACH_REQ, and wait for the OK_ACK
 */
    attach_req->dl_primitive = DL_ATTACH_REQ;
    attach_req->dl_ppa = ppa;
    if (put_ctrl(fd, sizeof(dl_attach_req_t), 0) == -1)
        return(-1);
    if (get_msg(fd) == -1)
        return(-1);
    if (check_ctrl(DL_OK_ACK) == -1)
        return(-1);

    memcpy(&ppa_info, ppa_info_temp, sizeof(ppa_info));

    return(0);
} /* end attach_ppa() */
```
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

detach from the current PPA
*******************************************************************/

int
detach_ppa()
{
dl_detach_req_t *detach_req = (dl_detach_req_t *)ctrl_area;

   detach_req->dl_primitive = DL_DETACH_REQ;
   put_ctrl(fd, sizeof(dl_detach_req_t), 0);
   get_msg(fd);
   check_ctrl(DL_OK_ACK);
}
/* end detach_ppa() */

/* Local Macro and constants definition */

#define COM_TERMLINES 24     /* number of screen lines for most of */
/* the terminals supported on HP-UX*/#define MIN_DISP_LINES 16    /* min screen line number for a*/
/* reasonable display of lanadmin*/
/*commands and results, i.e lanadmin*/
/*doesn't correctly display info for a screen*/
/*<MIN_DISP_LINES*/
#define TERMSTR "TERM"/* to get information on the terminal from*/
/* which the diagnostic is executed*/
#define TERM_INFO 1/* returned value when asking for info in*/
/* terminfo*/

/* system functions declaration    */
char
*getenv();/* get then environment variable TERM*/

Appendix D 631
void
admin_screen_size()
{
intrtncode; /* returned code from setupterm() */
char*termtype; /* pointer to a string returned by getenv() */

/* get the terminal number of lines from terminfo database. */
/* if the TERM variable is not defined in the environment, */
/* or if no info is available for that terminal in terminfo, the */
/* number of lines is set to COM_TERMLINES */

/* initialize the number of lines */
termlines = COM_TERMLINES;

if ((termtype = getenv(TERMSTR)) != NULL) {
    setupterm(termtype, 1, &rtncode);
}

/* info for this terminal found in terminfo data base */
if (rtncode == TERM_INFO)
    termlines = lines; /* lines defined in <term.h> */
}

/* reset termlines to the default if the number of lines got is < */
/* MIN_DISP_LINES */
if (termlines < MIN_DISP_LINES)
    termlines = COM_TERMLINES;
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

} /* end admin_screen_size() */

#define NODE_MAX_NAME_LENGTH 50 /* max node name length*/
#define BUFLEN 100 /* length for integer, boolean and */
/* command strings */
#define INVALID_COMMAND -1 /* invalid command error code*/
#define COMMAND_AMBIGUOUS -2 /* ambiguous command code*/
#define COMMAND_DIALOG_LINES 3 /* maximum number of dialog*/
/* lines displayed when an*/
/* operator enters a bad*/
/* command, including the*/
/* prompt line */
#define MIN_DAY_MONTH 0 /* day, month min, max codes: returned*/
#define MAX_DAY 6 /* by time()*/
#define MAX_MONTH 11
#define MAX_DWORD_ST_LEN 10 /* length of max dword string*/

/* local variables declaration */

/* day table conversion*/
static word day_tab[MAX_DAY+1] =
{ L_SUN, L_MON, L_TUE, L_WED, L_THU, L_FRI, L_SAT };

/* month table conversion*/
static word month_tab[MAX_MONTH+1] =
{
L_JAN, L_FEB, L_MAR, L_APR, L_MAY, L_J UN,
L_J UL, L_AUG, L_SEP, L_OCT, L_NOV, L_DEC
};
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

/* buffer holding strings read from stream files (nodes file or stdin) */
static byte string_buffer[MAX_PATH_NAME_LENGTH+1];

static byte max_dword_string[] = /* max dword ascii string*/
   {'2147483647'};

/* system functions declaration */
dword time();/* return current system date and time*/
struct tm *localtime();/* return pointer to date and time*/
/* structure*/
/***********************
word
utils_get_cmd(menu_type)
word menu_type;/* type of menu from which the command is get */
{
    word command_code; /* holds the code of the command*/
    word lines_count; /* number of lines below menu top line*/
    * tells when menu scrolls off screen*/
    word length; /* command length, returned by*/
    /*utils_get_field()*/
    word j; /* index used to convert upper case letters*/
    /* into lower case letters*/
    byte cmd[BUFLEN+1]; /* input command buffer*/

    break_hit = FALSE;

    while (TRUE) { /* command loop*/

    /* display menu if terse = FALSE*/

    */
/* calculate lines_count depending of terse*/
if ( terse == FALSE )
    lines_count = utils_d_menu(menu_type);
else
    lines_count = 0;

/* while the following dialog stays on the screen...*/
while ( lines_count < termlines -1 ) {
    /* prompt operator*/
    fprintf(stderr, "\n%s",
        utils_get_message(U_ENTER_COMMAND));

    /* get command*/
    length = utils_get_field(BUFLEN+1, cmd);

    if ((( break_hit == TRUE ) || ( length == 0 )) { 
        if ( break_hit == TRUE ) { /* treat break key as field erase */
            break_hit = FALSE;
            fprintf(stderr, "\n\n");
        }
        lines_count = lines_count + COMMAND_DIALOG_LINES-1;
    } else {
        for ( j = 0; j < length; j++ )
            cmd[j] = tolower(cmd[j]);
    }

    /* parse command*/
    command_code = utils_parse_cmd(cmd,menu_type);
}
The driveradmin sample code

/* get some white spaces in msg_buf_1*/
strncpy(msg_buf_1, utils_get_message(D_BLANKS));

if ( command_code == INVALID_COMMAND )
fprintf(stderr, strcat(msg_buf_1,
    utils_get_message(U_UNRECOGNIZED_COMMAND_TRY_AGAIN)),
    cmd);
else
    if ( command_code == COMMAND_AMBIGUOUS )
        fprintf(stderr, strcat(msg_buf_1,
    utils_get_message(U_AMBIGUOUS_COMMAND_TRY_AGAIN)), cmd);
else
    /* return the message number associated to the command */
    return(command_code);

    /* increment the line count due to the dialog*/
    lines_count = lines_count + COMMAND_DIALOG_LINES;
}
} /* end while the displayed menu stays on the screen*/
} /* end while (TRUE) */
} /* end utils_get_cmd() */

******************************************************************

word
utils_d_menu(menu_type)
word menu_type;
{
    word first_header_index,
last_header_index,
first_trailer_index,
last_trailer_index;

word i;/* current header index*/
j;/* current trailer index*/
word line_count;/* displayed menu line counter*/

    /* set the first/last header/trailer index according to the menu type */
    switch (menu_type) {

        case MAIN_COMMAND_MENU :
            first_header_index  = MAIN_LAN_CMD;
            last_header_index   = MAIN_VERBOSE_CMD;
            first_trailer_index = MAIN_LAN_CMD_DEF;
            last_trailer_index  = MAIN_VERBOSE_DEF_CMD_DEF;
            break;

        case LAN_CARD_TEST_MENU :
            first_header_index  = LCS_CLEAR_CMD;
            last_header_index   = LCS_RESET_CMD;
            first_trailer_index = LCS_CLEAR_CMD_DEF;
            last_trailer_index  = LCS_RESET_CMD_DEF;
            break;

        /* end switch (menu_type) */

    /* initialize index search and line_count*/
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c
i = first_header_index;
j = first_trailer_index;
line_count = 0;

/* skip one line before displaying the menu*/
fprintf(stderr, "\n");

/* display the menu lines and count them*/
while ((i <= last_header_index) && (j <= last_trailer_index)) {
    fprintf(stderr, "%s", utils_get_message(D_BLANKS));
fprintf(stderr, "%s", utils_get_message(i));
fprintf(stderr,"%s\n", utils_get_message(j));
i++;
j++;
    line_count++;
}

if (menu_type == LAN_CARD_TEST_MENU) {
    fprintf(stderr, "%s", utils_get_message(D_BLANKS));
fprintf(stderr, "%s", utils_get_message(LCS_DRIVER_MENU_CMD));
fprintf(stderr,"%s\n", utils_get_message(LCS_DRIVER_MENU_CMD_DEF));
    line_count++;
}
/* for third parties */
if (menu_type == MAIN_COMMAND_MENU) {
    fprintf(stderr, "%s", utils_get_message(D_BLANKS));
```
fprintf(stderr, "%s",
utils_get_message(LCS_THIRD_PARTY_MENU_CMD));
fprintf(stderr, "%s\n",
utils_get_message(LCS_THIRD_PARTY_MENU_CMD_DEF));
    line_count++;
}
    line_count = line_count+1;
    return(line_count);
} /* end utils_d_menu() */

/******************************************************************/

word
utils_get_field(lim, cptr )
word lim;    /* return buffer size, including trailing \0*/
byte *cptr;  /* ptr to buffer where data is to be copied */
{
static boolean first_field /* TRUE, this is the first field on line */
    =TRUE;
word len;/* length of the read field*/
word return_length;/* length returned by the function*/
byte c;/* holds input character*/
int s_errno;/* saved errno value*/

    /* flush the input stream until the first field character*/
    /* or a field separator ( end of line )*/

    while (((c = getchar()) != EOF) &&
        ((c == ’’) ||
        /* flush the input stream until the first field character*/
        /* or a field separator ( end of line )*/

Appendix D 639
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

(c == \0') || /* skip null char if any (after break)*/
(c == \t'));

/* get the stream file status*/
/* EOF encountered */
if ( feof(stdin) != FALSE ) { /* EOF while reading the file*/
fprintf(stderr, "\n%s\n",
utils_get_message(U_EOF_ON_INPUT_DIAGNOSTIC_TERMINATED)
);
exit(EXIT_NORMAL);
}

/* error encountered while reading the file? */
/* get the stream file status*/

if ( ferror(stdin) != FALSE ) { /* error while reading the file*/
/* save errno*/
s_errno = errno;

/* break key was hit?*/
if ( break_hit == TRUE ) {
/* reset error indication on stdin*/
clearerr(stdin);

cptr[0] = \0; /* return an empty string*/
return (0); /* return string length = 0*/
}
fprintf(stderr, "\%s\n",
utils_get_message(U_DIAGNOSTIC_ERROR_READING_STDIN));

/* print the system generated error*/
utils_p_syser(stderr, s_errno);
exit(EXIT_ON_PROGRAM_ERROR);
    /* exit with error status*/
    }

    /* input field empty */
    if ( (c == '\n') || (c == ',') ) { /* this is an empty field*/
    /* terminate the input field*/
    cptr[0] = '\0';

    /* set the returned length to 0*/
    return_length = 0;
    } else { /* this is not an empty field*/
    /* copy data from the input stream into a big buffer until*/
    /* an end of field is encountered. ( space, tab, comma, eol )*/

    /* put back into stdin the first character already read*/
    ungetc(c, stdin);

    /* copy data from input stream to field*/
    /* the separators are between [ ]*/
    scanf("\%[^,\t\n]", string_buffer);

    /* read into c the next character*/
    c = getchar();
/* error while reading the file encountered */
/* get the stream file status */
if ( ferror(stdin) != FALSE ) {/* error while reading the file */
    /* save errno */
    s_errno = errno;

    fprintf(stderr, "%s
",
    utils_get_message(U_DIAGNOSTIC_ERROR_READING_STDIN));

    /* print the system generated error */
    utils_p_syser(stderr, s_errno);
    exit(EXIT_ON_PROGRAM_ERROR); /* exit with error status */
}

/* length of the read field */
len = strlen(string_buffer);

/* copy the read field into the returned buffer according to the */
/* length of this buffer. */
if ( len <= lim - 1 ) {/* field <= returned buffer length */
    /* copy the field into the returned buffer as it is*/
    /* including the string terminator*/
    strcpy(cptr, string_buffer);

    /* set the returned length*/
    return_length = len;
} else { /* field > returned buffer length */
/* truncate and copy the field into the returned buffer*/
strncpy(cptr, string_buffer, lim-1);

;/* add string terminator*/
cptr[lim-1] = '\0';

;/* set the returned length*/
return_length = lim-1;
}
}

;/* look for a field separator*/
/* the input holds trailing blanks and/or tabulation char */
if ((c == ' ') || (c == '	')) {
/* flush input until start of next field*/
while ((( c = getchar()) == ' ') || (c == '	'));
/* these get tossed*/
ungetc(c, stdin); /* all other get put back*/
}

;/* Echo the command if it*/
/* was not the first command on the input line. This cause the*/
/* commands on a multi command line to be displayed next to the*/
/* prompt, as well as on the input line*/
if (first_field == TRUE) {/* first field on input line*/
if (c != '\n') /* more than one field on the input line*/
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c
first_field = FALSE; /* not first field now*/
} else { /* not first field on input line*/
  fprintf(stderr, /* echo with new line*/
    "\n",
    cptr);
  if (c == \n) /* end of the input line*/
    first_field = TRUE; /* next field is first*/
}

/* Echo input if echocmd is TRUE.*/
if (echocmd == TRUE) {
  printf("\n", cptr);
  fflush(stdout);
}

/* return field length*/
return(return_length);
} /* end utils_get_field() */

/******************************************************************/

word
utils_parse_cmd(cmd, menu_type)
byte   *cmd;   /* command to be analyzed*/
word   menu_type;   /* type of menu associated to the command */
{
    word first_header_index,/* message number of the first command*/
    /* menu header in the message catalog */
    last_header_index;/* message number of the last command menu */
```
/* header in the message catalog */
word index;/* current header message number */
word return_index;/* returned command message number */
dword cmd_len;/* length of input command */
dword msg_len;/* length of command in the massage catalog */
dword i;/* index in command strings */
boolean command_match;/* TRUE if passed command matches*/
/* the first characters of the command message or the */
/* complete command message */
boolean already_match;/* TRUE if command has already matched*/
/* the first characters of a command message or a */
/* complete command message */
byte *pt_msg;/* pointer to the command get from the message */
/* catalog */
int third_p_display_menu = FALSE;

    cmd_len = strlen(cmd);

    switch (menu_type) {
        case MAIN_COMMAND_MENU :
            first_header_index = MAIN_LAN_CMD;
            last_header_index = MAIN_VERBOSE_CMD;
            third_p_display_menu =TRUE;
            break;
        case LAN_CARD_TEST_MENU :
            first_header_index = LCS_CLEAR_CMD;
            last_header_index = LCS_DRIVER_MENU_CMD;
            break;
    }
/* initialize the current index */
index = first_header_index;
already_match = FALSE;

while (index <= last_header_index) { /* loop on commands in the*/
    /* message catalog */
    /* get command in the message catalog*/
    pt_msg = utils_get_message(index);

    msg_len = strlen(pt_msg);

    if (cmd_len <= msg_len) { /* input command length <= command*/
        /*len compare the passed command with the command*/
        /*in the message catalog*/
        i = 0;
        command_match = TRUE;
        while ((i <= cmd_len-1) && (command_match == TRUE))
            if ( *(cmd+i) == *(pt_msg+i) )
                i++; /* continues*/
            else /* chars differ,* /
                command_match = FALSE;
        /* the loop is exited*/
        if ((command_match == TRUE) && ( cmd_len == msg_len ))
            return(index);

        if ((command_match == TRUE) && ( already_match == TRUE )) {
            /* if strict equality between cmd and msg, return index*/
            return(index);
        }
    }
}
/* the input command matches the first chars of the command */
/* get from the message catalog, but the input command has */
/* already matched the first chars of an other command get */
/* from the message catalog; the input cmd is ambiguous.*/
return(COMMAND_AMBIGUOUS);
}

if (command_match == TRUE) {
    /* the input command matches the first characters of*/
    /* the current command get from the message catalog.*/
    already_match = TRUE;
    return_index = index;
}

index++;
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c
/* catalog

i = 0;
command_match = TRUE;
while ((i <= cmd_len-1) & & (command_match == TRUE))
    if ( *(cmd+i) == *(pt_msg+i) ) /* chars match, the loop */
        i++; /* continues */
    else /* chars differ, */
        /* the loop is exited */
        command_match = FALSE;

/* if strict equality between cmd and msg, return index */
if ((command_match == TRUE) & & (cmd_len == msg_len))
    return(LCS_THIRD_PARTY_MENU_CMD);

if ((command_match == TRUE) & & (already_match == TRUE)) {
    /* the input command matches the first chars of the command */
    /* get from the message catalog, but the input command has */
    /* already matched the first chars of an other command get */
    /* from the message catalog; the input cmd is ambiguous. */
    return(COMMAND_AMBIGUOUS);
}

if (command_match == TRUE) {
    /* the input command matches the first characters of */
    /* the current command get from the message catalog. */
    already_match = TRUE;
    return_index = LCS_THIRD_PARTY_MENU_CMD;
}
```
The driveradmin sample code

if (already_match == TRUE) {/* good abbreviated command*/
    return(return_index);
}

/* not found */
return(INVALID_COMMAND);
} /* end utils_parse_cmd() */

/***************************************************/

void
utils_quit_cmd()
{
#define OPEN_ERROR -1
#define ACCESS_ERROR -2

    if ((nls_lanadmin != OPEN_ERROR) && (nls_lanadmin != ACCESS_ERROR)) */
    catclose (nls_lanadmin);
    exit(EXIT_NORMAL);
} /* end utils_quit_cmd() */

/***************************************************/
utils_d_datetime(file_ptr)
FILE *file_ptr;
{
    dword clock;/* holds the system time value*/
    struct tm *ptdate; /* pointer to the date and time structure */
    /* set by the localtime() function*/
    word nls_day_code; /* holds the ptdate->tm_wday day code*/
    word nls_month_code; /* holds the ptdate->tm_mon month code */
    /* get system time */
    clock = time((dword *) 0);

    /* get pointer on date and time structure*/
    ptdate = localtime(&clock);

    /* get date and time from the Message Catalog*/
    /* select the day code according to the day number*/
    nls_day_code = day_tab[ptdate->tm_wday];

    /* get the month code according to the month number*/
    nls_month_code = month_tab[ptdate->tm_mon];

    /* get the day string; copy it in a buffer*/
    strcpy(msg_buf_1, utils_get_message(nls_day_code));

    /* get the month string; copy it in a buffer*/
    strcpy(msg_buf_2, utils_get_message(nls_month_code));
/* print out the date according to the localized format*/
    fprintf(file_ptr, "%s",
/* get the date and time localized format*/
    utils_get_message(DIAG_DATE_FORMAT),
    msg_buf_1,        /* day string*/
    msg_buf_2,        /* month string*/
    ptdate->tm_mday,  
    ptdate->tm_year + 1900,
    ptdate->tm_hour,  
    ptdate->tm_min,   
    ptdate->tm_sec);

    fprintf(file_ptr, "\n \n");
} /* end utils_d_datetime() */

/*---------------------------------------------*/

byte
*utils_get_message(msg_number)
int msg_number; /*!< Message number of the message asked for */
{
    char *msg_buffer; /*!< holds the message*/
    int base;        /*!< base value of the messages*/
    int offset;      /*!< index into the message array*/
    int message_set; /*!< number of set in nls message catalog */
    char *default_message; /*!< message to use if nls file is not there */
    #define L_DIAG_BASE 7000 /*!< base value of lanadmin messages*/
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

#define L_SHARE_BASE 9000   /* base value of shared messages*/

/* compute the index into the nls catalog file and get the message */

     base= (msg_number/1000) * 1000;
     offset = msg_number - base;
     switch (base) {
         case L_DIAG_BASE :
             default_message = admin_msgtable[offset];
             message_set = 1;
             break;
         case L_SHARE_BASE :
             default_message = share_msgtable[offset];
             message_set = 2;
             break;
         default:
             fprintf (stderr, "bad message number %d", msg_number);
             exit (-1);
     }

     msg_buffer =
     catgets(nls_lanadmin,message_set,msg_number,default_message);
     return(msg_buffer);
} /* end utils_get_message() */

**************************************************************************

void
utils_p_syser(file_ptr, s_errno)
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

FILE *file_ptr;
int s_errno; /* saved errno value*/
{
/* print the system generated error*/
    fprintf(file_ptr, "errno = %d\n", s_errno);
} /* end utils_p_syser() */

get_mac_type()
{
    dl_info_req_t *get_info_req = (dl_info_req_t *) ctrl_area;
    dl_info_ack_t *get_info_ack = (dl_info_ack_t *) ctrl_area;
    int get_info_failed = 0;

    get_info_req->dl_primitive = DL_INFO_REQ;
    if (put_ctrl(fd, sizeof(dl_info_req_t), 0) == -1)
        get_info_failed = TRUE;
    else if (get_msg(fd) == -1)
        get_info_failed = TRUE;
    else if (check_ctrl(DL_INFO_ACK) == -1)
        get_info_failed = TRUE;

    if (get_info_failed) {
        fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5020,
            "get_info: unable to get DL info\n"));
        exit(-1);
    }
}
The driveradmin sample code

```
    return (get_info_ack->dl_mac_type);
} /* end get_mac_type() */

/*******************************************************************/
int
display_rif_flag()
{
    struct strioctl strioctl;
    int mac_type, rif_flag = -1;

    /* DL_TPR, MAC type, is defined in h/dlpi.h or
    * DEV_8025 in sio/lan_dlpikrn.h
    */

    if ((mac_type = get_mac_type()) != DL_TPR) {
        fprintf(stderr, catgets(nls_lanadmin,NL_SETN,5021,
            "Wrong interface: %d. Source Routing applies to token ring
            only.\n"), mac_type);
        exit(-1);
    }

    strioctl.ic_cmd = DLPI_GET_SRC_ROUTE_FLAG;
    strioctl.ic_timout = 0;
    strioctl.ic_len = sizeof(rif_flag);
    strioctl.ic_dp = &rif_flag;
    if (ioctl(fd, I_STR, &strioctl) < 0) {
        fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5022,
            "Wrong interface: %d. Source Routing applies to token ring only.\n"), mac_type);
        exit(-1);
    }
```
/* Unable to get RIF flag: errno = %d
  
  exit(1);
  */

} /* end display_rif_flag() */

/*****************************/

change_rif_flag()
{
    char rif_arg[20];
    struct strioctl strioctl;
    int i, mac_type, rif_flag, old_rif = -1;

    for (i = 0; new_rif[i]; i++)
        rif_arg[i] = tolower(new_rif[i]);
    rif_arg[i] = '\0';
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c
if (!strcmp (rif_arg, STR_ON))
    rif_flag = TURN_ON;
else if (!strcmp (rif_arg, STR_OFF))
    rif_flag = TURN_OFF;
else {
    fprintf(stderr, catgets(nls_lanadmin,NL_SETN,7024,
        "Unrecognized command line argument: %s, try again.\n"),
        new_rif);
    exit(-1);
}

if ((mac_type = get_mac_type()) != DL_TPR) {
    fprintf(stderr, catgets(nls_lanadmin,NL_SETN,5021,
        "Wrong interface: %d. Source Routing applies to token ring
only.\n"), mac_type);
    exit(-1);
}

printf((catgets(nls_lanadmin,NL_SETN,6005, "Old ")));
old_rif = display_rif_flag();
if (old_rif == rif_flag) {
    fprintf(stderr, catgets(nls_lanadmin,NL_SETN,5023,
        "No change has been made.\n"));
    exit(0); /* prakash : INDaa27035 : Not an error condition, return 0 */
}

/* Must be super-user */
if (getuid() != 0) {
```
fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5005, "Must be super-user\n"))); /* catgets 5005 */
exit(-1);
}

  strioctl.ic_cmd = DLPI_SET_SRC_ROUTE_FLAG;
  strioctl.ic_timeout = 0;
  strioctl.ic_len = sizeof(rif_flag);
  strioctl.ic_dp = &rif_flag;
  if (ioctl(fd, I_STR, &strioctl) < 0) {
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5024, "Unable to change RIF flag: errno = %d\n")), errno);
    exit(1);
  }

  printf((catgets(nls_lanadmin,NL_SETN,6006, "New ")));
  display_rif_flag();
} /* end change_rif_flag() */

/*************************************************************************/

set_driver_special()
{
  int(*driver_special)();
  shl_t*libptr;
  intfn_found = 0;
  inerror;
/* Must be super-user */
if (getuid() != 0) {
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5005,
        "Must be super-user\n")));
    exit(-1);
}

/* The name of the shared library that contains the function to
 * do the firmware download associated with this device is:
 * "/usr/lib/lanadmin/libdl<driver_name>.sl"
 * OR "/usr/lib/lanadmin/libds<driver_name>.sl" */

    sprintf(shlib_filename, "/usr/lib/lanadmin/libdl%s.sl", ppa_info.dl_name);

    /* Load the shared library */
    if ((lib = shl_load(shlib_filename, BIND_IMMEDIATE, 0)) != -1) {

    /* The name of the shared library function to do the firmware */
    *download associated with this device is:
    * "dl<driver_name>_set_special"

    sprintf(shlib_funcname, "dl%s_set_special", ppa_info.dl_name);

    /* Get the address of the shared library function */
    libptr = &lib;
    if (shl_findsym(libptr, shlib_funcname, TYPE_PROCEDURE, &driver_special) < 0)
        shl_unload(lib);
    else fn_found = 1;
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

if (!fn_found) {
    sprintf(shlib_filename, "/usr/lib/lanadmin/libds%s.sl", ppa_info.dl_name);
    /* Load the alternate shared library */
    if ((lib = shl_load(shlib_filename, BIND_IMMEDIATE, 0)) == 0) {
        fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5116,
            "This option is not supported for this hardware.\n")));
        detach_ppa();
        return;
    }
    sprintf(shlib_funcname, "ds%s_set_special", ppa_info.dl_name);
    /* Get the address of the shared library function */
    libptr = &lib;
    if (shl_findsym(libptr, shlib_funcname, TYPE_PROCEDURE, &driver_special) < 0) {
        fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5116,
            "This option is not supported for this hardware.\n")));
        detach_ppa();
        exit(-1);
    }
}

/* Load/unload firmware for this interface */
error = (*driver_special)(fd, ppa_info.dl_ppa, driver_special_arg);

/* Unload the shared library after we're done with it */
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c
shl_unload(lib);

if (error)
exit(error);
} /* end set_driver_special() */

/***************************************************/

get_driver_special()
{
int(*driver_special)();
shl_t*libptr;
intfn_found = 0;
interror;
char *ret;

/* The name of the shared library that contains the function to
 * do the firmware download associated with this device is:
 *   
 *   
 *   *   
 *   */

sprintf(shlib_filename, "/usr/lib/lanadmin/libdl<driver_name>.sl",
ppa_info.dl_name);

/* Load the shared library */
if ((lib = shl_load(shlib_filename, BIND_IMMEDIATE, 0)) != -1) {

/* The name of the shared library function to do the firmware
 * download associated with this device is:
 *   
 *   */
```
* "dl<driver_name>_get_special" */
  sprintf(shlib_funcname, "dl%s_get_special", ppa_info.dl_name);

  /* Get the address of the shared library function */
  libptr = &lib;
  if (shl_findsym(libptr, shlib_funcname, TYPE_PROCEDURE, &driver_special) < 0)
    shl_unload(lib);
    else fn_found = 1;
  }

  if (!fn_found) {
    sprintf(shlib_filename, "/usr/lib/lanadmin/libds%s.sl", ppa_info.dl_name);
    /* Load the alternate shared library */
    if ((lib = shl_load(shlib_filename, BIND_IMMEDIATE, 0)) == 0) {
      fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5116,
        "This option is not supported for this hardware.\n")));
      detach_ppa();
      return;
    }
    sprintf(shlib_funcname, "ds%s_get_special", ppa_info.dl_name);

    /* Get the address of the shared library function */
    libptr = &lib;
    if (shl_findsym(libptr, shlib_funcname, TYPE_PROCEDURE, &driver_special) < 0) {
      fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5116,
        "This option is not supported for this hardware.\n")));
      
      /* Get the address of the shared library function */
      libptr = &lib;
      if (shl_findsym(libptr, shlib_funcname, TYPE_PROCEDURE, &driver_special) < 0) {
        fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5116,
          "This option is not supported for this hardware.\n"));

Appendix D  661
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

detach_ppa();
exit(-1);

} /* end get_driver_special() */

/******************************************************************/

menu_driver_special()
{
int(*driver_special)();
shl_t*libptr;
intfn_found = 0;
tererror;
char *ret;

if (error)
exit(error);
} /* end menu_driver_special() */

Appendix D
/* The name of the shared library that contains the function to
do the firmware download associated with this device is:
*    "/usr/lib/lanadmin/libdl<driver_name>.sl"
*  OR "/usr/lib/lanadmin/libds<driver_name>.sl" */

if (attach_ppa(cur_ppa) == -1)
{
    fprintf(stderr, "%s %d
",
    utils_get_message(L_UNABLE_TO_ACCESS_NMID),
    cur_ppa);
    return(-1);
}

sprintf(shlib_filename, "/usr/lib/lanadmin/libdl%s.sl",
    ppa_info.dl_name);

/* Load the shared library */
if ((lib = shl_load(shlib_filename, BIND_IMMEDIATE, 0)) != -1) {
    /* The name of the shared library function to do the firmware
    *download
    * associated with this device is:
    *    "dl<driver_name>_menu" */
sprintf(shlib_funcname, "dl%s_menu", ppa_info.dl_name);

    /* Get the address of the shared library function */
    libpstr = &lib;
    if (shl_findsym(libpstr, shlib_funcname, TYPE_PROCEDURE,
        &driver_special) < 0)
        shlUnload(lib);
    else fn_found = 1;

Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c
}

if (!fn_found) {
  sprintf(shlib_filename, "/usr/lib/lanadmin/libds%s.sl", 
  ppa_info.dl_name);
  /* Load the alternate shared library */
  if ((lib = shl_load(shlib_filename, BIND_IMMEDIATE, 0)) == 0) {
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5116,
      "This option is not supported for this hardware.\n")));
    detach_ppa();
    return(-1);
  }
  sprintf(shlib_funcname, "ds%s_menu", ppa_info.dl_name);
  /* Get the address of the shared library function */
  libptr = &lib;
  if (shl_findsym(libptr, shlib_funcname, TYPE_PROCEDURE, &driver_special) < 0) {
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5116,
      "This option is not supported for this hardware.\n")));
    detach_ppa();
    return(-1);
  }

  /* Load/unload firmware for this interface */
  error = (*(driver_special))(fd, ppa_info.dl_ppa);

  /* Unload the shared library after we're done with it. */
```

Appendix D
shl_unload(lib);

detach_ppa();
/* Exit on error */
if (error)
exit(error);
return(0);
} /* end menu_driver_special() */

############################################################
/** enetadmin.h **/
/*
  * incorporated the types from ntypes.h  12/4/89
  * DESCRIPTION:       Global type and constant definitions.
  */

typedef char byte;     /* 8 bits -128..127 */
typedef short word;    /* 16 bits -32768,,32767 */
typedef long dword;    /* 32 bits */

/* NLS Defines */
#define NL_SETN 1/* set in lanadmin message catalog */

/* compare codes */
#define EQUAL 0 /* string equality */
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

#define NULL 0

/* error codes */
#define NO_SYS_ER 0 /* code returned by system calls when */
/* successful */
#define SYS_ER -1 /* code returned by system calls when */
/* an error occurs */
#define NO_ERROR 0 /* diagnostic function return code if no */
/* error occurs */
#define FILE_ERROR -2 /* returned by function when an error */
/* while reading a file */
/* exit codes */
#define EXIT_ON_BAD_EXEC_OPTIONS 2
#define EXIT_NORMAL 0
#define EXIT_ON_PROGRAM_ERROR -1

/* well known constants */
#define MAX_PATH_NAME_LENGTH 1024 /* HP-UX max path name */
/* length */
/* in the 4.0 version */
#define MAIN_COMMAND_MENU 1 /* Main command menu type */
#define LAN_CARD_TEST_MENU 2 /* LAN Interface test menu type */
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

/* network message numbers */

#define D_BLANKS  7000

/* main module messages */

#define M_WAKE_UP  7001
#define M_COPYRIGHT_1  7002
#define M_COPYRIGHT_2  7003
#define M_TEST_SELECTION_MODE  7004
#define M_DO_NOT_DISPLAY_COMMAND_MENU  7005
#define M_DISPLAY_COMMAND_MENU  7006

#define L_LAN_CARD_TEST_MODE  7007
#define L_END_OF_LAN_CARD_TEST_MODE  7008
#define L_NOT_AUTHORIZED_TO_CLEAR_STATISTICS  7009
#define L_LAN_CARD_NOT_ACTIVE_UNABLE_TO_CLEAR_STATISTICS  7010
#define LCLEARING_LAN_CARD_STATISTICS_REGISTERS  7011
#define L_UNABLE_TO_CLEAR_STATISTICS_REGISTERS  7012
#define L_LAN_CARD_STATUS_DISPLAY  7013
#define L_NM_ID  7014
#define L_ENTER_NM_ID_CURRENTLY  7015
#define L_NOT_ALLOWED_TO_RESET_LAN_CARD  7016
#define L_RESETTING_LAN_CARD_TO_RUN_SELFTEST  7017
#define L_UNABLE_TO_RESET_LAN_CARD  7018
#define L_UNABLE_TO_ACCESS_NMID  7019
#define L_UNABLE_TO_OPEN_DEVICE_FILE  7020
#define L_INVALID_NMID_ENTRY  7021
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c
#define U_ENTER_COMMAND 7022
#define U_AMBIGUOUS_COMMAND_TRY_AGAIN 7023
#define U_UNRECOGNIZED_COMMAND_TRY_AGAIN 7024
#define U_EOF_ON_INPUT_DIAGNOSTIC_TERMINATED 7025
#define U_DIAGNOSTIC_ERROR_READING_STDIN 7026
#define U_HIT_RETURN_KEY_TO_KEEP_IT_OR_ENTER_A_NEW_VALUE 7027

/* main menu */
#define MAIN_LAN_CMD /* header part */ 7029
#define MAIN_MENU_CMD 7030
#define MAIN_QUIT_CMD 7031
#define MAIN_TERSE_CMD 7032
#define MAIN_VERBOSE_CMD 7033
#define MAIN_LAN_CMD_DEF /* trailer part */ 7034
#define MAIN_MENU_CMD_DEF 7035
#define MAIN_QUIT_CMD_DEF 7036
#define MAIN_TERSE_CMD_DEF 7037
#define MAIN_VERBOSE_CMD_DEF 7038

/* menu of lcs commands */
#define LCS_CLEAR_CMD /* header part */ 7039
#define LCS_DISPLAY_CMD 7040
#define LCS_END_CMD 7041
#define LCS_LAN_CMD 7042
```
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

#define LCS_NMID_CMD     7043
#define LCS_QUIT_CMD     7044
#define LCS_RESET_CMD     7045
#define LCS_CLEAR_CMD_DEF /* trailer part */     7046
#define LCS_DISPLAY_CMD_DEF 7047
#define LCS_END_CMD_DEF    7048
#define LCS_MENU_CMD_DEF    7049
#define LCS_NMID_CMD_DEF    7050
#define LCS_QUIT_CMD_DEF    7051
#define LCS_RESET_CMD_DEF    7052
#define DIAG_DATE_FORMAT    7053
#define LCS_DRIVER_MENU_CMD    7054
#define LCS_DRIVER_MENU_CMD_DEF 7055

/* for third party options */
#define LCS_THIRD_PARTY_MENU_CMD 7056
#define LCS_THIRD_PARTY_MENU_CMD_DEF 7057

void
break_key();

void
lcs_get_name();

void
lcs_reset();
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

dword
lcs_display_trn_ring();

word
utils_get_cmd();

word
utils_d_menu();

word
utils_get_field();

word
utils_parse_cmd();

void
utils_quit_cmd();

void
utils_d_datetime();

byte
*utils_get_message();

void
utils_p_syser();
#define MAX_MSG_LENGTH 140 /* maximum message length */
#define MSG_BUFFER_SIZE 141 /* maximum message length + */
/* string terminator character */

/* "shared" message numbers: days and months msg numbers */
#define L_SUN 9000
#define L_MON 9001
#define L_TUE 9002
#define L_WED 9003
#define L_THU 9004
#define L_FRI 9005
#define L_SAT 9006
#define L_JAN 9007
#define L_FEB 9008
#define L_MAR 9009
#define L_APR 9010
#define L_MAY 9011
#define L_JUN 9012
#define L_JUL 9013
#define L_AUG 9014
#define L_SEP 9015
#define L_OCT 9016
#define L_NOV 9017
#define L_DEC 9018

#**************************************************************************

Appendix D 671
Shared Library Examples for the driveradmin and lanscan Commands
The driveradmin sample code

/* BEGIN.IMS enetadmin_mg.c */

***************************************************************************/

****
****enetadmin_mg.c
****

***************************************************************************/

* Description
* This file is used to supply the messages to lanadmin when enetadmin.cat
* does not exist.
* To Do List
*
* Notes:
*
* Module Size (lines of non-commented source)
* Expected: 0
*
***************************************************************************/

* END.IMS enetadmin_mg.c */

#define MODULEID
#if defined(MODULEID) && !defined(lint)
static char rcsid[]="@(#) lanadmin_mg.c: PHNE_15969 98/09/15";
#endif

char *admin_msgtable[] = {
" ", /* catgets 7000*/
"LOCAL AREA NETWORK ONLINE ADMINISTRATION, Version
", /* catgets 7001*/
"Copyright 1994 Hewlett Packard Company.", /* catgets 7002*/
"All rights are reserved.", /* catgets 7003*/
"Test Selection mode.", /* catgets 7004*/
"Do not display command menu.", /* catgets 7005*/
"Display command menu.", /* catgets 7006*/

"LAN Interface test mode. LAN Interface Instance Number (PPA) = ", /* catgets 7007*/
"End of LAN Interface test mode.", /* catgets 7008*/
"Not authorized to clear statistics.", /* catgets 7009*/
"LAN Interface is not active, unable to clear statistics.", /* catgets 7010*/
"Clearing LAN Interface statistics registers.", /* catgets 7011*/
"Unable to clear statistics registers.", /* catgets 7012*/
"LAN INTERFACE STATUS DISPLAY", /* catgets 7013*/
"LAN Interface Instance Number (PPA) = ", /* catgets 7014*/
"Enter PPA Number. Currently ", /* catgets 7015*/
"Not authorized to reset LAN Interface.", /* catgets 7016*/
"Resetting LAN Interface to run selftest.", /* catgets 7017*/
"Unable to reset LAN Interface.", /* catgets 7018*/
"Unable to access PPA Number ", /* catgets 7019*/
"Unable to open device file", /* catgets 7020*/
"Invalid PPA Number entry\n", /* catgets 7021*/
"Enter command: ", /* catgets 7022*/
"Ambiguous command, try again.", /* catgets 7023*/
"Unrecognized command, try again.", /* catgets 7024*/

Appendix D 673
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

"Administration terminated by EOF on input.", /* catgets 7025*/
"Administration error reading 'stdin', /* catgets 7026*/
"Hit RETURN to keep it, or enter a new value: ", /* catgets 7027*/
"", /* catgets 7028*/

/* main menu */

"lan", /* catgets 7029*/
"menu", /* catgets 7030*/
"quit", /* catgets 7031*/
"Terse", /* catgets 7032*/
"verbose", /* catgets 7033*/
" = LAN Interface Administration", /* catgets 7034*/
" = Display this menu", /* catgets 7035*/
" = Terminate the Administration", /* catgets 7036*/
" = Do not display command menu", /* catgets 7037*/
" = Display command menu", /* catgets 7038*/

/* menu of lcs commands */

"clear", /* catgets 7039*/
"display", /* catgets 7040*/
"end", /* catgets 7041*/
"menu", /* catgets 7042*/
"ppa", /* catgets 7043*/
"quit", /* catgets 7044*/
"reset", /* catgets 7045*/
" = Clear statistics registers", /* catgets 7046*/

674  Appendix D
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

" = Display LAN Interface status and statistics registers",
/* catgets  7047*/
" = End LAN Interface Administration, return to Test Selection",
/* catgets  7048*/
" = Display this menu", /* catgets  7049*/
" = PPA Number of the LAN Interface", /* catgets  7050*/
" = Terminate the Administration, return to shell", /* catgets  7051*/
" = Reset LAN Interface to execute its selftest", /* catgets  7052*/
/* Administration date print format */
" %1$s,%2$s %3$d,%4$d  %5$.2d:%6$.2d:%7$.2d",
/* catgets  7053*/
"specific", /* catgets  7054*/
" = Go to Driver specific menu", /* catgets  7055*/
"3rd party",
" = Enter <path>/filename for third parties’ options",

};

char *share_msgtable[] = {
"Sun", /* catgets  9000 */
"Mon", /* catgets  9001 */
"Tue", /* catgets  9002 */
"Wed", /* catgets  9003 */
"Thu", /* catgets  9004 */
"Fri", /* catgets  9005 */
"Sat", /* catgets  9006 */
" Jan", /* catgets  9007 */
" Feb", /* catgets  9008 */
Shared Library Examples for the `driveradmin` and `lanscan` Commands

**The `driveradmin` sample code**

```c
"Mar", /* catgets 9009 */
"Apr", /* catgets 9010 */
"May", /* catgets 9011 */
"Jun", /* catgets 9012 */
"Jul", /* catgets 9013 */
"Aug", /* catgets 9014 */
"Sep", /* catgets 9015 */
"Oct", /* catgets 9016 */
"Nov", /* catgets 9017 */
"Dec" /* catgets 9018 */
};
```

# Makefile for `enetadmin` tool and its shared library.

```plaintext
SRCLOC=/opt/enetadmin

LIB=/usr/lib
CP = /usr/ccs/bin/cp
CC = /usr/ccs/bin/cc
LD =/usr/ccs/bin/ld
GENCAT = /usr/bin/gencat
INCLUDES= -I /usr/include
LDFLAGS= +b $(LIB)
```
CFLAGS=$(INCLUDE) $(LAN_CFLAGS) -D__HP_CURSES \ 
-Wp,-H300000 -L $(LIB)

OBJ S = enetadmin.o enetadmin_mg.o
LIBS = /usr/lib/libHcurses.a

default all: enetadmin  libdsenet.1
enetadmin: $(OBJ S)
$(CC) $(CFLAGS) -o enetadmin $(OBJ S) $(LIBS) -ldld
dsenet.o:
$(CC) $(CFLAGS) +z -c $(SRCLOC)/dsenet.c

libdsenet.1:  dsenet.o
$(LD) $(LDFLAGS) +h libdsenet.1 -b -o libdsenet.1 dsenet.o

$(CC) -o linkloop linkloop.o
The enetlinkloop’s sample code

/**************************** enetlinkloop.c*************************/

*MODULE:  enetlinkloop
*
*DESCRIPTION:
*This user-space program checks LAN connectivity by
*writing TEST frames to specified nodes.
*DEPENDENCIES:
*This program requires DLPI.
*
*******************************************************************/

#define MODULEID
#define WHAT_STRING(a, b, c, d) a##b##c##d
#if defined(MODULEID) && !defined(lint)
static char rcsid[] = WHAT_STRING("@(#) linkloop.c: PHNE_17113 ",
__DATE__, "", __TIME__);#endif

#include <stdio.h>
#include <ctype.h>
#include <fcntl.h>
#include <signal.h>
#include <netio.h>
#include <nl_types.h>
The enetlinkloop's sample code

```c
#include <locale.h>
#include <unistd.h>
#include <sys/topsyms.h>
#include <sys/stat.h>
#include <sys/param.h>
#include <machine/param.h>
#include <machine/pde.h>
#include <sys/libio.h>
#include <sys/llio.h>
#include <sys/stream.h>
#include <sys/mib.h>
#include <sys/stream.h>
#include <sys/dlpi.h>
#include <sys/dlpi_ext.h>

extern char*sys_errlist[];
extern interno;
extern char*setlocale();
extern char*getenv();
char*catgets();
voidnlStart();
voidtimeout_hdtr();
intmyintr();

#define NL_SETN1
#define NATIVE_COMPUTER "C"
```
Shared Library Examples for the driveradmin and lanscan Commands

The enetlinkloop’s sample code

#define ADDRSIZE 6
#define FCADDRSIZE 3 /* Fibre Channel (FC) N_Port Address Size */
#define FC_TYPE 4 /* Fibre Channel (FC) Linkloop type */
#define BUFSIZE 65280
#define MAX_802_31497
#define MAX_802_54135
#ifdef NOCODE
#define MAX_FDDI 4475
#endif
#define MAX_FDDI 4000
#define MAX_FC 65280 /* Max. user specifiable pkt size for FC */
#define SEND_SAP 0x80
#define MAX_INT 0x7FFFFFFF
#define INDIVIDUAL(addr)((addr)[0] & 0x1) == 0
#define TOKINDIVIDUAL(addr)((addr)[0] & 0x80) == 0
#define TODIGIT(c)((int)(c) - (int)'0')
#define MAX_RIF_SIZE_U 16 /* max user input for rif */
#define MAX_RIF_SIZE 18 /* max rif size - user + 2 control bytes */

u_char rftmp[MAX_RIF_SIZE_U*3];

/* These global variables can be accessed during signal handler */
int sent;/* number of frames sent */
int recv_ok;/* number of frames received which matched */
int recv_bad_len;/* number of frames received with length error */
int recv_bad_data;/* number of frames received with data error */
int recv_bad_header;/* number of frames received with bad header */
int recv_timeout;/* number of frame not received because of timeout */
int isr_print;/* boolean if the isr should print out the status */

int mtusize;/* size of buffer to transmit */
int maxmtusize;/* Maximum size of MTU for link type */
nl_catd nls_linkloop;

int fd;
int timeout;/* timeout (in seconds) */
int addrsize;

#define BYTE_INVERT(byte) bit_reverse[byte]

static unsigned char bit_reverse[] = {
0x00, 0x80, 0x40, 0xc0, 0x20, 0xa0, 0x60, 0xe0,
0x10, 0x90, 0x50, 0xd0, 0x30, 0xb0, 0x70, 0xf0,
0x08, 0x88, 0x48, 0xc8, 0x28, 0xa8, 0x68, 0xe8,
0x18, 0x98, 0x58, 0xd8, 0x38, 0xb8, 0x78, 0xf8,
0x04, 0x84, 0x44, 0xc4, 0x24, 0xa4, 0x64, 0xe4,
0x14, 0x94, 0x54, 0xd4, 0x34, 0xb4, 0x74, 0xf4,
0x0c, 0x8c, 0x4c, 0xcc, 0x2c, 0xac, 0x6c, 0xec,
0x1c, 0x9c, 0x5c, 0xc2, 0x3c, 0xbc, 0x7c, 0xfc,
0x02, 0x82, 0x42, 0xc2, 0x22, 0xa2, 0x62, 0xe2,
0x12, 0x92, 0x52, 0xd2, 0x32, 0xb2, 0x72, 0xf2,
0x0a, 0x8a, 0x4a, 0xca, 0x2a, 0xaa, 0x6a, 0xea,
0x1a, 0x9a, 0x5a, 0xda, 0x3a, 0xba, 0x7a, 0xfa,
0x06, 0x86, 0x46, 0xc6, 0x26, 0xa6, 0x66, 0xe6,
Shared Library Examples for the driveradmin and lanscan Commands

The enetlinkloop's sample code

```c
0x16, 0x96, 0x56, 0x36, 0xb6, 0x76, 0xf6, 0x0e, 0x8e, 0x4e, 0xce, 0x2e, 0xae, 0x6e, 0xee,
0x1e, 0x9e, 0x5e, 0x3e, 0xbe, 0x7e, 0xfe, 0x01, 0x81, 0x41, 0xc1, 0x21, 0xa1, 0x61, 0xe1,
0x11, 0x91, 0x51, 0x31, 0xb1, 0x71, 0xf1, 0x09, 0x89, 0x49, 0xc9, 0x29, 0xa9, 0x69, 0xe9,
0x19, 0x99, 0x59, 0x39, 0xb9, 0x79, 0xf9, 0x05, 0x85, 0x45, 0xc5, 0x25, 0xa5, 0x65, 0xe5,
0x15, 0x95, 0x55, 0x35, 0xb5, 0x75, 0xf5, 0x0d, 0x8d, 0x4d, 0xcd, 0x2d, 0xad, 0x6d, 0xed,
0x1d, 0x9d, 0x5d, 0x3d, 0xbd, 0x7d, 0xfd, 0x03, 0x83, 0x43, 0xc3, 0x23, 0xa3, 0x63, 0xe3,
0x13, 0x93, 0x53, 0x33, 0xb3, 0x73, 0xf3, 0x0b, 0x8b, 0x4b, 0xcb, 0x2b, 0xab, 0x6b, 0xeb,
0x1b, 0x9b, 0x5b, 0xdb, 0x3b, 0xbb, 0x7b, 0xfb, 0x07, 0x87, 0x47, 0xc7, 0x27, 0xa7, 0x67, 0xe7,
0x17, 0x97, 0x57, 0x37, 0xb7, 0x77, 0xf7, 0x0f, 0x8f, 0x4f, 0xcf, 0x2f, 0xaf, 0x6f, 0xef,
0x1f, 0x9f, 0x5f, 0xdf, 0x3f, 0xbf, 0x7f, 0xff};

#define LKL_CANONICAL_WIRE_TOGGLE(addr, size) {  
int index;
for ( index = 0 ; index < size; index++ ) 
    addr[index] = BYTE_INVERT(addr[index]);
}

/***************************************************************************/
global areas for sending and receiving streams messages
`
The enetlinkloop's sample code

#define AREA_SIZE 65280 /* bytes; big enough for largest possible msg */
#define LONG_AREA_SIZE (AREA_SIZE / sizeof(u_long)) /* AREA_SIZE / 4 */

u_long ctrl_area[LONG_AREA_SIZE]; /* for control messages */
u_long data_area[LONG_AREA_SIZE]; /* for data messages */
u_long ppa_area[LONG_AREA_SIZE]; /* for saving ppa area */

struct strbuf ctrl_buf = {
  AREA_SIZE, /* maxlen = AREA_SIZE */
  0, /* len gets filled in for each msg */
  ctrl_area /* buf = control area */
};

struct strbuf data_buf = {
  AREA_SIZE, /* maxlen = AREA_SIZE */
  0, /* len gets filled in for each msg */
  data_area /* buf = data area */
};

dl_hp_ppa_info_t ppa_info;
int default_ppa;
int ppa_count;

/* Main program:
 *
 *process parameters
Shared Library Examples for the driveradmin and lanscan Commands

The enetlinkloop's sample code

*set-up LAN IO
*for all addresses
*  for all count
*write
*receive
*check for errors in receive
*  print results
************************************************************************/
main(argc, argv)
int argc;
char *argv[];
{
  int verbose;/* to print the details of the errors or not */
  int count;/* number of frames to send */
  char *ppa_str;/* PPA Number of interface to test */
  int opt;/* argument list option letter */
  int opterr;/* boolean indicating error in arguments */
  int rif_len;
  char rif[18];
  extern int optind;/* getopt indicating current index of argv */
  extern char *optarg;/* getopt global points to current argument */
  struct fis arg;/* used for ioctl's */
  unsigned char txbuf[BUFSIZE];/* buffer used for transmitting */
  int i;/* indexes through argv for the link addrs */
  char *count_str;/* char ptr to the parameter of the -n option */
  char *timeout_str;/* char ptr to the parameter of the -t option */
  char *size_str;/* char ptr to the parameter of the -s option */
  char *rif_str;/* chat ptr to the parameter of the -r option */
inttemp_ppa;
u_chardlsap[20];
intdlsap_len;
intthirdp_opt=FALSE;
char* thirdp_drv_path;

nls_start(); /* initialize NLS parameters */

for (i = 0; i < BUFSIZE; i++)
    txbuf[i] = i % 256;

count = 1;
timeout = 2;
verbose = FALSE;
ppa_str = count_str = timeout_str = size_str = rif_str = NULL;

/*
 * process the parameters
 */
opterr = FALSE;
while ((opt = getopt(argc, argv, "vi:n:t:s:r:3:")) != EOF) {
    switch (opt) {
    case 'i':
        ppa_str = optarg;
        break;
    case 'n':
        count_str = optarg;
        break;
    default:
        break;
    }
}
Shared Library Examples for the driveradmin and lanscan Commands

The enetlinkloop's sample code

case 't':
    timeout_str = optarg;
    break;
case 's':
    size_str = optarg;
    break;
case 'r':
    rif_str = optarg;
    break;
case 'v':
    verbose = TRUE;
    break;
case '3':
    thirdp_opt = TRUE;
    thirdp_drv_path = optarg;
    break;
default:
    opterr = TRUE;

if (opterr || optind >= argc)
    usage_error();

/**************************************************************************
 *      *
 * Insert this part to add an option for 3rd party. It is temporarily *
 */
Shared Library Examples for the driveradmin and lanscan Commands
The enetlinkloop's sample code

* getting third parties's device file from user input at command line
*
* as -3 /dev/dlpi_xxx
*

******************************************************************

if (thirdp_opt) {
/* Open the third parties' device file, /dev/dlpi_xxx. */
    if ((fd = open(thirdp_drv_path, O_RDWR)) == -1) {
        fprintf(stderr, "Unable to open device file
%s
", thirdp_drv_path);
            exit(1);
    }
}
else {
    /* Open the DLPI device file, /dev/dlpi. */
    if ((fd = open("/dev/dlpi", O_RDWR)) == -1) {
        fprintf(stderr, catgets(nls_linkloop, NL_SETN, 31,
               "Unable to open device file /dev/dlpi
"));
            exit(1);
    }
}

/* Find the first PPA. */
if (get_ppa_list() == -1)
    exit(1);

/* Was an PPA Number specified? */
if (ppa_str) {


Appendix D 687
/* Yes, convert it to an integer */
temp_ppa = atoi(ppa_str);

/* Try to attach to it. */
if (attach_ppa(temp_ppa) == -1) {
    fprintf(stderr, catgets(nls_linkloop, NL_SETN, 34,
        "No valid interface associated with PPA Number %s\n"),
        ppa_str);
    usage_error();
}
} else {
    /* No, try to attach to default PPA Number. */
    if (attach_ppa(default_ppa) == -1) {
        fprintf(stderr, catgets(nls_linkloop, NL_SETN, 35,
            "No valid interface associated with default PPA Number %d\n"),
            default_ppa);
        exit(1);
    }
}

/*
 * Check option parameters.
 */
if (count_str) { /* count is specified */
    if (!toint(count_str, MAX_INT, &count) || count < 0) {
        fprintf(stderr, catgets(nls_linkloop, NL_SETN, 25,
            "linkloop: illegal count parameter '%s\n"), count_str);
        opterr = TRUE;
    }
if (timeout_str) { /* timeout is specified */
if (!toint(timeout_str, MAX_INT/1000, &timeout) || timeout < 0) {
    fprintf(stderr, catgets(nls_linkloop, NL_SETN, 27,
"linkloop: illegal timeout parameter "\%s\ n"), timeout_str);
    opterr = TRUE;
}
}

/* Get the maximum MTU size */
get_maxmtusize();

if (size_str) { /* size is specified */
if (!toint(size_str, maxmtusize, &mtusize) || mtusize < 0) {
    fprintf(stderr, catgets(nls_linkloop, NL_SETN, 28,
"linkloop: illegal size parameter "\%s\ n"), size_str);
    fprintf(stderr, catgets(nls_linkloop, NL_SETN, 40,
"Note: Maximum packet size for this link: %d\ n"),
maxmtusize);
    opterr = TRUE;
}
}

if (ppa_info.dl_mac_type == DL_TPR) {
if (rif_str) { /* rif is specified */
    /* get_rif_value will obtain value and log it */
    if (get_rif_value(rif_str))
opterr = TRUE;
#else /* set up a default rif */
   riftmp[0] = 2;/* broadcast 0:3 = 0 nobroadcast */
/* length 3:5 = 2 #bytes in rif */
   riftmp[1] = 0x30;/* direction 0:1 = 0 left to right */
/* longest frame 1:3 = 34472 info bytes */
/* Reserved 4:3 = 0 reserved */
#endif
#else
if (rif_str) { /* rif is specified on a non-token ring link??*/
   fprintf(stderr, catgets(nls_linkloop, NL_SETN, 43,
                "Note : rif is valid for 802.5 links only\n"));
   fprintf(stderr, catgets(nls_linkloop, NL_SETN, 44,
                " Parameter will be ignored\n"));
}
#endif

if (opterr)
usage_error();

/*
 * Now we have to bind to an IEEESAP. We will ask for connectionless
 * data link service with the DL_CLDLS service mode. Since we are
 * connectionless, we will not have any incoming connections so we
 * set max_conind to 0. bind() will return our local DLSAP and its
 * length in the last two arguments we pass to it.
 */
bind(SEND_SAP, 0, DL_CLDLS, dlsap, &dlsap_len);
/*
 * set up signals (only catch the foreground signals)
 */
isr_print = FALSE;

if (signal(SIGINT, SIG_IGN) != SIG_IGN)
    signal(SIGINT, myintr);

    if (ppa_info.dl_mac_type == DL_FC)
        addrsize = FCADDRSIZE;
    else
        addrsize = ADDRSIZE;

for (i = optind; i < argc; i++)
    loopback(argv[i], count, mtusize, verbose, ppa_info.dl_phys_addr, txbuf);
exit(0);
} /* end main() */

/***************************************************************************/
read the rif address for 802.5
/***************************************************************************/
get_rif_value(rif)
char *rif;
{
    char *cp = rif;
    char *cpp;
    int i, j, p, q[MAX_RIF_SIZE_U], riflen, tlen, n, error;

Appendix D
for (i=0; i < MAX_RIF_SIZE; i++)
    riftmp[i] = 0;

tlen = strlen(rif);
    riflen = 0;
while ((tlen >= 0) && (riflen < MAX_RIF_SIZE_U)) {
    /* test for valid character e.g 8s should be rejected */
    if (error = check_char(rif, tlen)) {
        fprintf(stderr, catgets(nls_linkloop, NL_SETN, 45,
                "linkloop: invalid rif parameter %s\n"), cp);
        return(1);
    }

        n = sscanf(rif, "%x:\", &o[riflen]);
    else {
        if (tlen > 2) {
            fprintf(stderr, catgets(nls_linkloop, NL_SETN, 45,
                    "linkloop: invalid rif parameter %s\n"), cp);
            return(1);
        } else
            n = sscanf(rif, "%x", &o[riflen]);
    }

bad:    if (n == 1)
            ;
else {

fprintf(stderr, catgets(nls_linkloop, NL_SETN, 45, "linkloop: invalid rif parameter \"%s\n\", cp);
return(1);
}

    /* advance the scanning */
    if (rif[1] == ':') {
        rif = rif+2;
        tlen = tlen-2;
    } else {
        rif = rif+3;
        tlen = tlen-3;
    }
    riflen++;
}  /* end while */

    if ((riflen == MAX_RIF_SIZE_U) && (tlen > 0)) {
        fprintf(stderr, catgets(nls_linkloop, NL_SETN, 46, "linkloop: rif parameter too long \"%s\n\", cp);
return(1);
}

    if (riflen & 0x1) {
        fprintf(stderr, catgets(nls_linkloop, NL_SETN, 47, "linkloop: rif parameter length must be even \"%s\n\", cp);
return(1);
    }
Shared Library Examples for the driveradmin and lanscan Commands

The enetlinkloop’s sample code

```c
for (i=2, j=0; i <= (riflen+2); i++, j++)
    riftmp[i] = o[j];

    riftmp[0] = riflen+2; /* broadcast 0:3 = 0 nobroadcast */
/* length 3:5 = len+2 # bytes in rif */
    riftmp[1] = 0x30; /* direction 0:1 = 0 left to right */
/* longest frame 1:3 = 3 4472 info bytes */
/* Reserved 4:3 = 0 reserved */
    return (0);
} /* end get_rif_value() */
```

/test for valid character e.g 8s should be rejected

```
check_char(rif,len)
char *rif;
int len;
{
    char *cpp;
    int n, p;

    if (len > 2) {
        if (rif[1] == ':')
            cpp = rif;
        else
            if (rif[2] == ':')
                cpp = (rif + 1);
```

---

694 Appendix D
else
    return(1);
} else { /* last 2 hexadecimal bytes */
if (len == 1)
    cpp = rif;
else
    cpp = (rif + 1);
}

n = sscanf(cpp, "%x:", &p);
if (n != 1)
    return(1);

return(0);
} /* end check_char() */

*/

**************************************************************************

nls_start()
**************************************************************************

void
#define CATNAME "linkloop"

nls_start()
{
    if (!setlocale(LC_ALL,"")) {
        fprintf(stderr, "\n"
            "Warning! One or more of your selected locales are not available.\n"
            "Use the command ‘locale’ to verify your selections, and\n"
            "use the command ‘locale -a’ to review the available locales.\n"
        )
        return(1);
    }

    if (len == 1)
        cpp = rif;
    else
        cpp = (rif + 1);

    n = sscanf(cpp, "%x:", &p);
    if (n != 1)
        return(1);

    return(0);
} /* end check_char() */
continuing processing using the "C" locale.

nls_linkloop = (nl_catd)-1;
else
nls_linkloop = catopen(CATNAME,0);
return;

} /* end nls_start() */

get_maxmtusize()

get_maxmtusize()
{
/*
 * Set mtu size using the value in ppa_info
 * For ethernet this caused a problem since dl_mtu is set to 1500 as given
 * by driver but dlpi uses IEEE8023_MTU (1497).
 * Need to clean up dlpi later.
 */
 switch (ppa_info.dl_mac_type) {
   case DL_CSMACD:
   case DL_ETHER:
maxmtusize = MIN(MAX_802_3, ppa_info.dl_mtu);
break;
   case DL_TPR:
/*maxmtusize = MIN(MAX_802_5, ppa_info.dl_mtu); */
maxmtusize = ppa_info.dl_mtu;
break;
    case DL_FDDI:
        maxmtusize = MIN(MAX_FDDI, ppa_info.dl_mtu);
        break;
    case DL_FC:
        default:
            maxmtusize = MIN(MAX_FC, ppa_info.dl_mtu);
            break;
    }
    mtusize = maxmtusize;
} /* get_maxmtusize() */

bind to a sap with a specified service mode and max_conind;
returns the local DLSAP and its length
**************************************************************************
bind(sap, max_conind, service_mode, dlsap, dlsap_len)
int sap;/* 802.2 SAP to bind on */
int max_conind;/* max # of connect indications to accept */
int service_mode;/* either DL_CODLS or DL_CLDLS */
unsigned char* dlsap;/* return DLSAP */
int* dlsap_len;/* return length of dlsap */
{
    dl_bind_req_t* bind_req = (dl_bind_req_t*)ctrl_area;
    dl_bind_ack_t* bind_ack = (dl_bind_ack_t*)ctrl_area;
    unsigned char* dlsap_addr;
/* fill in the BIND_REQ */
bind_req->dl_primitive = DL_BIND_REQ;
bind_req->dl_sap = sap;
bind_req->dl_max_conind = max_conind;
bind_req->dl_service_mode = service_mode;
bind_req->dl_conn_mgmt = 0;/* conn_mgmt is NOT supported */
bind_req->dl_xidtest_flg = 0;/* Auto response to TEST & XID pkts */

/* send the BIND_REQ and wait for the OK_ACK */
put_ctrl(fd, sizeof(dl_bind_req_t), 0);
get_msg(fd);
check_ctrl(DL_BIND_ACK);

/* return the DLSAP to the caller */
*dlsap_len = bind_ack->dl_addr_length;
dlsap_addr = (u_char *)ctrl_area + bind_ack->dl_addr_offset;
memcpy(dlsap, dlsap_addr, *dlsap_len);
} /* end bind() */

**********************************************************************
get the next message from a stream
**********************************************************************/

int
get_msg(fd)
intfd;/* file descriptor */
{

int flags = 0; /* 0 ---> get any available message */

/* zero first byte of control area so the caller can call check_ctrl */
/* without checking the get_msg return value; if there was only data */
/* in the message and the user was expecting control or control + data, */
/* then when he calls check_ctrl it will compare the expected primitive */
/* zero and print information about the primitive that it got. */
ctrl_area[0] = 0;

/* call getmsg and check for an error */
if (getmsg(fd, &ctrl_buf, &data_buf, &flags) < 0) {
    fprintf(stderr, (catgets(nls_linkloop,NL_SETN,71,
      "error: get_msg getmsg failed, errno = %d\n")), errno);
    return(-1);
}
return(1);

/**********************************************************
get the next message from a stream; get_msg2() returns one of the
following defines
*************************************************************************/
#define GOT_CTRL1 /* message has only a control part */
#define GOT_DATA2 /* message has only a data part */
#define GOT_BOTH3 /* message has both control and data parts */

******************************************************************
The enetlinkloop's sample code
int get_msg2(fd)
int fd;/* file descriptor */
{
    int flags = 0;/* 0 ---> get any available message */
    int result = 0;/* return value */

    /*
    * zero first byte of control area so the caller can call check_ctrl
    * without checking the get_msg return value; if there was only data
    * in the message and the user was expecting control or control + data,
    * then when he calls check_ctrl it will compare the expected primitive
    * zero and print information about the primitive that it got.
    */
    ctrl_area[0] = 0;

    /* call getmsg and check for an error */
    alarm(timeout);/* Start read timeout */
    if (getmsg(fd, &ctrl_buf, &data_buf, &flags) < 0) {
        fprintf(stderr, (catgets(nls_linkloop,NL_SETN,72,
                        "error:  get_msg2 getmsg failed, errno = %d\n")), errno);
        /* Did a timeout occur? */
        if (errno == EINTR) /* Yes */
            return (EINTR);
        else
            return(-1);
    }
alarm(0); /* Stop read timeout */

if (ctrl_buf.len > 0) {
    result |= GOT_CTRL;
}
if (data_buf.len > 0) {
    result |= GOT_DATA;
}
return(result);
} /* end get_msg2() */

/******************************************************************
check that control message is the expected message
******************************************************************/
int
check_ctrl(ex_prim)
int ex_prim;/* the expected primitive */
{
dl_error_ack_t*err_ack = (dl_error_ack_t *)ctrl_area;

    /* did we get the expected primitive? */
    if (err_ack->dl_primitive != ex_prim) {
        /* did we get a control part */
        if (ctrl_buf.len) {
            /* yup; is it an ERROR_ACK? */
            if (err_ack->dl_primitive == DL_ERROR_ACK) {
                /* yup; format the ERROR_ACK info */

Appendix D
Shared Library Examples for the driveradmin and lanscan Commands

The enetlinkloop's sample code

```c
fprintf(stderr, (catgets(nls_linkloop,NL_SETN,76,
"error: expected primitive 0x%02x, "), ex_prim));
fprintf(stderr, (catgets(nls_linkloop,NL_SETN,77,
"got DL_ERROR_ACK\n")));
fprintf(stderr, (catgets(nls_linkloop,NL_SETN,78,
" dl_error_primitive = 0x%02x\n")),
err_ack->dl_error_primitive);
fprintf(stderr, (catgets(nls_linkloop,NL_SETN,79,
" dl_errno = 0x%02x\n")), err_ack->dl_errno);
fprintf(stderr, (catgets(nls_linkloop,NL_SETN,80,
" dl_unix_errno = %d\n")), err_ack->dl_unix_errno);
return(-1);
} else {
/*
 didn't get an ERROR_ACK either; print whatever
 primitive we did get
 */
fprintf(stderr, (catgets(nls_linkloop,NL_SETN,76,
"error: expected primitive 0x%02x, ")), ex_prim);
fprintf(stderr, (catgets(nls_linkloop,NL_SETN,81,
" got primitive 0x%02x\n")), err_ack->dl_primitive);
return(-1);
}
} else {
/* no control; did we get data */
if (data_buf.len) {
/* tell user we only got data */
fprintf(stderr, (catgets(nls_linkloop,NL_SETN,82,
"error: check_ctrl found only data\n")));
```
return(-1);
  } else {
/* no message???: well, it was probably an
 * interrupted system call */
  fprintf(stderr, (catgets(nls_linkloop,NL_SETN,83,
  "error: check_ctrl found no message\n")));
  return(-1);
  } /* end else */
} /* end else */
} /* end if */
} /* end check_ctrl() */

/**************************************************************************/
put a message consisting of only a control part on a stream
**************************************************************************/
int
put_ctrl(fd, length, pri)
intfd;/* file descriptor */
intlength;/* length of control message */
intpri;/* priority of message: either 0 or RS_HIPRI */
{
  /* set the len field in the strbuf structure */
  ctrl_buf.len = length;

  /* call putmsg and check for an error */
  if (putmsg(fd, &ctrl_buf, 0, pri) < 0) {
    fprintf(stderr, (catgets(nls_linkloop,NL_SETN,74,
    "error: put_ctrl putmsg failed, errno = \%d\n")), errno);
Shared Library Examples for the `driveradmin` and `lanscan` Commands

The `enetlinkloop`'s sample code

```c
return(-1);
}
} /* end put_ctrl() */

/**************************************************************************
put a message consisting of both a control part and a control part
on a stream
**************************************************************************/

int
put_both(fd, ctrl_length, data_length, pri)
    int fd; /* file descriptor */
    int ctrl_length; /* length of control part */
    int data_length; /* length of data part */
    int pri; /* priority of message: either 0 or RS_HIPRI */
{
    /* set the len fields in the strbuf structures */
    ctrl_buf.len = ctrl_length;
    data_buf.len = data_length;

    /* call putmsg and check for an error */
    if (putmsg(fd, &ctrl_buf, &data_buf, pri) < 0) {
        fprintf(stderr, (catgets(nls_linkloop,NL_SETN,75,
            "error: put_both putmsg failed, errno = %d\n")), errno);
        return(-1);
    }
} /* end put_both() */

/**************************************************************************

Appendix D
Get the list of available PPAs.

```c
int get_ppa_list()
{
    dl_hp_ppa_req_t *ppa_req = (dl_attach_req_t *)ctrl_area;
    dl_hp_ppa_ack_t *ppa_ack = (dl_hp_ppa_ack_t *)ctrl_area;
    dl_hp_ppa_info_t *ppa_info_temp;

    /* find a PPA to attach to; we assume that the first PPA on the
     * remote is on the same media as the first local PPA
     */

    /* send a PPA_REQ and wait for the PPA_ACK */
    ppa_req->dl_primitive = DL_HP_PPA_REQ;
    if (put_ctrl(fd, sizeof(dl_hp_ppa_req_t), 0) == -1)
        return(-1);
    if (get_msg(fd) == -1)
        return(-1);
    if (check_ctrl(DL_HP_PPA_ACK) == -1)
        return(-1);
    /* make sure we found at least one PPA */
    if (ppa_ack->dl_length == 0) {
        fprintf(stderr, (catgets(nls_linkloop, NL_SETN, 84,
            "error: no PPAs available\n")));
        return(-1);
    }
}
```
Shared Library Examples for the driveradmin and lanscan Commands
The enetlinkloop’s sample code

/* Save all the PPA information. */
memcpy((u_char *)ppa_area, (u_char *)ctrl_area+ppa_ack->dl_offset, ppa_ack->dl_length);

ppa_count = ppa_ack->dl_count;

/* examine the first PPA */
ppa_info_temp = (dl_hp_ppa_info_t *) ppa_area;
default_ppa = ppa_info_temp->dl_ppa;

return(0);
} /* end get_ppa_list() */

/***************************************************************************/
attach to the current PPA
******************************************************************************/

int
attach_ppa(ppa)
int ppa;
{
    dl_attach_req_t *attach_req = (dl_attach_req_t *)ctrl_area;
dl_hp_ppa_info_t *ppa_info_temp;
    int count;
    int found;

    /* See if this ppa is even in list */
    for (count = found = 0, ppa_info_temp = ppa_area; count < ppa_count; count++, ppa_info_temp++) {

    } /* end attach_ppa() */
if (ppa == ppa_info_temp->dl_ppa /*&&
    !strncmp(ppa_info_temp->dl_module_id_1, "lan", 3)*/) {
    found = TRUE;
    break;
} /* end if */
} /* end for */

if (!found) {
    /* */
    fprintf(stderr,"attach: ppa not found in ppa_area\n");
    /* */
    return(-1);
}

/*
 * fill in ATTACH_REQ with the PPA we found, send the
 * ATTACH_REQ,
 * and wait for the OK_ACK
 */
attach_req->dl_primitive = DL_ATTACH_REQ;
attach_req->dl_ppa = ppa;
if (put_ctrl(fd, sizeof(dl_attach_req_t), 0) == -1) {
    /* */
    fprintf(stderr,"attach: put_ctrl failed\n");
    /* */
    return(-1);
}
if (get_msg(fd) == -1) {
    /* */
}
Shared Library Examples for the driveradmin and lanscan Commands

The enetlinkloop's sample code

```c
fprintf(stderr,"attach: get_msg failed\n");
/* */
return(-1);
}
if (check_ctrl(DL_OK_ACK) == -1) {
/* */
    fprintf(stderr,"attach: check_ctrl failed\n");
/* */
return(-1);
}
memcpy(&ppa_info, ppa_info_temp, sizeof(ppa_info));

return(0);
} /* end attach_ppa() */

void
timeout_hdlr()
{
    recv_timeout++;
} /* end timeout_hdlr() */

usage_error()
{
    fprintf(stderr, catgets(nls_linkloop, NL_SETN, 29,
"Usage: linkloop \[ -n count \] \[ -i PPA Number \] \[ -t timeout \] \n\[ -s size \] \[ -r rif \] \[ -v \] \[ -3 thirdp_filename\] linkaddr ...
\n\n"));
    fprintf(stderr, catgets(nls_linkloop, NL_SETN, 30,
" Usage: linkloop \[ -n count \] \[ -i PPA Number \] \[ -t timeout \] \n\[ -s size \] \[ -r rif \] \[ -v \] \[ -3 thirdp_filename\] linkaddr ...
\n\n"));
    exit(1);
```
Appendix D

709

Shared Library Examples for the driveradmin and lanscan Commands

The enetlinkloop's sample code

} /* end usage_error() */

/************************************************************
 * printresults -- print out the current status of recv and sent
 ************************************************************/

void
printresults()
{
    printf ((catgets (nls_linkloop, NL_SETN, 10, " frames sent : 
        %d\ n")), sent);
    printf ((catgets (nls_linkloop, NL_SETN, 11, " frames received
        correctly : %d\ n")), recv_ok);
    if (recv_bad_len > 0)
        printf ((catgets (nls_linkloop, NL_SETN, 12, " frames with length
            error  : %d\ n")), recv_bad_len);
    if (recv_bad_data > 0)
        printf ((catgets (nls_linkloop, NL_SETN, 13, " frames with data
            error    : %d\ n")), recv_bad_data);
    if (recv_bad_header > 0)
        printf ((catgets (nls_linkloop, NL_SETN, 14, " reads that timed out
            : %d\ n")), recv_bad_header);
    if (recv_timeout > 0)
        printf ((catgets (nls_linkloop, NL_SETN, 15, " reads that timed out
            : %d\ n")), recv_timeout);
}

/*****************************************
 * myintr -- this will be called when an interrupt signal occurs
 ******************************************/

int
myintr()
{
    printf(catgets(nls_linkloop, NL_SETN, 37, \" n\"));
    if (isr_print)
        printresults();
    fflush(stdout);
    exit(0);
}

/******************************************************************
* compare -- byte by byte comparison of the contents of two buffer
*buf1
*buf2 -- the buffers
*size -- the size of the buffers
*******************************************************************/
int compare(buf1, buf2, size)
unsigned char *buf1;
unsigned char *buf2;
int  size;
{
    while (size-- >0)
    if (*buf1++ != *buf2++)
        return(FALSE);

    return(TRUE);
}
toint -- converts from string to integer, same as atoi except:
* checks for invalid characters
* checks for overflow
*
* this function will return:
* FALSE -- overflow or illegal integer (i will be undefined)
* TRUE -- conversion worked

int
toint(s, max, i)
char *s;/* char string pointing to ascii integer (hopefully) */
int max;/* maximum integer wanted */
int *i;/* converted integer */
{
    int sign;
    int sum;

    /* skip leading blanks and tabs */
    while (*s != '\0' & (s = ' ' | *s == 't'))
        s++;

    /* check for sign bit */
    if (*s == '-') {
        sign = -1;
        s++;
    } else if (*s == '+') {
        sign = 1;

Appendix D 711
Shared Library Examples for the driveradmin and lanscan Commands

The enetlinkloop’s sample code

```c
s++;
    } else {
sign = 1;
    }

sum = 0;
do {
if (!isdigit(*s))
    return(FALSE);
sum = sum * 10 + TODIGIT(*s);
if (sum < 0 || sum > max) /* check for overflow */
    return(FALSE);
s++;
    } while (*s != '\0' && *s != ' ' && *s != '\t');

/* skip trailing blanks and tabs */
    while (*s != '\0' && (*s == ' ' || *s == '\t')) {
s++;
    }

    if (*s == '\0') {
*i = sum * sign;
return(TRUE);
    } else
return(FALSE);
} /* end toint() */
```

******************************************************************
send_test_req()
**************************************************************************

send_test_req(hisaddr, txbuf, size)
unsigned char *hisaddr;
unsigned char *txbuf;
int size;
{

dl_test_req_t *test_req = (dl_test_req_t *)ctrl_area;
intrlen = 0;
intfclen = 0;
unsigned char *sap_p;
unsigned char *fctype_p;

  /* If 802.5, factor in RIF len */
  if (ppa_info.dl_mac_type == DL_TPR)
    rlen = riftmp[0];

  if (ppa_info.dl_mac_type == DL_FC)
    fden = 1;

test_req->dl_primitive = DL_TEST_REQ;
test_req->dl_dest_addr_length = addrsize + rlen + fden + 1;
test_req->dl_dest_addr_offset = sizeof(dl_test_req_t);
memcpy((char *)ctrl_area + test_req->dl_dest_addr_offset, hisaddr, addrsize);

  /* If Fibre Channel copy in type field info */
  if (ppa_info.dl_mac_type == DL_FC)
Shared Library Examples for the driveradmin and lanscan Commands

The enetlinkloop's sample code

```c
{
    fctype_p = (char *)ctrl_area + test_req->dl_dest_addr_offset + addsize;
    *fctype_p = FC_TYPE4;
}

/* Copy in the SAP */
    sap_p = (char*)ctrl_area + test_req->
        dl_dest_addr_offset + addsize + fclen;
    *sap_p = 0;

    /* If 802.5, copy in RIF info */
    if (ppa_info.dl_mac_type == DL_TPR)
        memcpy((char *)ctrl_area + test_req->
            dl_dest_addr_offset + addsize + 1,
            riftmp, rlen);

        memcpy((char *)data_area, txbuf, size);
        put_both(fd, sizeof(dl_test_req_t) + addsize + rlen + fclen + 1, size, 0);
}

/******************************************************************
loopback -- run a loopback test to a specific LAN station
******************************************************************/

loopback(destaddr, count, size, verbose, myaddr, txbuf)
    u_char *destaddr;/* the ascii representation of the destination */
    int  count;/* how many times to loop (0 is infinite) */
    int  size;/* size of buffer to transmit */
```
int verbose;/* whether to print specific errors or not */
int j;/* indexes through count */

u_char *myaddr;/* local node addr (used to compare with response) */

u_char *txbuf;/* the transmit data */

u_char dest_addr[ADDRSIZE];

u_char src_addr[ADDRSIZE];

u_char hisaddr[ADDRSIZE];/* the remote node's link level address */

u_char hexaddr[ADDRSIZE*2+3];/* contains hex formatted vers of link addr */

int header_error;/* a receive is repeated if an hdr err occur*/

int printed;/* boolean/if the OK/FAILED msg already prntd */

int msg_res;
dl_test_con_t *test_con = (dl_test_con_t *)ctrl_area;

printed = FALSE;

printf((catgets(nls_linkloop, NL_SETN, 16, "Link connectivity to LAN
station: %s\ n")), destaddr);

fflush(stdout);

if (net_aton(hisaddr, destaddr, addrsize) == NULL) {
  fprintf(stderr, catgets(nls_linkloop, NL_SETN, 17, " -- FAILED\ n"));
  fprintf(stderr, catgets(nls_linkloop, NL_SETN, 18, "    Address has
bad format\ n"));

if (ppa_info.dl_mac_type == DL_FC)
  printf((catgets(nls_linkloop, NL_SETN, 50, "Note: For Fibre
Channel the 3 byte N_port address identifier needs to
be\ n specified. Use lanscan/landiag on the remote node
to\ n determine the N_port ID.\ n"));

fflush(stdout);
return;
}

if (ppa_info.dl_mac_type == DL_TPR) {
if (!TOKINDIVIDUAL(hisaddr)) {
    fprintf(stderr, catgets(nls_linkloop, NL_SETN, 17, " -- FAILED\n"));
    fprintf(stderr, catgets(nls_linkloop, NL_SETN, 19, " Address is not individual\n"));
    fflush(stdout);
    return;
}
} else {
if (ppa_info.dl_mac_type != DL_FC) {
    if (!INDIVIDUAL(hisaddr)) {
        fprintf(stderr, catgets(nls_linkloop, NL_SETN, 17, " -- FAILED\n"));
        fprintf(stderr, catgets(nls_linkloop, NL_SETN, 19, " Address is not individual\n"));
        fflush(stdout);
        return;
    }
}
}

recv_ok = 0;
recv_bad_data = 0;
recv_bad_len = 0;
recv_timeout = 0;
Shared Library Examples for the driveradmin and lanscan Commands
The enetlinkloop’s sample code
recv_bad_header = 0;
sent = 0;

if (sigset(SIGALRM, timeout_hdlr) == BADSIG) {
fprintf(stderr, catgets(nls_linkloop, NL_SETN, 70,
"Could not set SIGALRM %s\n"), sys_errlist[errno]);
exit(1);
}

isr_print = TRUE;
for (j = 0; j < count || count == 0; j++) {
send_test_req(hisaddr, txbuf, size);
sent++;

do {
header_error = FALSE;
msg_res = get_msg2(fd);
/* Did a timeout occur? */
if (msg_res == EINTR)
/* Yes */
continue;

check_ctrl(DL_TEST_CON);
if (msg_res == GOT_BOTH) {
memcpy(dest_addr,
(char *)ctrl_area+test_con->dl_dest_addr_offset,
addrsize);
memcpy(src_addr,

Appendix D

717


Shared Library Examples for the driveradmin and lanscan Commands

The enetlinkloop's sample code

```c
(char *)ctrl_area+test_con->dl_src_addr_offset,
   addrszie);

/* If Token Ring, mask out the RIF bit */
if (ppa_info.dl_mac_type == DL_TPR)
   src_add[0] -= 0x7f;

if (!compare(myaddr, dest_addr, addrszie)) {
   if (verbose) {
      if (!printed) {
         printf((catgets(nls_linkloop, NL_SETN, 17, " --
                 FAILED
"));
         printed = TRUE;
      }
      printf((catgets(nls_linkloop, NL_SETN, 20,
         " %d: bad destination address %s\n")),
         j, net_ntoa(hexaddr, dest_addr, addrszie));
      } recv_bad_header++;
      header_error = TRUE;
   }
   else if (!compare(hisaddr, src_addr, addrszie)) {
      if (verbose) {
         if (!printed) {
            printf((catgets(nls_linkloop, NL_SETN, 17, " --
                   FAILED\n"));
            printed = TRUE;
         }
      } printf((catgets(nls_linkloop, NL_SETN, 21,
```

Appendix D
"    %d: bad source address %s\n"), j,
net_ntoa(hexaddr, src_addr, addrsz);
}
recv_bad_header++;
header_error = TRUE;

} else if (data_buf.len != size) {
if (verbose) {
if (!printed) {
    printf(catgets(nls_linkloop, NL_SETN, 17, "--
FAILED\n"));
    printed = TRUE;
}
printf((catgets(nls_linkloop, NL_SETN, 23,"    %d: bad
length %d\n")), j, data_buf.len);
}
recv_bad_len++;

} else if (!compare(txbuf, data_area, size)) {
    recv_bad_data++;
} else {
    recv_ok++;
}

} else {
    fprintf(stderr, catgets(nls_linkloop, NL_SETN, 38, "error - did
not receive data part of message\n"));
exit(1);
}
Shared Library Examples for the driveradmin and lanscan Commands

The enetlinkloop's sample code

```c
while (header_error);
} /* end for */

if (!printed) {
if (recv_ok == sent) {
  printf(catgets(nls_linkloop, NL_SETN, 24, " -- OK\n"));
} else {
  printf(catgets(nls_linkloop, NL_SETN, 17, " -- FAILED\n"));
  printresults();
}
}
fflush(stdout);
isr_print = FALSE;
} /* end loopback() */

#MAKEFILE for enetlinkloop

LIB =/usr/lib
CP =  /usr/bin/cp
CC =/usr/bin/cc

SRCLOC=/opt/enetlinkloop
LAN_CFLAGS=-WI,-E -I:libdld.sl -DCONVERGED_IO
INCLUDES=I /usr/include

CFLAGS=${LAN_CFLAGS} ${INCLUDES} -Wp,-H300000 -D_PTHREADS_DRAFT4 -L $(LIB)
```

Appendix D
Shared Library Examples for the driveradmin and lanscan Commands

The enetlinkloop’s sample code

LDLFLAGS= +DA2.0W
LINKFLAGS= -lIO -lV3

default all: linkloop

linkloop.o : linkloop.c
$(CC) $(CFLAGS) -c linkloop.c -o linkloop.o

linkloop : linkloop.o
Shared Library Examples for the driveradmin and lanscan Commands

The enetlinkloop's sample code
Glossary

A-B

32-Bit Program A program compiles to run in 32-bit mode. For example, programs compiled for the PA RISC 1.x processors.

64-Bit Program A program compiled to run in 64-bit mode. For example, programs compiled for the PA-RISC 2.0 processor in wide mode.

100BT 100BASE-T is the technical term for the Fast Ethernet or IEEE802.3u standard.

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.

BAR Base Address Register. One of six registers in a PCI device function’s configuration space. Each BAR may specify the base and size of a memory or I/O range for the device.

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

BN-CDIO Bus Nexus CDIO, software that manages platform-dependent bus connection hardware.

Bus Mastering Taking control of a bus and mastering DMA transactions.

Bus Nexus Connection between two buses.

C

Cache Coherence Consistency of data in host memory as viewed by processor caches and I/O devices.

CDIO Context-Dependent I/O module. A module in GIO framework which contains all bus specific and/or driver environment specific functionality.

Class A logical grouping of device or hardware modules by type. For instance the 'class tape' would include all tape devices regardless of bus interface.

Coherent I/O Accesses to data in host memory by I/O devices are consistent with accesses by processor caches. Hardware in the platform maintains the consistent view of data in host memory as DMA transactions flow through
the hardware.

**D-E**

**DLPI** Data Link Provider Interface

**DMA** Direct Memory Access - I/O transactions for which the device interacts directly with memory without processor intervention.

**Driver** Software module which controls a device, interface card or bus-nexus.

**Driver Environment** A defined set of services and entry points which allow a driver to function.

**F-I**

**GIO** General I/O System

**IHV** Independent Hardware Vendor

**ILP32** C language data model where int, long, and pointer data types are 32 bits in size.

**Init List** A linked list of device driver init routines (\<drv\>_init) which is built as the drivers configure themselves and run as the I/O system configuration is completed to perform any device driver-specific initialization.

**Instance** A number assigned to an I/O tree node. The number is unique within a driver class.

**I/O Bus** Interconnect bus for I/O cards and devices. PCI is an example of an I/O bus.

**I/O Node** An element of an I/O tree which includes all relevant information needed for configuring a single hardware module.

**I/O Adapter** Hardware to provide IOVA translation between an I/O bus and the processor/memory interconnect. Devices on the I/O bus issue bus transactions to IOVA memory targets and the I/O adapter translates IOVA memory targets to physical addresses. The I/O adapter also participates in the coherency protocol of the processor caches for platforms that are coherent or semicoherent.

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

**I/O Tree** Data structure for recording the I/O subsystem configuration information.

**IOVA** I/O Virtual Address. Address used by I/O devices to access host memory. Platforms that are semicoherent or coherent, or where the processor/memory
interconnect is greater than 32 bits wide, generally implement IOVAs.

**IP** Internet Protocol.

**ISC** Interface Select Code. Usually used as a pointer to an element of a table of *isc_table_type* structures (one per interface card). Each ISC entry is used by WSIO to maintain interface device driver information.

**ISR** Interrupt Service Routine. A driver-specific routine which handles interrupts from the device.

**ISV** Independent Software Vendor

**J-M**

**LAN** Local Area Network.

**LP64** C language data model where the int data type is 32 bits wide, but long and pointer data types are 64 bits wide.

**MAC** Medium Access Control.

**Map PCI -> 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 -> 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()`.

**MP** Multi-Processor

**MP Safe** Describes a module which is protected in an MP environment through the use of various spinlocks and semaphores. Note that MP-safeness does not imply any performance considerations due to the granularity of the semaphores (e.g., use of a single I/O Empire semaphore or separate semaphores for each instance all imply MP-safeness).

**MP Scalable** Describes an MP module which may add components without causing more drain on other MP modules. An MP-scalable driver will provide a separate spinlock for each instance of the driver. Non MP-scalable drivers may still be MP-safe but perhaps only provide a single semaphore and spinlock for all instances of the driver. Adding more instances of a non
MP-scalable driver will therefore cause additional taxing of those resources for each instance added to the system.

**N-Q**

NIC Network Interface Card.

**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.

**PA** Precision Architecture. When referring to busses, these are busses which conform to the Precision I/O Architecture.

**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 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. Typical cards map their registers into PCI memory space, meaning they can only be accessed by PCI memory cycles.

**Physical Address** Real address by which host memory or an I/O device register is accessed.

**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()`.

**PPA** Physical Point of Attachment

**R-S**

**SAP** Service Attach Point

**SCSI** Small Computer System Interface. An industry standard
external I/O bus available on all HP9000 systems.

**Semicoharent I/O** Similar to coherent I/O. However, for the case of data read from an I/O device, software must synchronize the data that have been read into host memory after the DMA transaction completes.

**Series700** HP9000/7XX family of PA-RISC workstations.

**Series800** HP9000/8XX family of PA-RISC business servers.

**SIO** Server I/O; I/O environment for port-server drivers with origins in S/800 systems.

**SNAP** Sub-Network Access Point

**Spinlock** Basic locking primitive used by the kernel for short-term locks. When a thread acquires a spinlock, the thread’s current processor becomes the effective owner until the spinlock is released. Threads (processors) waiting to acquire an owned spinlock will spin while waiting; they do not block. For the duration that a processor owns a spinlock, external interrupts to the processor are disabled.

**T-Z**

**Virtual Address** Address used by processors, when executing in virtual mode, to access host memory. Address translation hardware converts a virtual address to a physical address before host memory is accessed. Virtual addresses may also be used to map and access I/O device registers.

**WSIO** Workstation and Server I/O; I/O environment for reentrant drivers with origins in S/700 systems and converged with S/800 systems.
Index

Device Driver, RAM Disk Static Module, 441
Device Driver, RAM Disk WSIO DLKM, 455
device drivers, 36
   and major numbers, 39
   kernel association, 39
device file, 182
device special files
   major and minor numbers, 39
Device switch table
   SCSI, 323
devices
   block, 36
   character, 36
   raw, 36
disclaimers
   support/compatibility, 19
DLKM (Dynamically Loadable Kernel Module) feature, 362, 363, 365, 397
DLKM format, 362, 363, 367
DLKM module types, 363, 365, 370, 373
DLKM modules
   defined, 362, 425
   advantages of, 364
   writing, 367–375
   initializing and terminating, 376–386
   support tools for, 387–389
   developing, 390–408
DLPI protocol
   kinds, 510
   primitives, 211, 503
   sequence, 205
DMA (Direct Memory Access), 365, 406
dma.h header file, 435
dmesg command, 409, 410
driver environment, 25
driver fragment skeleton, 52
Driver install routine
   SCSI, 322
Driver set up structures
   SCSI, 319
driver writing, steps, 175
driver_attach(), 196
driver_attach() routine, 101
driver_close()
   declaration, 123
   driver_close() routine, 123
Driver_dev_init() routine
   SCSI, 323
   driver_dev_init() routine, 103
driver_if_init() routine, 104
driver_init(), 199
driver_install(), 194
driver_install() routine, 98
driver_ioctl() routine, 131
driver_isr() routine, 150
driver_minor_build() routine, 117
driver_minphys() routine, 139
driver_open() routine, 119
driver_psize() routine, 152
driver_read() routine, 126
driver_select() routine, 141
driver_strategy(), 146, 147
driver_strategy() routine, 145
driver_write() routine, 126
driveradmin shared library example, 551
Drivers
   pre-11.0, 423
drivers
   adding driver to system file, 166
   adding Series 700 driver to configuration files, 160
   building and installing your kernel, 167
   device, 36
   interface, 36
   interrupt context, 42
   lower half, 42
   pseudo, 36
   two levels, 42
   upper half, 42
Drivers, example, 439
dynamically assigned major numbers, 39
dynamically loadable modules
   defined, 362, 425
E
   encapsulation method, 502
   enetlinkloop, 678
Entry point routines
   SCSI, 328
Error handling
   SCSI, 331
Ethernet protocol, TYPE values, 512
example
   I/O read request, 44
Example drivers, 439
Examples
   Shared Library for driveradmin, 551
   Shared Library for lanscan, 551
exclusive open, 120
extracting the system file, 166

730
Index

F
figures
   Call Graph of SCSI Routines and Services, 332
   Driver Configurations, 30
   WSIO CDIO as a buffer zone, 30
file
   kernel configuration, 170
filter_packet(), 527
format_link_nice(), 528
format_link_raw(), 530
formatting messages, 271

G
General I/O system (GIO), 27

H
handling interrupts, 51
header file
   conf.h, 435
dma.h, 435
uia.h, 436
header files
device classes, 317
geneneral, 317
SCSI, 317
hp_dlpi_wakeup(), 209
HP-UX system file
defined, 363
hw_dlpi structure
description, 188
introduction, 184
hw_if structure
description, 184
encapsulation method, 502
introduction, 184
MAC type, 502
protection, 189
protocol type, 502

I
I/O
   flow of a request, 43
   kernel data structures, 46
   system calls, 42
I/O subsystem structure, 27
ID number
   subsystem, N, 258
IEEE 802.2 LLC use, SAP values, 511

IKON PCI Monolithic Device Driver, 470
ILP32 data models, 426
initializing and terminating DLKM modules, 376–386
insf command, 405
installing your Series 700 kernel, 167
int32_t, 430
int64_t, 431
intended audience, 18
interface drivers, 36, 150
interface select code table, 46
Interrupt Control Stack (ICS), 65
interrupt service routine, 150
Interrupt Service Routine (ISR), 67
interrupts
   handling, 51
intptr_t, 430
ioctl command word, 132
iovec structure, 48

K
kernel and drivers association, 39
kernel configuration file, 170
kernel configuration tool set, 387–389
   config, 387
   kmadmin, 388
   kmininstall, 388
   kmsystem, 389
   kmtune, 389
   kmupdate, 389
kernel data structures, 46
kernel module
defined, 365
kget_log_instance(), 267
KLOG_CK(), 267
klogg_write(), 268
kmadmin command
defined, 388
   using the -k option, 403
   using the -L option, 401
   using the -Q option, 402
   using the -q option, 402
   using the -S option, 403
   using the -s option, 403
   using the -U option, 406
   using the -u option, 406
kmininstall command
defined, 388
   using the -a option, 395
   using the -d option, 408
   using the -u option, 407
Index

kmsystem command
   defined, 389
   using the -c and -l options, 396
kmtune command
   defined, 389
   using the -q option, 396
   using the -r option, 396
   using the -s option, 396, 405, 410
kmupdate command
   defined, 389
   using the -M and -a options, 400
   using the -M and -i options, 400
ktrc_write(), 264

L
lanadmin
   description, 246
lanscan
   description, 249
lanscan shared library example, 551
load function
   defined, 371
   example of WSIO class driver _load function, 376
   example of WSIO interface driver _load function, 381
   example of STREAMS driver _load function, 384
   example of STREAMS module _load function, 385
loadable modules
   defined, 362, 425
Lock acquisition rules, 69
Lock orders, 69
SCSI, 353
logging/tracing
   assigning subsystem ID, 258
   classifying log data, 260
   classifying trace data, 258
   configuring developed subsystems, 545
   formatting trace data, 259
   internationalization, 544
   makefiles, 262
   message catalog, 544
   overview, 253
   source files, 262
   subsystem installation testing, 549
   what and when to log, 261
LP64 considerations, 427
LP64 data models, 426

lsdev command, 404

M
MAC types, 502
major and minor numbers
   in device special files, 39
major numbers
   and device drivers, 39
   defined, 39
   dynamic assignment and ranges, 39
make command, 410
master configuration file
   defined, 395
master file, 363, 372, 417
mechanism
   software trigger, 51
Mechanisms synchronization, 66
MIB II
   extended structure, 499
   standard structure, 499
MIB, event message, 504
minor numbers
   defined, 40
miscellaneous modules, DLKM, 364, 386
mkdir command, 410
mknod command, 403, 405
mod.o file, 368–372
   module wrapper, 368
   load function, 371
   unload function, 371
Modstub.o file, 374
module component files, DLKM
   mod.o, 368
   master, 372
   system, 373
   space.h, 374
   Modstub.o, 374
module packaging
   11.0 versus pre-11.0, 363
module type
   defined, 365
module wrapper, 368–371
   structures of, 369
   example of, 370
module's wrapper, 368, 401
   see also module wrapper
modules
   DLKM, 362, 425
   see also DLKM modules
dynamically loadable, 362, 425
Index

loadable, 362, 425
static, 362, 425
see also static modules
statically linked, 363
see also kernel module
modwrapper structure
defined, 365
code definition of, 368
elements of, 369
MTU values, 505
multiple open, 120
Multiprocessing, 63
Idle Time, 66
Timing hazards, 66
Multiprocessing to Uniprocessing
Comparison, 65
Multiprocessor implications
SCSI, 327

N
N, subsystem ID number, 258
Native Language Support (NLS), 544
netfmt(1M), 256, 271
nett1(1M), 256
nett1conf(1M), 256
netttgen.conf(4), 256
network interface, architecture, 179
network management
SNMP, 240
network protocol IP layer, 182
networking device driver
driver_attach(), 196
driver_init(), 199
driver_install(), 194
initializing, 191, 193
structure, 175

P
packet
format, 202
headers, 506
types, 509
PCI
Automatic IRQ determination, 292
Blocking transactions, 285
Bus-dependent implementation, 300
Bus-independent functionality, 300
byte swapping, 288
Configuration space restrictions, 293
Configuration Spaces, 283
 Constants and data structures, 304
Device operation, 295
Device setup, 290
DMA, 296
driver entry point routines, 313
Endian issues, 288
I/O base registers, 292
I/O space, 284
Leveraging existing drivers, 299
Mapping base address registers into I/O
space, 290
Mapping base address registers into PCI
memory, 290
Mapping the memory base register, 292
Mastered reads and writes to host memory,
286
Masters and Coherence, 298
Memory base registers, 291
Multiprocessor safety, 303
Nonblocking transactions, 285
PCI_LITTLE_ENDIAN_ONLY flag, 295
pre-swapping, 289
Processor and card mastered reads and
writes to host memory, 286
Processor mastered reads and writes, 285
Register Spaces, 283
sample driver_attach(), 309
sample driver_install(), 308
Transaction ordering, 284
Unsupported WSIO functions, 302
Using base address registers, 290
Write side-effects, 285
PCI (Peripheral Component Interconnect)
defined, 365
PCI Combined User and Kernel Driver, 482
PCI Monolithic Hardcopy Interface Device
Driver, 468
physio(), 128
processing, packets, 201
protocol
binding, 201
demultiplexing, 201, 202
format, 202
interface layer, 182
kinds, 510
types, 502
pseudo drivers, 36
ptr32_t, 431
Index

Q
Q4 debugger, 405

R
RAM Disk Static Module Device Driver, 441
RAM Disk WSIOL DLKM Device Driver, 455
raw devices, 36
references to other material, 22
regenerating kernel, 170
routines
calling physio(), 130
driver_strategy() read, 147
driver_strategy() write, 146
software trigger, 51
writing a driver_attach(), 101
writing a driver_close(), 123
writing a driver_dev_init(), 103
writing a driver_if_init(), 104
writing a driver_install(), 98
writing a driver_ioctl(), 131
writing a driver_isr(), 150
writing a driver_minor_build(), 117
writing a driver_minphys(), 139
writing a driver_open(), 119
writing a driver_psize(), 152
writing a driver_read() routine, 126
writing a driver_select(), 141
writing a driver_strategy(), 145
writing a driver_write() routine, 126
Routines and services
SCSI, 332
Rules for lock acquisition, 69

S
SAP values, 511
SCSI
  Data protection, 353
  Device switch table, 323
  Driver install routine, 322
  Driver_dev_init() routine, 323
  Entry point routines, 328
  Error handling, 331
  Include header files, 317
  Lock and semaphore orders, 353
  Multiprocessor implications, 327
  Routines and services, 332
  Services, 355
  Underlying routines, 332
  SCSI lock
Bus, 353
LUN, 353
Subsystem, 353
Target, 353
select code
  interface, 46
Semaphore orders
  SCSI, 353
Services
  SCSI, 355
set_up_8022(), 525
set_up_ether(), 527
set_up_ip(), 526
set_up_link(), 526
Shared Library Examples, 551
shared open, 120
shutdown command, 402
skeletons
driver fragment, 52
sleep(), 70
SNAP, protocol TYPE values, 512
software trigger mechanism, 51
software trigger routine, 51
space.h file, 374, 418
spinlock
deadlock avoidance, 190
  protection model, 189
Spinlock routines
  overview, 67
Spinlocks, 66
  overview, 67
SSCSI
  Driver set up structures, 319
static modules
  defined, 362, 425
  converting to DLKM format, 367
statically linked modules
  defined, 363
stream
  defined, 366
STREAMS drivers, DLKM, 383–385
  example of _load and _install functions for, 384
  example of _unload function for, 385
streams environment layer, 183
STREAMS head, 182
STREAMS modules, DLKM, 385–386
  example of _load and _install functions for, 385
  example of _unload function for, 386
structure
ui0, 48
structures
buf, 46
iovec, 48
subformatter
calls, 533
designing, 273
functions, 518
option, 543
subsys_N_format(), 522
subsys_N_get_options(), 522
subsystem
ID number, N, 258
support disclaimers, 19
Synchronization
sleep(), 70
wakeup(), 70
Synchronization Mechanisms, 66
system calls, 42
system description file
defined, 395
system file, 373, 418
system file, extracting the, 166

T
terminating DLKM modules
see initializing and terminating DLKM modules
timeout mechanisms, 49
tl_banner_char(), 542
tl_check_cat_version(), 541
tl_format_fprintf(), 537
tl_format_write(), 536
tl_get_line(), 543
tl_get_parms(), 540
tl_header_format1(), 534
tl_log_class(), 542
tl_raw_format(), 538
tl_trace_kind(), 542
tracing/logging
assigning subsystem ID, 258
classifying log data, 260
classifying trace data, 258
configuring developed subsystems, 545
formatting trace data, 259
internationalization, 544
makefiles, 262
message catalog, 544
overview, 253
source files, 262
subformatter code, 515

subsystem installation testing, 549
troubleshooting
alternative means, 274
HP approach, 253
TYPE values
Ethernet, 512
SNAP, 512

U
ui0 structure, 48
ui0.h header file, 436
uiomove(), 130
Underlying routines
SCSI, 332
unint32_t, 430
unint64_t, 431
unintptr_t, 430
Uniprocessing to Multiprocessing comparison, 65
unload function
defined, 371
element of WSIO class driver _unload function, 379
element of WSIO interface driver _unload function, 383
element of STREAMS driver _unload function, 385
element of STREAMS module _unload function, 386
using physio(), 128

W
wakeup(), 70
wrapper code, 367
see also module wrapper
writing DLKM modules, 367-375
module wrapper, 368
load function, 371
unload function, 371
master file, 372
system file, 373
space.h file, 374
Modstub.o file, 374
writing driver, steps, 175
WSIO (Workstation Input/Output)
defined, 366
WSIO class driver, sample DLKM, 409-421
sample component files, 411
sample makefile, 418
WSIO class drivers, DLKM, 376-379
Index

example of _load function for, 376
example of _install function for, 377
example of _init function for, 378
example of _unload function for, 379
WSIO interface drivers, DLKM, 379–383
  example of loading and initialization coding for, 381
example of _unload function for, 383
WSIO monolithic drivers, DLKM, 376, 383