Legal Notices

This document contains information which is protected by copyright. All rights are reserved. Reproduction, adaptation, 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 incorporate all updates since the previous edition. The manual part number and printing date identify the current edition.

Edition / Part Number / Date
First / B2355-90669 / September 1999
Second / B2355-90727 / August 2000
Third / B2355-90728 / November 2000
  • Minor technical and format changes throughout.
Fourth / B2355-90760 / July 2002
  • New Interrupt Migration chapter and minor technical and format changes throughout.
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 .................................. 45
   Kernel Data Structures Used for I/O ............................ 47
     The buf Structure ............................................ 47
     The isc_table_type Structure ............................... 47
     The uio Structure ............................................ 49
   Timeouts Mechanisms .......................................... 50
     How Timeouts Work ......................................... 50
     The timeout() Routine ...................................... 51
     The untimeout() Routine .................................... 51
   Interrupt Handling ............................................ 52
     The Software Trigger Mechanism .............................. 53
   Memory Allocation and Mapping I/O Space ....................... 56
Contents

Memory Allocation Services ........................................ 56
Memory Mapped Registers Services ................................ 57
Configuration Space Accesses ..................................... 58
I/O Port Access Services ........................................... 58
Cache Coherence .................................................... 60
Driver Requirements for Coherency ............................... 61
Rules For Using dma_sync_IO .................................... 62
DMA Mapping ....................................................... 63
I/O Adapters ......................................................... 64
WSIO Mapping Services ........................................... 65
Programming Considerations ..................................... 67

4. Multiprocessing
   Comparing Uniprocessing to Multiprocessing .................. 71
   Synchronization Mechanisms ................................... 72
      Timing Hazards and Idle Time ............................... 72
   Spinlocks ...................................................... 73
      Spinlock Routines ......................................... 73
   Beta Semaphores ............................................... 74
      Beta Semaphore Routines .................................. 74
   Deadlocks ...................................................... 75
      Rules for Lock Acquisition ............................... 75
   Lock Orders ................................................... 75
   Synchronization Using sleep() and wakeup() ............... 76

5. Writing a Driver
   Suggested Driver Writing Methodology ....................... 81
   Writing a Driver: Step-by-Step ................................ 81
      Step 1: Choosing a Driver Name ........................... 82
      Step 2: Choosing System Header Files .................... 83
         System-Defined Header Files .......................... 83
      Step 3: Defining Installation Structures ................. 85
         The drv_ops_t Structure Type ........................ 85
         The drv_info_t Structure Type ....................... 88
         The wsio_drv_data_t Structure Type .................. 90
         The wsio_drv_info_t Structure Type .................. 91
Contents

Support for PCI OLAR .......................................................... 176
Registering a Driver Handler ............................................ 176
Registering a Driver Event Mask ........................................ 179
Setting Event Timeout Values ........................................... 179
Driver Event Handler ......................................................... 181

6. Installing Your Driver
   Step 1: Compile Your Driver ............................................ 187
   Compiling a Driver ......................................................... 187
   Step 2: Create a Master File .......................................... 190
   Creating a Master File for Your Driver ........................... 190
   Editing the Master File .................................................. 191
   Example Master File .................................................... 192
   Step 3: Modify the System File ....................................... 196
   Step 4: Build a Kernel Containing Your Driver .................. 197
   Step 5: Create a Device-Special File ............................... 202

7. Creating Networking Device Drivers
   HP-UX Networking Interface Architecture ......................... 207
   Data Link Interface Layer ............................................. 209
   Network Protocol Layer ............................................... 209
   Protocol Interface Layer, Device File, and STREAMS Head .... 209
   STREAMS Environment .................................................. 210
   Overview of Networking Driver Structure ......................... 211
   STREAMS Model Device Drivers ...................................... 215
   Data Structures and Interfaces ....................................... 216
   hw_ift_t Structure Description and Initialization ............... 216
   hw_dlpi Structure Description and Initialization ............... 221
   Protection and Synchronization for Driver and Transport ....... 222
   hw_ift Structure Protection ............................................ 222
   Driver Structure Protection .......................................... 223
   Initializing Networking Device Drivers ............................ 224
   Calling driver_install() ............................................... 226
   Calling driver_attach() ................................................. 228
   Protocol Binding and Demultiplexing .............................. 233
   Protocol Binding and Unbinding .................................... 234
Contents

Protocol Demultiplexing ........................................... 234
Distinguishing Packet Protocol Format ............................... 235
Inbound Promiscuous and Outbound Promiscuous ................... 236
Message Block and Queue Functions and Macros .................... 238
DLPI Interface to Upper Layers .................................... 239
  Device file, Interface name and PPA number ...................... 239
  IP and ARP Configuration ...................................... 240
STREAMS DLPI and Network Driver Overview .......................... 242
  Device Driver/DLPI Driver Synchronization ....................... 244
  STREAMS Synchronization ....................................... 245
  Entering STREAMS from ICS ..................................... 246
  Driver Support for DLPI ......................................... 246
  Driving the NIC .................................................. 257
Platform Specifics .................................................. 273
  DMA Mapping ..................................................... 273
  V Class ......................................................... 273
Network Management Support .................................... 276
Network Tracing and Logging Support for Troubleshooting ............ 280
  Introductory Overview of HP-UX Tracing and Logging .......... 280
  Using HP-UX Logging and Tracing for Troubleshooting Support 284
  Passing Data to HP-UX Tracing/Logging ........................ 290
  Formatting Networking Trace/Log Messages ....................... 299
Alternative Means of Development Debugging .......................... 302
Configuring a Networking Driver Through SAM ........................ 303
  SAM Interface to IHV Network Drivers ......................... 304
  Generic Network Configuration Supported by SAM ............... 306
  Driver Configuration File and Init Script ....................... 306
  SAM/Driver get/set Request Flow ................................ 309
  Defines Required at the Driver .................................. 313
Network Monitoring Commands ................................... 314
  driveradmin command ........................................ 314
  lanscan Command .............................................. 318
  driverlinkloop Command ....................................... 320

8. Writing a SCSI Interface Driver

  Overview of HP-UX SCSI Subsystem ............................ 323
  External Interfaces to an Interface Driver ..................... 325
  WSIO CDIO DDE ................................................. 325
Contents

SCSI Services Interface .................................................. 325
I/O Path ............................................................................. 353
I/O Within SCSI Device Driver ......................................... 354
I/O Within SCSI Service Layer .......................................... 355
I/O Within SCSI Interface Layer ...................................... 356
Interface Driver Development ......................................... 357
  Driver Installation and Initialization .............................. 357
  Register Mapping and DMA ............................................ 366
  Protection and Synchronization ................................... 368
  Sending SCSI I/O Requests to the Target ...................... 370
  Processing Completed I/O Requests ............................. 371
  Interrupt Processing ....................................................... 374
  Special Routines .......................................................... 374
Sample Driver .................................................................. 378
  Description of the Device .............................................. 378
  Driver Architecture ....................................................... 379
  Data Structures ............................................................ 379
  qlisp_multi driver ........................................................ 380
  qlisp driver .................................................................. 385

9. Writing SCSI Device Drivers
   SCSI Driver Development, Step 1: Include Header Files ..... 397
       General Header Files .................................................. 397
       Header Files for SCSI Drivers .................................... 397
       Header Files for Device Classes ................................ 397
   SCSI Driver Development, Step 2: Set Up Structures ...... 399
   SCSI Driver Development, Step 3: Create the driver_install Routine .............................................. 402
   SCSI Driver Development, Step 4: Create the driver_dev_init() Routine ............................................. 403
       Setting up the Device-Switch Table (scsi_ddsw) ........ 403
   SCSI Driver Development, Step 5: Analyze Multiprocessor Implications ........................................... 407
   SCSI Driver Development, Step 6: Create the Entry-Point Routines .................................................... 408
       driver_open() Routine .................................................. 408
       driver_close() Routine ................................................ 408
## Contents

Multibus Drivers .......................................................... 461  
PCI Services Summary .................................................... 463  
Unsupported WSIO Functions ........................................... 464  
Multiprocessor (MP) Safety ............................................. 465  
Constants and Data Structures ....................................... 466  
  User Visible PCI-Specific Data Structures ..................... 466  
  Defined Constants ................................................... 466  
A Sample PCI Driver .................................................... 468  
Sample WSIO Setup and Structures .................................. 468  
Sample WSIO Routines ................................................. 471  
Other Driver Entry Point Routines .................................. 475

11. On-Line Addition/Replacement  
  Introduction ...................................................................... 479  
  Supported Hardware and Software .................................. 479  
  Overview of Driver Modifications Required for OLA/R ...... 480  
  WSIO Interfaces .......................................................... 481  
  wsio_drv_info_t .......................................................... 481  
  WSIO Event Handling Structures ................................... 481  
  Driver Modifications for OLA/R Support ....................... 484  
  Driver Registering Its Event Handler ............................. 484  
  Driver Registering Its Event Mask .................................. 485  
  Driver Event Timeout Values ....................................... 487  
  Event Handling Function ............................................. 488  
  Handling OLA/R .......................................................... 490  
  OLA .............................................................................. 490  
  OLR ............................................................................. 492  
  Miscellaneous Changes Required at the Driver ............... 499  
  Performing OLA/R of PCI I/O Cards ............................. 500

12. Dynamically Loadable Kernel Modules  
  Managing Dynamically Loadable Kernel Modules .............. 503  
  DLKM Concepts .......................................................... 504  
  DLKM Tools ............................................................... 515  
  DLKM Procedures for Dynamically Configured Loadable Modules 520  
  DLKM Procedures for Statically Configured Loadable Modules 530
## Contents

### 13. How To Make Pre 11.0 Drivers 64-Bit Safe
- Introduction .................................................. 535
- The 64-bit Data Model ........................................ 536
- LP64 Considerations ........................................... 537
- General Guidelines ............................................ 539
- 64-bit Clean Headers .......................................... 539
- Useful Data Types ............................................. 540
- Hardware Considerations .................................... 542
- Software Considerations ..................................... 543
- ANSI Function Prototypes .................................... 544

### 14. Interrupt Migration
- Introduction .................................................. 551
- Overview of Interrupt Migration Impact on Driver ........ 552
- LBI Drivers .................................................... 552
- TBI Drivers .................................................... 552
- Interrupt Line Sharing ....................................... 552
- Interrupt Migration Event Masks and Registration with WSIO ........................................... 553
- LBI Event Flags ................................................ 553
- TBI EVENT FLAGS ............................................ 557
- Interrupt Migration Flow ...................................... 563
- Assumptions and Dependencies ............................... 565

### A. Data Structures, Defines, Routines, Flags, and Code Examples
- Interface Group MIB Objects ............................... 569
- Standard MIB II .............................................. 569
- Extended MIB II for 802.3 and 802.5 ....................... 570
- MAC Types and Protocol Types ............................. 572
- Message Types for DLPI Primitive and Acknowledgment 573
- MIB Event and Event Record ............................... 574
- MTU Values .................................................... 575
- Packet Headers ............................................... 576
- Packet Types .................................................. 579
- Protocol Kinds and Values ................................... 580
- SAP Values for IEEE 802.2 LLC Packets .................. 581
- TYPE Values for Ethernet and SNAP Protocols .......... 582
B. How to Design a Networking Trace/Log Subformatter and a Sample Subformatter
   - Designing a Subformatter ............................................................... 585
   - Alternative Subformatter Implementations .................................. 586
   - Network Trace/Log Subsystem Installation Testing ...................... 621

C. Shared Library Examples for the driveradmin and lanscan Commands
   - The lanscan's shared library sample code ..................................... 625
   - The driveradmin's shared library sample code ............................. 627
   - The driveradmin sample code ..................................................... 641
   - The enetlinkloop's sample code .................................................. 703

Glossary
Chapter 1

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 SCSI Interface Drivers**

SCSI bus interface driver routines.

**Chapter 9: Writing SCSI Device Drivers**

SCSI bus device driver routines.
Chapter 10: Writing PCI Device Drivers

PCI bus driver routines.

Chapter 11: On-Line Addition and Replacement

Driver requirements when adding or removing a PCI card with power on.

Chapter 12: Developing Dynamically Loadable Kernel Modules

Adding a kernel module to a running UNIX system without rebooting the system or rebuilding the kernel.

Chapter 13: How To Make Pre 11.0 Drivers 64-Bit Safe

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

Chapter 14: Interrupt Migration

How to use this mechanism for managing interrupt assignments.

Appendix A: Data Structures, Defines, Routines, Flags, and Code Examples

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

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

Generic subformatter code for networking drivers.

Appendix C: 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. Chapter 10, “Writing PCI Device Drivers” contains the bus specific routines for PCI. Chapter 11, *On-Line Addition and Replacement*, describes how a driver coordinates the on-line addition or replacement of a PCI card.

Chapter 12, “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:

- **Hardware Provider Program** at [http://www.hp.com/dspp/hphp](http://www.hp.com/dspp/hphp)
- **Hardware Provider Program E-mail** at interface@fc.hp.com
- **Developer Resource Driver Page** at [http://h21007.www2.hp.com/dspp/topic/topic_PETaskDetailPage_IDX/1,,10303,00.html](http://h21007.www2.hp.com/dspp/topic/topic_PETaskDetailPage_IDX/1,,10303,00.html)
Reference Documentation

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

— 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
Overview of the Driver Environment
This book is intended for individuals writing interface and device drivers for HP-UX Workstations and servers.

This chapter describes how the HP-UX I/O subsystem is structured, and how 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.
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 CDIOs to be configured in or out of a system as needed.

There are two categories of CDIOs:

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

- **Driver Services**

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

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

- **Drivers**

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

- **Management of I/O resources**

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

**How the Driver Environment Works**

The WSIO CDIO was originally designed for the workstation single-processor environment. With HP-UX Release 10.20, its functionality was expanded to encompass the server multiprocessor environment 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.
Overview of the Driver Environment
How the I/O Subsystem is Structured

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 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

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
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 I/O Space” section discusses general kernel memory allocation and gives pointers to accessing card registers.
- The “Cache Coherence” section discusses maintenance of cache coherence.
- The “DMA Mapping” section describes I/O virtual address mapping and WSIO mapping services.
Overview of Driver Types

WSIO supports 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 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()` 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 85, 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.
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:

<table>
<thead>
<tr>
<th>Bits 0-7</th>
<th>The major number, which can range from 0 to 255. Character and block major numbers are separate ranges.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits 8-31</td>
<td>The minor number. The conventional notation for the minor number follows the format 0xhhhhhh, where h is a four-bit hexadecimal digit. As a general guideline, bits 8-15 encode the instance number of the interface card. The number represents the order in which HP-UX encounters the interface card within a class when binding it into the system. This number is displayed (in decimal notation) in the I column of <code>ioscan -i</code> output. Bits 16-31 encode device and driver dependent characteristics. These can include special</td>
</tr>
</tbody>
</table>
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 102, Chapter 5, “Step 3: Defining Installation Structures” on page 85, Chapter 5, and “Step 2: Create a Master File” on page 190, Chapter 6.
System Calls

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

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

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

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

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

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 device and the needs of the driver.

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

4. In the interface driver

Process request.

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

5. In the device driver

Complete request.

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

6. In the kernel

Return control to user process.

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

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

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

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 `wsio_get_isc(WSIO3)`) and then calls the interface driver through the switch table to perform the operation.
This allows the device driver to trigger interface-specific routines without actually having to know which interface driver is configured with it. Multiple interface drivers could be configured interchangeably with this device driver, as long as they share the same ifsw type definition.

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

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

```
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.

See `timeout(KER2)` for details.

The untimeout() Routine

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

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

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

In 11i there is a set of new interrupt services that allow drivers to allocate multiple interrupt resources. Drivers can specify whether an interrupt is to be line based or transaction based. The new WSIO interrupt services are:

Table 3-1  New WSIO Interrupt Services

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

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

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

There are two new event types that can be passed to a driver’s event handler. Drivers that use transaction based interrupts must support them. They are `CPU_ONLINE` and `CPU_OFFLINE`. The first indicates that a new CPU is available and can be used by the driver for transaction based interrupts. The second indicates that a CPU is going away and the driver must migrate its interrupts to another CPU. For more on the driver event handler see the `man` pages for the `wsio_install_drv_event_handler` and `wsio_reg_drv_capabilites`. Also, see the sections on these services in Chapter 5 of this manual.

**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`. 
Understanding HP-UX I/O Subsystem Features

Interrupt Handling

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

static struct sw_intloc mycard_intloc;

static void
mycard_isr(void)
{
    caddr_t reason;

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

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

    /* set up swtrigger() request to perform */
    /* remainder of interrupt processing at */
    /* a lower priority level. */

    sw_trigger(&mycard_intloc, mycard_isr2, reason, 3, 0);

    return;
}

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

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

Memory Allocation Services

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

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

Drivers can allocate multiple memory handles for the different types of memory buffers they use. For example, a driver could specify a memory handle that only allocates physical memory below four Gbytes. It could then specify another handle for allocating memory that is always physically contiguous. The following table lists these new services.

Table 3-2 Memory Allocation Services

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wsio_alloc_mem_handle()</code></td>
<td>This service is used to specify the type of memory a driver will allocate. It returns a memory handle.</td>
</tr>
<tr>
<td><code>wsio_free_mem_handle()</code></td>
<td>This service frees a memory handle allocated by <code>wsio_alloc_mem_handle()</code>.</td>
</tr>
</tbody>
</table>
Table 3-2 Memory Allocation Services (Continued)

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wsio_alloc_mem()</td>
<td>This service is called to allocate memory.</td>
</tr>
<tr>
<td>wsio_free_mem()</td>
<td>This service is called to free memory.</td>
</tr>
</tbody>
</table>

For more specific information on how to call these services and the parameters passed to them, see their man pages in the DDR.

Memory Mapped Registers Services

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

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

Table 3-3 Memory Mapped Registers Services

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

For more specific information on how to call these services and the parameters passed to them, see their man pages in the DDR.
Configuration Space Accesses

This set of services is used by device drivers to discover and access configuration space resources. The first service, `wsio_map_cfg_handle()` is used to obtain a configuration space handle. The handle is then passed to the services `wsio_cfg_inXX()` and `wsio_cfg_outXX()`. There are separate versions of these services for 8, 16, 32, and 64 bit accesses (XX == 8, 16, 32, 64). The following table lists the new set of services for configuration space accesses.

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

For more specific information on how to call these services and the parameters passed to them, see their man pages in the DDR.

I/O Port Access Services

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

The following table lists the I/O port access services.
Table 3-5  I/O Port Access Services

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

For more specific information on how to call these services and the parameters passed to them, see their man pages in the DDR.
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.

Semicohherent 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 10, Writing PCI Device Drivers. They are also discussed in pci_errata(PCI5), wsio_map_dma_buffer(WSIO3), and wsio_fastmapd_dma_buffer(WSIO3) in the HP-UX 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 flushed 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 driver must call dma_sync_IO() with the IO_READ hint set. On noncoherent platforms, this will cause the associated processor caches to be purged of data that may have been prefetched. For all but the last buffer, the IO_NO_SYNC hint should also be set to reduce the performance penalty of synchronizing the cache purges on noncoherent platforms. On semicoherent platforms, the processor caches will be made to synchronize with the data read when the IO_NO_SYNC hint is not set.
DMA Mapping

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

- **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 do not have special address translation hardware imbedded in I/O adapters, implement only the physical address view to I/O devices. Platforms that do have address translation hardware for I/O generally implement the IOVA view.

  IOVAs enable devices on a 32-bit wide I/O bus to access host memory physical addresses 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.

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

An I/O adapter provides I/O virtual address (IOVA) translation between an I/O bus and the processor/memory interconnect. Devices on the I/O bus issue bus transactions to IOVA memory targets and the I/O adapter translates IOVA memory targets to physical addresses. The I/O adapter also participates in the coherency protocol of the processor caches for platforms that are coherent or semicoherent.
Address translation is assisted by the I/O PDIR (Page DIrectory) associated with an I/O adapter. The IO PDIR is analogous to the PDIR used by processors for virtual-to-physical address translations. It is a table maintained by the kernel to provide mappings between IOVAs and physical addresses.

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

**Coherent I/O Systems**

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

**Noncoherent I/O Systems**

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

**WSIO Mapping Services**

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

<table>
<thead>
<tr>
<th>Table 3-6</th>
<th>WSIO DMA Mapping Services</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Interface Name</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>wsio_allocate_dma_handle()</td>
<td>Obtain a DMA handle used to set up DMA</td>
</tr>
<tr>
<td>wsio_free_dma_handler()</td>
<td>Free a DMA handle</td>
</tr>
</tbody>
</table>
### Understanding HP-UX I/O Subsystem Features

#### DMA Mapping

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

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

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

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

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

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

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

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

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

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

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

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

4 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:
Multiprocessing

Synchronization Using sleep() and wakeup()

```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()`.

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

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.
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 10, “Writing PCI Device Drivers.” If you are writing a loadable driver module, also read Chapter 12, “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 196, 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, `mydriver` and `skel` are used as sample driver names.
Step 2: Choosing System Header Files

A driver's header contains definitions of data structures, macros, and constants that are used in the compilation of the driver module. 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.
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;
```
Writing a Driver

Step 3: Defining Installation Structures

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>
<thead>
<tr>
<th>Field</th>
<th>Device Type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>d_open()</td>
<td>both</td>
<td>Pointer to your driver_open() routine, which enables a device for subsequent operations. If the device is off line, does not exist, or cannot be accessed, an error should be returned. When appropriate, it is permissible for the open routine to do nothing.</td>
</tr>
<tr>
<td>d_close()</td>
<td>both</td>
<td>Pointer to your driver_close() routine, which disables interrupts, resets a device, frees resources, and performs other tasks required when a device is closed.</td>
</tr>
<tr>
<td>d_strategy()</td>
<td>block</td>
<td>Pointer to your driver_strategy() routine, which queues I/O requests for either reading or writing. Drivers of character devices often call physio() from their read and write routines; physio() calls the strategy routine passed in as a parameter, but it is not an entry point into a character driver.</td>
</tr>
<tr>
<td>d_psize()</td>
<td>block</td>
<td>Pointer to your driver_psize() routine. For a swapping device, it should return the size of the swap partition. Consider writing this routine only if your device is used for swapping.</td>
</tr>
<tr>
<td>d_read()</td>
<td>character</td>
<td>Pointer to your driver_read() routine, which should return the requested data transferred from the device.</td>
</tr>
</tbody>
</table>
### Step 3: Defining Installation Structures

Writing a Driver

Table 5-1 Device Driver Fields, `drv_ops_t` Structure (Continued)

<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 <code>true</code> 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>
Writing a Driver

Step 3: Defining Installation Structures

Table 5-1 Device Driver Fields, drv_ops_t Structure (Continued)

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

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.
Writing a Driver

Step 3: Defining Installation Structures

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></td>
<td>DRV_CHAR Character device driver.</td>
</tr>
<tr>
<td></td>
<td>DRV_BLOCK Block device driver.</td>
</tr>
<tr>
<td></td>
<td>DRV_PSEUDO Pseudo driver.</td>
</tr>
<tr>
<td></td>
<td>DRV_SCAN Driver supports bus scanning.</td>
</tr>
<tr>
<td></td>
<td>DRV_MP_SAFE Driver provides its own multiprocessing protection.</td>
</tr>
<tr>
<td></td>
<td>DRV_SAVE_CONF Save configuration information to /etc/ioconfig. This file retains potentially volatile information, such as dynamic major numbers and card instance numbers, across reboots.</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>
</tbody>
</table>
Writing a Driver

Step 3: Defining Installation Structures

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

NOTE

The values 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 190, 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:

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

The fields are described below.

<table>
<thead>
<tr>
<th>Field</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>drv_path</td>
<td>Follow these guidelines:</td>
</tr>
<tr>
<td></td>
<td>- 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.</td>
</tr>
<tr>
<td></td>
<td>- For interface drivers, drv_path should match the card's type. For example, scsi instead of ext_bus.</td>
</tr>
<tr>
<td></td>
<td>- For pseudo drivers, drv_path should match the card's class. For example, graphics.</td>
</tr>
</tbody>
</table>
Writing a Driver

Step 3: Defining Installation Structures

drv_type

One of the following values:

- T_DEVICE - The driver controls a hardware device.
- 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.

drv_get_minor,

io_path_mgr

Drivers should set these fields to NULL.

The wsio_drv_info_t Structure Type

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

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

Sample Header for a Device Driver

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

Step 3: Defining Installation Structures

```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,
    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",
```
Writing a Driver

Step 3: Defining Installation Structures

Chapter 5

T_DEVICE,
DRV_CONVERGED,
NULL,
NULL,
NULL,
NULL,

};

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

Sample Header for an Interface Driver

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

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

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

/*
 * No entry points for an interface driver,
 * so set all the fields to NULL.
 */

static drv_ops_t skel_ops =
{
NULL, NULL, NULL, NULL, NULL,
NULL, NULL, NULL, NULL, NULL,
NULL, NULL, NULL, NULL, NULL,
0,
};

static drv_info_t skel_info =
{

"skel",
"ext_bus",
DRV_SAVE_CONF | DRV_MP_SAFE,
NULL,
NULL,
NULL,
NULL,
NULL,
NULL,

};

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

};

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

};

**Sample Header for a Pseudo Driver**

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

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

Step 3: Defining Installation Structures

#include <sys/conf.h>

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

static drv_ops_t my_ops =
{
    my_open, my_close, NULL, NULL, NULL, my_read, my_write, NULL, NULL, NULL, NULL, NULL, NULL, NULL, 0
};

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”).
Writing a Driver

Step 3: Defining Installation Structures

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

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

static wsio_drv_info_t my_wsio_info =
{
    &my_info,
    &my_ops,
    &my_ops,
    &my_data,
    WSIO_DRV_CURRENT_VERSION
};
```
### 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></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() b</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_psize()</td>
<td></td>
</tr>
<tr>
<td>Block Pseudo</td>
<td>driver_install()</td>
<td>driver_open()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>driver_dev_init()</td>
<td>driver_close()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_strategy()</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_psize()</td>
<td></td>
</tr>
<tr>
<td>Character Device</td>
<td>driver_install()</td>
<td>driver_open()</td>
<td></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></td>
<td></td>
<td>driver_strategy()</td>
<td></td>
</tr>
<tr>
<td>Interface</td>
<td>driver_install()</td>
<td>None c</td>
<td></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></td>
<td></td>
<td>driver_isr()</td>
<td></td>
</tr>
<tr>
<td>Block Monolithic d</td>
<td>driver_install()</td>
<td>driver_open()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>driver_attach()</td>
<td>driver_close()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>driver_dev_init()</td>
<td>driver_strategy()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>driver_if_init()</td>
<td>driver_psize()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>driver_probe()</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver_isr()</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 97. 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.
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.
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.

- \texttt{driver_isr()} is the interrupt service routine for a device or interface, established by the interface driver.

### Table 5-3 For a Block Driver

<table>
<thead>
<tr>
<th>System Call</th>
<th>Executes</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>open()</td>
<td>driver_open()</td>
<td>Open the device</td>
</tr>
<tr>
<td>close()</td>
<td>driver_close()</td>
<td>Close the device</td>
</tr>
<tr>
<td>read()</td>
<td>driver_strategy()</td>
<td>Perform block read</td>
</tr>
<tr>
<td>write()</td>
<td>driver_strategy()</td>
<td>Perform block write</td>
</tr>
<tr>
<td>kernel</td>
<td>driver_psize()</td>
<td>Specify swap partition size</td>
</tr>
</tbody>
</table>

### Table 5-4 For a Character Driver

<table>
<thead>
<tr>
<th>System Call</th>
<th>Executes</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>open()</td>
<td>driver_open()</td>
<td>Open the device</td>
</tr>
<tr>
<td>close()</td>
<td>driver_close()</td>
<td>Close the device</td>
</tr>
<tr>
<td>read()</td>
<td>driver_read()</td>
<td>Perform character read</td>
</tr>
<tr>
<td>write()</td>
<td>driver_write()</td>
<td>Perform character write</td>
</tr>
<tr>
<td>ioctl()</td>
<td>driver_ioctl()</td>
<td>Perform special command</td>
</tr>
<tr>
<td>select()</td>
<td>driver_select()</td>
<td>Test I/O completion</td>
</tr>
</tbody>
</table>
Step 4: Identifying Routines for Your Driver

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

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

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

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

Configuration Overview

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

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

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

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

The install routines usually perform the following operations:

a. All install routines call wsio_install_driver() or install_driver(), which registers the driver with the system.
b. If a `devd_dev_init()` routine exists, `devd_install()` puts the routine on the applicable chain of init routines for that type of device.

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

d. `ifd_install()` puts `ifd_attach()` on the applicable chain(s) of attach routines for supported interfaces (e.g., the `eisa_attach()` and 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_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 WSIO, typically with certain pseudo drivers (see `install_driver(CDIO3)`).

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

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

- For an interface or monolithic driver, if a `driver_probe()` routine is defined, `driver_install()` calls either `wsio_register_dev_probe()` or `wsio_register_addr_probe()` to register the routine with the system. See `wsio_register_dev_probe(WSIO3)` and `wsio_register_addr_probe(WSIO3).`
For a device, pseudo, or monolithic driver, if a `driver_dev_init()` routine is defined, `driver_install()` saves a pointer to the current head of the chain and places its `driver_dev_init()` routine at the head of the chain. The chain is processed once all drivers have been configured. The head of the global init chain is pointed to by `dev_init()`.

**driver_install() Return Values**

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

Those values are:

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

If it fails, the appropriate message 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 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 */
```
Writing a Driver

Step 5: Writing Configuration Routines

wsio_register_dev_probe(drv_name, skel_probe, &skel_drv_info.name);

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


driver_install() Example for Monolithic Drivers

Installation routine for a driver named skel.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

int
sdisk_install(void)
{
    sdisk_link();

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

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

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

Writing a driver_if_init() Routine

The \texttt{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 \texttt{gfsw} structure during \texttt{driver_attach()}. 
static int
mydriver_attach(uint32_t card_id, struct isc_table_type *isc)
{
    if (card_id == my_card_id) {
        /* if mydriver has an init routine */
        isc->gfsw->init = mydriver_init;
        ...
    }
}

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

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

    status = wsio_intr_set_irq_line(isc, my_intr_obj,
                                    WSIO_IRQ_LINE_AUTO, 0);
    if(status != WSIO_OK)
        return(ERROR);

    status = wsio_intr_activate(isc, my_intr_obj);
    if(status != WSIO_OK)
        return(ERROR);

    /* successfully initialized the interface */
    return(0);
}
Writing Driver Probe Routines

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

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

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

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

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

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

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

**wsio_register_dev_probe()**

The calling semantics for `wsio_register_dev_probe()` are:

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

Where `type` can be either `IF_CLASS` or `DRV_NAME`, `func` is a pointer to the driver probe function and `str` is an ascii string.
The service `wsio_register_dev_probe()` is used to register a probe function for a driver based on either the driver's 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);
```
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
     */
```
Writing a Driver

Step 5: Writing Configuration Routines

```c
** parallel_scsi_probe function for the address probe.**
if (ret = wsio_install_driver(&c720_wsio_info)) {
    c720_saved_pci_attach = pci_attach;
    pci_attach = (int (*)())c720_pci_attach;
    wsio_register_addr_probe(parallel_scsi_probe,
                            "scsi_c720");
}
return ret;
```

Driver address probe functions registered by `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>
<thead>
<tr>
<th>Table 5-6 Address Probe Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>node</td>
</tr>
<tr>
<td>dev_probe</td>
</tr>
<tr>
<td>drv_info</td>
</tr>
<tr>
<td>probed</td>
</tr>
<tr>
<td>hw_path</td>
</tr>
<tr>
<td>hw_path</td>
</tr>
<tr>
<td>isc</td>
</tr>
<tr>
<td>probe_type</td>
</tr>
</tbody>
</table>
Chapter 5

Step 5: Writing Configuration Routines

Table 5-6  Address Probe Parameters (Continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>OUT</td>
<td>A pointer to a string initialized with the device’s name such as “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>

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

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

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

When a driver class probe function is used with an address probe function and hence is not called directly by the WSIO CDIO probe code, it does not have to adhere to the WSIO calling semantics that were described earlier for class probe functions. The driver stack can define its own parameter list for the class probe function. In this case though it is important that the class probe function is never used as a standalone probe function that is called directly by the WSIO CDIO probe code.
In the following two examples the `scsi_c720.c` interface driver registers an address probe function called `parallel_scsi_probe()` and the `scsi_ctl.c` device driver registers a class probe function called `scsi_probe()`. In these examples the two probe functions cooperate and hence the interface to the class probe function `scsi_probe()` does not adhere to the WSIO CDIO specification.

**Example of a driver_addr_probe() Routine**

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

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

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

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

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

#define NEXT_TARGET 1
#define NEXT_LUN 2

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

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

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

found = probe_func(isc, NULL, &probe_dev, probe_type,
                    probe_id, dev_class, desc);

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

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

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

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

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

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

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

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

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

    /*
     * Driver has used a field in the ISC telling the maximum
     * width of the bus it can support. The maximum number of
     * targets is one less (since the IDs start at 0).
     */
```

---

Chapter 5 121
A sanity check is done to ensure we're dealing with positive number.

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

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

/*
 * Get the major number of the driver
 * and build a dev_t structure.
 */
major_num = sctl_wsio_info.drv_info->c_major;
dev = ((major_num << 24) & 0xff000000);
dev |= ((probe_dev->instance << 16) & 0x00ff0000);
dev |= ((probe_dev->target << 12) & 0x0000f000);
dev |= ((probe_dev->opt_1 << 8) & 0x00000f00);
found = NO_DEV;
if ((sctl_open(dev)) == 0) {
    if ((sctl_ioctl(dev, SIOC_INQUIRY, &iqr_data, 0)) == 0) {
        dev_type = iqr_data.inq2.dev_type;
        if (dev_type != NO_SCSI_DEV) {
            found = VALID_TGT;

            /*
             * Get first four bytes of the inquiry data
             * for dev_id and the vendor and product id for
             * desc. and set dev class according to the
             * dev_type.
             */

            *(int *)dev_id = *((int *)&iqr_data);
            strncpy(desc, iqr_data.inq2.vendor_id, 8);
            strncpy(&desc[8], iqr_data.inq2.product_id, 16);
            desc[24] = '\0';
```

```
switch(dev_type) {
    case SCSI_PROCESSOR:
        strcpy(dev_class, "processor");
        break;
    case SCSI_DIRECT_ACCESS:
        case SCSI_CDROM:
        case SCSI_MO:
        case SCSI_WORM:
            strcpy(dev_class, "disk");
            break;
    case SCSI_SEQUENTIAL_ACCESS:
        strcpy(dev_class, "tape");
        break;
    case SCSI_AUTOCHANGER:
        strcpy(dev_class, "changer");
        break;
    case SCSI_PRINTER:
        strcpy(dev_class, "printer");
        break;
    case SCSI_SCANNER:
        strcpy(dev_class, "scanner");
        break;
    case SCSI_COMMUNICATIONS:
        strcpy(dev_class, "comm");
        break;
} /* switch */

/*
 * We know we found a valid device now lets
 * make sure that it is also a valid 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;
} /* scsi_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:

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

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

    /*
    * dev_id & func are the last two elements of the
    * hardware path of the device.
    */
    dev_id = path->addr[path->last_index -1];
    func = path->addr[path->last index];
    minor |= ((dev_id << 8) & 0xFF00;
    minor |= (func & 0xFF);
    return minor;
}
```
Step 6: Writing Entry Point Routines

Most drivers have defined entry point routines. An entry point routine is a driver routine that is called through a (non NULL) field of your \texttt{drv_ops_t} structure. Interface drivers do not have entry point routines by this definition. Their routines such as \texttt{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 \texttt{driver_open()} Routine

The \texttt{driver_open()} routine prepares a device for I/O. \texttt{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 \texttt{d_open} field of the \texttt{drv_ops} structure. Commonly, \texttt{driver} is replaced by your driver's name.

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

In general, the \texttt{driver_open()} routine does the following:

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

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.)

```c
int driver_open(dev_t dev, int flag);
```
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 `major(KER2)` and `minor(KER2)` in the *HP-UX Driver Development Reference.*) See NOTE below.

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

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

Writing a driver_close() Routine

driver_close() is provided by the driver writer. It can have any unique name. 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:

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

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

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

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

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

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

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

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

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

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

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

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

4. The device must remain open until all active inodes for the device are closed. If more than one active inode can exist for the same device, (if two device files with the same major number can access the same device), the driver should set the `C_ALLCLOSES` flag and maintain its own count of how many processes have the device open. Note 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.
Writing a Driver

Step 6: Writing Entry Point Routines

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 128, for the corresponding driver_open() routine.

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

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

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

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

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

Writing a driver_read() or driver_write() Routine

The driver_read() and driver_write() routines control device I/O. The names reflect the kernel view of an I/O transaction. A read transaction moves data from the device to processor memory, while a write transaction moves data from processor memory to a device. driver_read() and driver_write() are provided by the driver writer. They can have any unique name. 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 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 uio_move(). The following sections describe the driver_read() and driver_write() routines, as well as how to use physio() and uio_move(). See physio(KER2) and uio_move(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 uio_move() to buffer the data and then to complete the request. If you use uio_move(), 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 uio_move() to transfer the data from the kernel's buffer to the user's buffer.
  6. Returns a value to the read() or readv() call.

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

- Use physio() and a driver_strategy() routine.
Use `uiomove()`, in which case the `driver_write()` routine does the following:

1. Initializes data structures.
2. Calls `uiomove()` to copy the data into kernel space.
3. Sets a flag indicating that I/O is in progress.
4. Requests that the device start the I/O operation.
5. Waits or sleeps while the device completes the I/O operation.
6. Returns a value.

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

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

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

- `dev` The device number of the associated device file. The routine can extract the major and minor numbers from the device number. 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 83. 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:

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

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

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

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

- *dev* The device number.

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

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

- *uiop* Pointer to the uio structure.

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

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

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

**Sample driver_read Routine Using physio()**

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

**Using uiomove()**

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

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

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

`cp` (pointer to kernel space to hold data)
Writing a Driver

Step 6: Writing Entry Point Routines

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

- Set `flag` to `UIO_READ` to have `uiomove()` copy `n` bytes, starting at address `cp`, into user space starting at address `uio`.
- Set `flag` to `UIO_WRITE` to have `uiomove()` copy `n` bytes from `uio` into kernel space starting at `cp`.
- If 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.
Writing a Driver

Step 6: Writing Entry Point Routines

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.

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

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

You can use ioctl() to:

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

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. \texttt{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 \texttt{ioctl()} requests include rewinding a tape and changing a printer's column width. (Refer to \texttt{ioctl(5)} in the \textit{HP-UX Reference} for a list of these requests and the standard processing your driver should perform.) Examine the header files in 
\texttt{/usr/include/sys} for examples of \texttt{ioctl()} command definitions. The command \texttt{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 \texttt{ioctl()} request argument. Define the command words for your driver in a header file. User programs that issue \texttt{ioctl()} calls for your driver must include this file.

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

\begin{verbatim}
#define command _IO('t', n)
#define command _IOR('t', n, object)
#define command _IOW('t', n, object)
#define command _IOWR('t', n, object)
\end{verbatim}

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

\textbf{\_IOR} \hspace{1cm} Read data from the driver. That is, the driver writes into the kernel buffer pointed to by \texttt{arg}. Before returning to the user, the system copies the kernel buffer to the user specified buffer.

\textbf{\_IOW} \hspace{1cm} Write data to the driver. That is, the driver reads from the kernel buffer pointed to by \texttt{arg}. Before calling the driver, the system copies the user specified buffer to the kernel buffer.

\textbf{\_IOWR} \hspace{1cm} Both \_IOR and \_IOW.

\textbf{\_IO} \hspace{1cm} Indicates that the ioctl command does not pass in an argument.
The parameters are defined as follows:

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

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

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

### LP64 Considerations

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

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 */
#include __LP64__
#define SOME_IOCTL_32 _IOR('X', 1, int)
/* from 32-bit app */
#undef __LP64__
/* Driver ioctl code snippet */
switch (cmd) {
```
Writing a Driver

Step 6: Writing Entry Point Routines

```c
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__
#define COPY_IOCTL_32 _IOWR('X', 2, struct buf_copy_32)
struct buf_copy_32 {
  ptr32_t buf_ptr32;
  int32_t buf_size;
};
#endif /* __LP64__ */

/* Driver ioctl code snippet */
switch (cmd) {
  case COPY_IOCTL:
    <do COPY_IOCTL processing>
    break;
  /* On 64 bit kernels these ioctls will process ioctls made by 32 bit applications */
 #ifdef __LP64__
  case COPY_IOCTL_32:
    <do COPY_IOCTL_32 processing>
    break;
 #endif /* __LP64__ */
}

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

```c
copyin((caddr_t)(struct _buf_copy_32 *)data->buf_ptr32), mybuf, sizeof(mybuf));
```
The `copyin()` and `copyout()` functions in the 64-bit kernel check to determine whether the calling thread is a 32-bit application and, if so, do the necessary 32-to-64-bit address conversion.

If 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.

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

/* Private Header File */
#ifdef __LP64__
#define _LONG_IOCTL_32(‘a’,1)
/* long data from 32 bit app */
#endif /* __LP64__ */

/* Driver ioctl code snippet */
switch (cmd) {
    case LONG_IOCTL:
        <do LONG_IOCTL processing>
        break;
    #ifdef __LP64__
    case _LONG_IOCTL_32(‘a’,1):
    /* long data from 32 bit app */
        <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;
};
```
Writing a Driver

Step 6: Writing Entry Point Routines

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

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

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

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

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

#define SET_TIMEOUT 0x04
#define MEMDEFAULT 0

struct mydevice_ioctl_arg ioctl_arg;

ioctl_arg.reg_value = SET_TIMEOUT;
ioctl_arg.location = SET;
if (ioctl(fd, CARD_CONTROL, &ioctl_arg) < 0)
    printf("ioctl call failed, error number = %d\n", errno);
```
Private Header file to handle 32-bit applications running on a 64-bit OS.

```c
#ifdef __LP64__

struct mydevice_ioctl_arg32 {
    char reg_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 ioctls. Note that the global variable
my_device registers point to the registers of a hypothetical piece of
hardware.

/* header files this code segment needs */

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

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

    switch(cmd)
    {
        case CARD_RESET:
            dev_rp->reset = 0;
            return(0);
        case CARD_STATUS:
            return(0);
    }
```
arg->reg_value = dev_rp->status;  
return(0);  
case CARD_CONTROL:  
    switch(arg->location)  
    {  
        case SET:  
            dev_rp->control |= arg->reg_value;  
            return(0);  
        case CLEAR:  
            dev_rp->control &= ~arg->reg_value;  
            default:  
                return(EINVAL);  
    } /* switch */  

case CARD_BUFADR:  
    arg->location = set_buf_addr(arg->location);  
    return(0);  

#ifdef __LP64__  
case CARD_STATUS_32:  
    arg32->reg_value = dev_rp->status;  
    return(0);  
case CARD_CONTROL_32:  
    switch(arg32->location)  
    {  
        case SET:  
            dev_rp->control |= arg32->reg_value;  
            return(0);  
        case CLEAR:  
            dev_rp->control &= ~arg32->reg_value;  
            return(0);  
        default:  
            return(EINVAL);  
    } /* switch */  
#endif  

    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.

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

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

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

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

Writing a driver_select() Routine

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

You do this by specifying the kernel function `seltrue()` in the `d_select` field of the `drv_ops` structure.
The `driver_select()` routine has no access to the `readfds`, `writefds`, and `exceptfds` values that the user passed to the `select()` system call. (See `select(2)` in the HP-UX Reference.) The `driver_select()` routine is passed only the device number and `flag`.

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

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

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

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

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

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

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

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

The `select()` system call handles collisions if a true value is passed in the `selwakeup()` call's second parameter. A true return (non-zero) indicates the `select` succeeded. When false is returned the `select(2)` system call sleeps and waits.

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

struct my_sel_struct
{
    struct kthread *read_waiter;
    struct kthread *write_waiter;
    int state;
}

extern int selwait;
extern lock_t *mylock; /* my driver's spinlock */
struct my_sel_struct *my_sel_struct;

static int
skel_select(dev_t dev, int flag)
{
    struct kthread *t;

    spinlock(my_lock);
    switch(flag) {
    case FREAD:
        if (available data) {
            spinunlock(my_lock);
            return 1;
        }
    if ((t=my_sel_struct->read_waiter) &&
        waiting_in_select(t))
```
Writing a Driver

Step 6: Writing Entry Point Routines

```c
mysel_struct->state |= READ_COLLISION;
else
    mysel_struct->read_waiter = u.u_kthreadp;
baby;
case FWRITE:
    if (ready for more data) {
        spinunlock(my_lock);
        return 1;
    }
    if ((t=my_sel_struct->write_waiter) &&
        waiting_in_select(t))
        mysel_struct->state |= WRITE_COLLISION;
    else
        mysel_struct->write_waiter = u.u_kthreadp;
baby;
} /* 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;
    }
}
```
Writing a Driver

Step 6: Writing Entry Point Routines

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
static void skel_input_ready(struct my_sel_struct * mystruct) {
    /*
     * if a process is sleeping on select for
     * this condition - wake it up
     */
    if (mystruct->read_waiter) {
        selwakeup(mystruct->read_waiter,
                  mystruct->state & READ_COLLISION);
        mystruct->read_waiter = NULL;
        mystruct->state &= ~READ_COLLISION;
    }
}
```

Writing a driver_strategy() Routine for a Block Device

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

For a block device, 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.
Writing a Driver

Step 6: Writing Entry Point Routines

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.

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().

void driver_strategy (struct buf * bp);

bp

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

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

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

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

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

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

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

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

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

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

Examples

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

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

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

static void
skel_strategy(struct buf * bp)
{
```
extern lock_t *my_lock;
register caddr_t addr;
register short cnt;
struct isc_table_type *isc;
isc = wsio_get_isc ( bp->b_dev,&isc,&skel_wsio_info )

spinlock( my_lock );
addr = bp->b_un.b_addr;
cnt = bp->b_bcount;
/* set up device */
/* isc->if_reg_ptr set up in attach */
skel = (struct skelregs *)isc->if_reg_ptr;
skel->registerX = ..........}

if (bp->b_flags & B_READ)
{
   /* This device doesn't read */
   bp->b_flags |= B_DONE;
   spinunlock( my_lock );
   return;
}
else
{
   /* Complete Write Transfer */
   if (~cnt)
   {
      bp->b_flags |= B_DONE;
      spinunlock( my_lock );
      return;
   }
   else
      skel_start(bp);
}
spinunlock( my_lock );

void
skel_start(struct buf *bp)
{
   addr = bp->b_un.b_addr;
cnt = bp->b_bcount;
/* special last byte setup, if needed */
if (cnt == 1) /* last byte, do it now */
   skel->control = AUX_SEOI;
Writing a Driver

Step 6: Writing Entry Point Routines

```c
skel->int_enable_reg = DOIE;
skel->data_out_reg = *addr++;
bp->b_bcount = cnt-1;
bp->b_un,b_addr++ ;
}
static void
skel_isr(void)
{
  skel->ch1.status_reg = D_CLEAR;
skel->int_enable_reg = ~DOIE;
r_int_enable_reg |= skel->int_enable_reg;
r_int_enable_reg &= ~DO;

cnt = skelbuf->b_bcount;
if (cnt == 0)
  biodone(skelbuf);
else
  skel_start(skelbuf);
}
```
Step 7: Writing Other Driver Routines

This section describes primary driver routines that are not defined entry point routines; that is, they are not called through entry points defined in fields of your `drv_ops_t` structure. Interface drivers typically have `driver_attach()` and `driver_isr()` routines. They also may have `driver_if_init()` and `driver_probe()` routines.

When an interface is shared by multiple device drivers, some method of linkage is required between the two types of drivers. Typically this is done using some type of I/O “switch” structure, as described in “The I/O Switch Tables” on page 48, 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 `wsio_intr_alloc()` function, executed in your `driver_attach()` or `driver_if_init()` routine. Commonly, `driver` is replaced by your driver's name.

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

```c
int wsio_intr_alloc (struct isc_table_type * isc,
                   wsio_drv_isr_t driver_isr,
                   uintptr_t arg,
                   uint64_t flags,
                   wsio_intr_object_t * wsio_intr)
```
The first parameter is a pointer to the \texttt{isc_table} entry that represents the device. The next two arguments, \texttt{driver_isr} and \texttt{arg} are the driver's \texttt{isr} routine and the argument that will be passed into the \texttt{driver_isr} when it is called. Typically the driver specifies the \texttt{isc} pointer as the argument. The next argument is a flags field. The flags can be used to indicate if the driver wants an exclusive or shared interrupt resource. With shared interrupt a driver's ISR can get called when its device did not interrupt. If this is not acceptable the driver should set the flag to \texttt{WSIO_INTR_EXCLUSIVE}.

The last argument is a pointer to an interrupt handle which \texttt{wsio_intr_alloc()} returns to the driver. The driver must pass this handle into the other WSIO interrupt services such as \texttt{wsio_intr_activate()} which enables the interrupt.

There are additional WSIO interrupt services that allow the driver to specify whether it wants to use a line based or transaction based interrupt. For example, if the driver uses a line based interrupt it would call the service \texttt{wsio_intr_set_irq_line()} to specify a line based interrupt and then \texttt{wsio_intr_get_irq_line()} to get the IRQ line. On the other hand, if the driver wanted to use a transaction based interrupt it would call \texttt{wsio_intr_set_cpu_spec()} to set up a transaction based interrupt and \texttt{wsio_intr_get_txn_info()} to get the transaction based address and vector. Of course, the underlying platform hardware may limit what type of interrupt a driver can use. Some I/O controllers do not allow interface cards underneath them to use transaction based interrupts.

In an interface driver or a monolithic driver, the ISR processes interrupts from an interface card. The ISR performs the following tasks:

- Stops the interface card from interrupting.
- Determines a reason for the interrupt (if appropriate).
- Take appropriate action, such as cleanup or retry.
- Calls \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 `sleep()` or a function that may block. An ISR has the following interface:

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

`arg1` is a driver-defined parameter passed in the call to `isrlink()` as `arg1`.

`driver_isr()` is expected to return the following values:

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

**Examples**

The following are examples of `driver_if_init` and `driver_isr` functions. First, the `driver_if_init` function calls `wsio_intr_alloc()` to allocate an interrupt handle and register its `driver_isr` and `arg`. Then it calls `wsio_intr_set_irq_line()` to specify a line based interrupt and `wsio_intr_activate()` to enable the interrupt. Note that the driver passes the flag `WSIO_IRQ_LINE_AUTO` to `wsio_intr_set_irq_line()` which informs the services that they should determine the IRQ line value for the particular device.

```c
static int CentIF_init (struct isc_table_type * isc)
{
    <<< do any additional driver initialization >>>
    ........................................
    ........................................

    /* Allocate an interrupt object */
    status = wsio_intr_alloc(isc, CentIf_isr, (uintptr_t)isc, 0, &intr_obj);
    if(status != WSIO_OK)
        return(ERROR);

    /* Get a level Sensitive IRQ */
    status = wsio_intr_set_irq_line(isc, intr_obj, WSIO_IRQ_LINE_AUTO, 0);
    if status != WSIO_OK
        return(ERROR);

    /* Active the interrupt */
    status = wsio_intr_activate(isc, intr_obj);
    if(status != WSIO_OK)
        return(ERROR);
}
```
Writing a Driver

Step 7: Writing Other Driver Routines

```c
return(ERROR);
else
    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

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

`dev` Contains encoded major and minor numbers;

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

\[ \begin{align*}
>0 & \quad \text{Successful completion. The value is the swap partition size.} \\
-1 & \quad \text{Error.}
\end{align*} \]

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 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.

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

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

**Data Structures for Managing Interface Cards** Routines that manage interface cards use two data structures, `buf` and `isc_table_type`. You need to be familiar with these data structures, their contents, and their intended usage.

The following sections describe the fields in these structures that are used to manage interface cards.
The **isc_table_type Structure**  The `isc_table_type` structure is defined in `<sys/io.h>`. See `isc_table_type(WSIO4)` in the *HP-UX Driver Development Reference*. The `isc_table_type` structure contains all pertinent information about the interface driver space. This structure is initialized by the WSIO before calling the driver's attach routine. When claiming an interface card the driver will initialize some additional fields such as the `isc->gfsw_init` field.
Writing a Driver

New WSIO Services for 11i

This section describes the new WSIO services that are introduced in 11i. Some of these services replace the functionality provided by older WSIO services, while others provide entirely new functionality. Driver writers are encouraged to convert to the new services, as these services provide enhanced functionality over the older ones.

The new services can be grouped into the following sets:

1. IO Space Services
   - Register Services
   - Configuration Space Services
   - IO Port Space Services
   - Endian Services

2. DMA Services

3. Interrupt Services

4. Memory Allocation Services

5. Driver Event Handling Services

6. System Services
   - Description Service
   - System Attribute Services
   - I/O Synchronization Services

The following provides an overview of each set of services. For detailed information on how to use the services driver, writers should consult the particular man page.

- Register Services - Drivers can call these services to discover, map and access memory mapped registers. Although the first register has already been mapped and is hence readily available to drivers, they can use these services to discover and map additional register sets. These services replace the legacy WSIO services:

  map_mem_to_host()
  unmap_mem_to_host()
Configuration Space Services - This set of services can be used by device drivers to discover and access an IO card’s configuration space. It replaces services provided by other CDIOs such as the PCI CDIO.

IO Port Space Services - This set of WSIO services allows a driver to discover, map and access a card’s IO port space. There are no equivalent legacy WSIO services that they replace.

Endian Services - These WSIO services can be used by device drivers to determine the endianness of the local bus. Drivers can use them to determine whether to perform endian translation for IO port accesses or memory mapped registers.

DMA Services - These new WSIO DMA services replace the older legacy WSIO DMA services (wsio_map, wsio_fastmap, wsio_remap, wsio_unmap, etc.). The advantages of the new services are that they allow drivers to allocate multiple DMA objects and tune each object for different types of DMA. They also allow drivers to register callback functions when resources become available.

Interrupt Services - This new set of WSIO services replaces the older WSIO DMA which include isrlink(). They allow drivers to claim and configure interrupt resources for their cards. With the new services, drivers can allocate multiple interrupt objects per card and configure each one to use different resources.

Description Service - This service can be used by drivers to set the description of a device. The description appears in the ioscan output.

Ordered Interrupt Service - This service can be used by drivers to determine whether the I/O system is ordered with respect to interrupts.

I/O Synchronization Service - This is a function that is used by device drivers to synchronize the CPU and the hardware device’s view of memory.

System Attribute Services - This set of services allows the driver to obtain information about the platform.

Driver Event Handling Services - The event handling services allow drivers to register both an event handler and event mask. The driver’s event handler is called to handle events and the event mask indicates which events the driver is interested in.
Memory Allocation Services - The new WSIO memory allocation services allow drivers to allocate host memory. With the new services a driver can specify different types of memory it wants to allocate, such as memory below four gigabytes (32-bit memory addresses) or physically contiguous memory.

Register Services

These services are used by device drivers to discover and map device registers. The first register set of a device is usually mapped in by the kernel, but the driver can discover additional register sets and map them when using these services. An interface driver would usually do this in its attach or init routine. Registers can contain information about the hardware device or sometimes be used to configure and fine-tune the device. The functions provided are listed and briefly described in the table below. For further information concerning parameters, return codes and example codes, consult that function's man page.

Table 5-7  Register Services

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wsio_map_reg()</td>
<td>Maps a device register to host memory</td>
</tr>
<tr>
<td>wsio_unmap_reg()</td>
<td>Unmaps a device register</td>
</tr>
<tr>
<td>wsio_get_all_registers()</td>
<td>Gets an array of all available device registers</td>
</tr>
<tr>
<td>wsio_read_regXX()</td>
<td>Reads from a mapped device register</td>
</tr>
<tr>
<td>wsio_write_regXX()</td>
<td>Writes to a device register</td>
</tr>
</tbody>
</table>
Configuration Space Services

These services are used by device drivers to discover and use configuration space resources. Configuration space is an I/O space that can contain information about the hardware device as well as some locations that are used to configure and fine-tune the device. The functions provided are listed and briefly described in the table below. For further information on each routine, including parameter information, return code information, and example code, the man page for that function should be consulted.

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wsio_map_cfg_handle()</td>
<td>Obtain a configuration space access handle</td>
</tr>
<tr>
<td>wsio_unmap_cfg_handle()</td>
<td>Release a configuration space handle</td>
</tr>
<tr>
<td>wsio_cfg_inXX()</td>
<td>Reads from configuration space</td>
</tr>
<tr>
<td>wsio_cfg_outXX()</td>
<td>Writes to configuration space</td>
</tr>
</tbody>
</table>

I/O Port Space Services

These services are used by device drivers to obtain access to I/O Port Space. I/O Port Space is an I/O space that is sometimes necessary to communicate with devices. I/O Space accesses do not do any endian checking so, if necessary, a driver will have to perform any necessary endian translation. The functions provided are listed and briefly described in the table below. For further information on each routine, including parameter information, return code information, and example code, the man page for that function should be consulted.
Writing a Driver

New WSIO Services for 11i

Table 5-9  I/O Port Space Services

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

Endian Services

These services are used by device drivers to determine the endianness of the local bus. This service is necessary to decide whether to perform endian translation for shared memory accesses. Such translation is automatically performed for any access to or from configuration space, but for access to registers, I/O ports, or shared memory, the device driver might need to perform their own translation. The functions provided for this service are listed and briefly described in the table below. For further information on each routine, including parameter information, return code information, and example code, the man page for that function should be consulted.

Table 5-10  Endian Services

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSIO_BIG_ENDIAN()</td>
<td>Returns true (1) if the local bus is big endian</td>
</tr>
<tr>
<td>WSIO_LITTLE_ENDIAN()</td>
<td>Returns true (1) if the local bus is little endian</td>
</tr>
</tbody>
</table>
DMA Services

These services are used by device drivers to configure and use DMA resources. DMA is a service that allows memory to be shared between an I/O device and the host processor’s main memory. Although routines already exist in WSIO to use DMA functionality, these new interfaces allow device drivers to: specify callback functions for allocating resources, allocate multiple DMA handles that can each be associated with different sets of hints, and specify the number of bits of addressing that the device will use. The functions provided are listed and briefly described in the table below. For further information on each routine, including parameter information, return code information, and example code, the man page for that function should be consulted.

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wsio_allocate_dma_handle()</td>
<td>Obtain a handle used to setup DMA</td>
</tr>
<tr>
<td>wsio_free_dma_handle()</td>
<td>Releases a DMA handle</td>
</tr>
<tr>
<td>wsio_init_map_context()</td>
<td>Initializes the context used for DMA mapping</td>
</tr>
<tr>
<td>wsio_allocate_shared_mem()</td>
<td>Setup an I/O virtually contiguous DMA buffer</td>
</tr>
<tr>
<td>wsio_free_shared_mem()</td>
<td>Release an I/O virtually contiguous DMA buffer</td>
</tr>
<tr>
<td>wsio_flush_shared_mem()</td>
<td>Flush an I/O virtually contiguous DMA buffer</td>
</tr>
<tr>
<td>wsio_map_dma_buffer()</td>
<td>Maps an existing memory object for packet DMA</td>
</tr>
<tr>
<td>wsio_fastmap_dma_buffer()</td>
<td>Maps an existing memory object for packet DMA</td>
</tr>
<tr>
<td>wsio_remap_dma_buffer()</td>
<td>Maps pre-allocated IOVAs to new host ranges</td>
</tr>
<tr>
<td>wsio_unmap_dma_buffer()</td>
<td>Removes a DMA packet mapping</td>
</tr>
</tbody>
</table>
Writing a Driver

New WSIO Services for 11i

Typically a driver would allocate one or more DMA handles in its driver int routine. It would than use the services

- `wsio_dma_set_service_attributes()` or
- `wsio_set_dma_attributes()` to specify attributes of the type of DMA it will use the object for. A driver may configure one handle for large packet DMA and another for continuous DMA involving small buffers. Later, when setting up a DMA the driver would pass the handle to one of the DMA mapping routines.

### Interrupt Services

These services are used by device drivers to obtain and set up interrupts. These particular services allow a device driver to allocate multiple interrupts, and associate separate interrupt service routines with each. The functions provided are listed and briefly described in the table below. For further information on each routine, including parameter information, return code information, and example code, the man page for that function should be consulted.

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wsio_iova_to_phys()</code></td>
<td>Translate an I/O virtual address to a host virtual address</td>
</tr>
<tr>
<td><code>wsio_set_dma_callback()</code></td>
<td>Sets the callback function and argument for DMA</td>
</tr>
<tr>
<td><code>wsio_dma_pass_thru()</code></td>
<td>Calls a pass-through function that might not otherwise be accessible</td>
</tr>
<tr>
<td><code>wsio_dma_set_service_attributes()</code></td>
<td>Associates DMA hints with a device</td>
</tr>
<tr>
<td><code>wsio_set_dma_attributes()</code></td>
<td>Associates DMA hints with a DMA handle</td>
</tr>
</tbody>
</table>
Writing a Driver
New WSIO Services for 11i

Chapter 5

Description Service

This service is used by device drivers to set their I/O tree description. This service is necessary because in some cases, a meaningful description cannot be figured out automatically. The one function provided for this service is listed and briefly described below. For further information on parameters, return codes and example code, check the man page.

Table 5-12

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wsio_intr_alloc()</td>
<td>Allocate a WSIO interrupt object</td>
</tr>
<tr>
<td>wsio_intr_free()</td>
<td>Free a WSIO interrupt object</td>
</tr>
<tr>
<td>wsio_intr_active()</td>
<td>Enable a WSIO interrupt object</td>
</tr>
<tr>
<td>wsio_intr_deactivate()</td>
<td>Disable a WSIO interrupt object. This call blocks</td>
</tr>
<tr>
<td>wsio_intr_deactivate_nowait()</td>
<td>Disable a WSIO interrupt object with callback. Non-block</td>
</tr>
<tr>
<td>wsio_intr_set_cpu_spec()</td>
<td>Set up transaction based interrupts</td>
</tr>
<tr>
<td>wsio_intr_set_irq_line()</td>
<td>Set up line based interrupts</td>
</tr>
<tr>
<td>wsio_intr_get_assigned_cpu()</td>
<td>Get the assigned CPU</td>
</tr>
<tr>
<td>wsio_intr_get_irq_line()</td>
<td>Get the IRQ line</td>
</tr>
<tr>
<td>wsio_intr_get_txn_info()</td>
<td>Get the transaction address and data values</td>
</tr>
</tbody>
</table>

A driver would call this in its init routine.

Table 5-13

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wsio_set_description()</td>
<td>Sets the I/O tree node description for this driver</td>
</tr>
</tbody>
</table>
Ordered Interrupts Service

This service is used by device drivers to determine whether interrupts are ordered with respect to DMA transactions. This tells device drivers whether it is necessary to explicitly perform a sync to ensure that DMA transactions have completed. The one function provided for this service is listed and briefly described below. For further information on parameters, return codes and example code, check the manpage.

Table 5-14 Ordered Interrupts Service

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSIO_ORDERED_INTERRUPTS()</td>
<td>Indicates whether interrupts are ordered with respect to DMA transactions</td>
</tr>
</tbody>
</table>

I/O Synchronization Service

This service is used by device drivers to explicitly synchronize the CPU and the I/O device’s views of memory. If such a synchronization is not necessary, this function call will avoid doing it. The one function provided for this service is listed and briefly described below. For further information on parameters, return codes and example code, check the man page.

Table 5-15 I/O Synchronization Service

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wsio_io_sync()</td>
<td>Performs a sync of shared memory if necessary</td>
</tr>
</tbody>
</table>
System Attribute Services

These services are used by device drivers to obtain information about specific system features. Currently, only information about cacheline size, default page size, and the number of CPUs in a system is available. The functions provided for these services are listed and briefly described below. For further information on parameters, return codes and example code, check the man page.

Table 5-16 System Attribute Services

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wsio_get_system_params()</td>
<td>Obtains information about the system</td>
</tr>
<tr>
<td>wsio_get_processor_count()</td>
<td>Number of processors on the system</td>
</tr>
<tr>
<td>wsio_get_active_processor_count()</td>
<td>Active number of processors on the system</td>
</tr>
<tr>
<td>wsio_get_cpu_name()</td>
<td>Name (Address) of a CPU</td>
</tr>
</tbody>
</table>

Driver Event Handling Services

These services allow drivers to register an event handler and event mask with the WSIO. When an event occurs and the event mask indicates that the driver is interested in that type of event it’s handler is called. Events include PCI OLAR actions such as suspend, resume or remove.

Event handlers are registered on a per driver basis. This is usually done in the driver's install routine. The mask, however, is registered by the driver on a per instance basis. This is done in the driver's init routine after claiming an I/O card. Using this method, drivers can register a single event handler, but specify what events it can handle on a per-instance basis.

When an event impacts an I/O device the WSIO first checks to see if there is an event handler associated with the device. If there is, it then checks the event mask to see if the handler will respond to that type of event. If it does, it then calls the handler.
All driver handlers must have the following caller syntax:

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

For more information on the driver handler and the structure `wsio_generic_event_t`, see the manpages.

### Table 5-17

**Driver Event Handling Services**

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

### Memory Allocation Services

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

The second two services, `wsio_alloc_mem()` and `wsio_free_mem()` are used to allocate and free memory using the memory allocation handle created by the service `wsio_alloc_mem_handle()`.

Typically drivers would call `wsio_alloc_mem_handle()` in their init routines to specify the type of memory they want to allocate. The service will return a handle that the driver would then pass into the services `wsio_alloc_mem()` and `wsio_free_mem()` when allocating and free buffers.

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

For more specific information on how to call these services and the parameters passed in see the man pages for each in the DDR.
### Table 5-18 Memory Allocation Services

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

PCI OLAR is available in the 11i version of HP-UX for those platforms that have hardware support. Drivers that want to take advantage of this feature must provide an event handler and register an event mask for each interface card they claim. The handler is called by the WSIO CDIO to take action when a PCI OLAR event occurs that impacts a card owned by the driver. It is only called if the driver specified an interest in the event via the registered event mask for the affected card.

For 11i the type of events that a driver can register an interest in are:

- WSIO_EVENT_SUSPEND
- WSIO_EVENT_RESUME

Drivers register their handlers with the WSIO in the driver install routines. The event mask is registered later in the driver attach routine each time it claims an I/O card.

When a driver handler is called with an event of type WSIO_EVENT_SUSPEND it should suspend the activities of the card, including DMA and interrupts. It does not have to give up any long term resources like MMIO space or memory buffers for long term mappings.

When the driver handler is called with an event type of WSIO_EVENT_RESUME it should resume the activities of the specified driver instance.

For 11i HP-UX supports PCI card replacement. This means that a driver instance can be SUSPENDED, the card replaced, and the driver instance RESUMED.

Registering a Driver Handler

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

```c
int wsio_install_drv_event_handler __
    ((wsio_drv_info_t * info_ptr,
      wsio_drv_event_handler_t handler));
```
The first parameter is a pointer to the driver's `wsio_drv_info_t` structure and the second parameter is a pointer to the driver handler. The driver handler should have the following calling interface:

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

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

```c
typedef struct wsio_generic_event {
    wsio_event_t event; /* suspend, resume, and so on */
    /* Event_id is a tag to identify an instance of an event. 
    * Driver just passes it back in the callback */
    wsio_event_id_t event_id;
    struct isc_table_type *isc;
    generic_complete_callback_t wsio_completion_cb;
    void *arg;
} wsio_generic_event_t;
```

The first field specifies the type of event that the driver handler is being called for. For PCI OLAR the following types of `wsio_event_t` are or will be defined.

```
WSIO_EVENT_SUSPEND
WSIO_EVENT_RESUME
```

The second argument, `event_id`, is used to identify a specific event. The third argument is the `isc` structure of the card. The fourth argument, `wsio_completion_cb`, is a callback function that the driver must call when it completes the event to report status. The driver must report the status of an event asynchronously via this callback mechanism. The calling interface for the callback function is:

```c
int (*wsio_completion_cb)( struct isc_table_type *isc,
                          wsio_event_id_t event_id,
                          void *status );
```
The first parameter is the `isc` of the driver instance, the second is the value of the `event_id` field passed into the driver handler and the third is the status of the action. If the handler successfully completed the event it should set the status to `WSIO_OK`, otherwise it should set it to either `WSIO_UNSUPPORTED_EVENT` or `WSIO_ERROR`. The former is returned if the event is not supported by the driver, the latter for a failure to complete the event successfully. The following example shows a driver install routine:

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

static wsio_drv_info_t my_drv_info {
    &my_drv_info;
    &my_drv_ops,
    &my_drv_data,
    WSIO_DRV_CURRENT_VERSION,
}

my_driver_install()
{
    wsio_install_driver (&my_drv_info);

    if ( (wsio_install_drv_event_handler(&my_drv_info,
        my_drv_handler))
        != WSIO_OK)
    {
        wsio_uninstall_driver(&my_drv_info);
        return(0);
    }

    /*******************************
    * Let's go ahead and link the driver's attach
    * routine into the global PCI attach list
    *******************************/
    my_drv_saved_attach = pci_attach;
    pci_attach = my_driver_attach;
    return(1);
}
```
Registering a Driver Event Mask

Drivers that register handlers must register an event’s capabilities mask for each interface card they claim. This mask indicates what events their handlers are interested in. The driver registers the capabilities mask by calling the WSIO service `wsio_reg_drv_capabilities_mask()`. The calling interface for the service is:

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

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

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

Setting Event Timeout Values

The WSIO CDIO sets a timeout when it calls the driver handler. If the handler does not reply via the callback function before the timeout pops, the WSIO assumes an error of some sort has occurred, and reports this. Drivers can change the value of the timeout by calling the WSIO service `wsio_set_parm()` in their attach routine. They can also query the default value for the timeout by calling the WSIO service `wsio_get_parm`. They can get and set the values for each type of event, `WSIO_EVENT_SUSPEND` or `WSIO_EVENT_RESUME`. The calling interfaces for these WSIO services are:

```c
int wsio_get_parm( struct isc_table_type * isc, 
                   wsio_parm_t I_parm, 
                   void **I_value);

int wsio_set_parm( struct isc_table_type *I_isc, 
                   wsio_parm_t I_parm, 
                   void *I_value);
```
The first parameter is a pointer to an `isc` structure of the driver instance. The second is of type `wsio_parm_t` which indicates what value is being set or queried. Valid values for this parameter are:

- `WSIO_HW_SUSPEND_TIMEOUT` - Hardware Suspend Timeout parameter
- `WSIO_HW_RESUME_TIMEOUT` - Hardware Resume Timeout parameter.

The final parameter is the value to be set or returned. For `wsio_get_parm()` the current value is returned in this parameter. For `wsio_set_parm()` the caller passes in the current value.

See the man pages in the DDR for more details on calling these services.

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

```c
int my_driver_attach(uint32_t id, struct isc_table_type * isc) {
    int ret;
    wsio_event_mask_t my_task;
    ...... do any other driver/instance processing ....

    /**********************************************************************************
    * Let’s claim the card and register an event’s capabilities
    **********************************************************************************/
    isc_claim(isc, &my_drv_info);

    /**********************************************************************************
    * The return value should be either WSIO_OK which indicates that the mask was registered or WSIO_HA_NA which indicates that the platform does not support PCI OLAR. If it’s not one or the other we’ve hit some other unknown condition, so let’s just bail.
    **********************************************************************************/
    my_mask = WSIO_EVENT_SUSPEND | WSIO_EVENT_RESUME;
    ret = wsio_reg_drv_capabilities_mask(isc, my_mask);
    if( (ret != WSIO_OK) && (ret != WSIO_HA_NA) ) {
        ....... free any allocated resources for ...... this card instance. .......
        isc_unclaim(isc, &my_drv_info);
    } else
}
Writing a Driver
Support for PCI OLAR

Chapter 5 181

{  
    /****************************************************************************
    * Let's modify the timeout values for
    * WSIO_EVENT_SUSPEND and WSIO_EVENT_RESUME.
    * The default values are 1 second. Timeout
    * values are given in microseconds.
    ***************************************************************************/
    wsio_set_parm(isc, WSIO_HW_SUSPEND_TIMEOUT, 2000000);
    wsio_set_parm(isc, WSIO_HW_RESUME_TIMEOUT, 1500000);

    return((*my_drv_saved_attach)(id, isc));
}

Driver Event Handler

This function has one input parameter, a pointer to a data structure. The
content of fields in the structure is dependent on the event. Most error
handlings in the example are left out for simplicity. A driver will return
the status of its operation in the callback function: WSIO_OK for success,
WSIO_UNSUPPORTED_EVENT for unknown request, and WSIO_ERROR for
failure (in this case, a driver should log messages to dmesg buffer).

#include <wsio/wsio.h>

/* Following is a sample of a global data structure to
 * temporarily save the WSIO's isc, hint, event_id, and
 * callback function. The algorithm for a suspend event is
 * like this: 1) The event handler is called, 2) The
 * handler stores away the isc, hint, even_id, and
 * callback function in this global data structure
 * 3) Call Ktimeout() to schedule another function,
 *  my_suspend(), to really do the suspension. 4) Event
 * handler returns right away. This is just an example.
 * This structure can be pointed to by one of the fields
 * in the isc pointer. The mechanism to implement the
 * handler is driver dependent. */

typedef struct my_suspend_information {
    struct isc_table_type *isc;
    wsio_event_id_t event_id;
    wsio_suspend_info_t hint;
    int (*complete_cb) (struct isc_table_type *isc;
        wsio_event_id_t event_id, void *status);
}
my_suspend_information_t;  
my_suspend_information_t *my_suspend_info;

/* This my_suspend() function is invoked by Ktimeout().  
* Its input parameter is a pointer to  
* my_suspend_information_t. The event handler schedules  
* the Ktimeout() call for this function */

void my_suspend(my_suspend_information_t *my_suspend_op) {
    struct isc_table_type *isc=my_suspend_cp->isc;
    wsio_suspend_info_t hint = my_suspend_cp->hint;

    /* Tell the card to stop interrupt */
    /* Do whatever is needed */

    if error, my_suspend_cp->complete_cb(isc,
             my_suspend_cp->event_id, (void *)WSIO_ERROR)

    /* The last parameter of complete_cb() is void *status. WSIO  
    * will typecast it according to what event is being handled.  
    * For suspend, resume, and remove, drivers should return a  
    * define, such as WSIO_OK, WSIO_UNSUPPORTED_EVENT, or  
    * WSIO_ERROR. It can be typecasted as (void *) to avoid  
    * compiler warning */

    my_suspend_cp->complete_cb(isc, my_suspend_cp->event_id,
                           (void *)WSIO_OK)
}

/* The following event handler first stores away the  
* event_id, isc, call back function, and hint into a global,  
* my_suspend info. Then it calls Ktimeout() with a timeout  
* value of 0 to schedule my_suspend(0 to run. The function  
* return right away. A driver can decide on a timeout value  
* depending on whether it wants to accomodate any in-progress  
* activities. This is just an example. A driver can design  
* the best approach to fit its requirement. The event ID is  
* tag that WSIO generates to prepare for future use (match up  
* request with a reply); driver should return in in the  
* completion callback. */

void my_handler(wsio_generic_event * handler_arg) {
    switch (handler_arg->event) {
    case WSIO_EVENT_SUSPEND:

/* Store away the values in signal my_suspend_info */

my_suspend_info->event_id = handler_arg->event_id;
my_suspend_info->isc = handler_arg->isc;
my_suspend_info->complete_cb =
    handler_arg->wsio_completion_cb;
my_suspend_info->hint = handler_arg->arg;

/* Schedule the call to my_suspend() to run immediately */

Ktimeout(my_suspend, my_suspend_info, 0, NULL);
break;

case WSIO_EVENT_RESUME:

/* Saving of my_resume_info not shown */

Ktimeout(my_resume, my_resume_info, 0, NULL);
break;

case default:

    handler_arg->wsio_completion_cb(handler_arg->isc,
        handler_arg->event_id,
        (void *) WSIO_UNSUPPORTED_EVENT);
    break;
}
return;
6 Installing Your Driver
This chapter describes how to build your driver into the kernel. For information about demand loadable module drivers see Chapter 12, “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=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) manpage for more details. For the mydriver example, change the Makefile line

   from:   CFLAGS= -w ${COPTS} $(K_CCOPTS)
   to:     CFLAGS= -Ae -w ${COPTS} $(K_CCOPTS)

5. If 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 +DA2.0. 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/usr/conf -I. -c -o mydriver.o \  
   -Ae -w -U__hp9000s700 -D__HIGHC__ -D__STDC_EXT__ \  
   -D_XPG4_EXTENDED -D_HPUX_SOURCE -D_UNSUPPORTED \  
   -D__hp9000s800 -D_KERNEL -DKERNEL +ES1.Xindirect_calls \  
   -Wp,-H300000 +XixdU +Hx0 +R500 -Wl,-a,archive +DA1.1 \  
   +DS2.0 +ESsfc +ESssf \  
   mydriver.c
If you are using an ANSI C compiler and building a 64-bit driver:

```
/opt/ansic/bin/cc -I/usr/conf -I. -c -o mydriver.o \
-Ae -w -U__hp9000s700 -D__HIGHC__ -D__STDC_EXT__ \n
-D_XPG4_EXTENDED -D_HPUX_SOURCE -D_UNSUPPORTED \n
-D__hp9000s800 -D_KERNEL -DKERNEL +ES1.Xindirect_calls \n
-Wp,-H300000 +XixdU +Hx0 +R500 -Wl,-a,archive +DA2.0 \n
+DS2.0 +ESsfc +ESssf \n
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 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.
Step 2: Create a Master File

- If your driver has dependencies on other drivers, list these other drivers in the $DRIVER_DEPENDENCY table in the master file. The format is:
  
  driver_name otherdriver1 otherdriver2 ...

- Use the $DRIVER_LIBRARY table to specify libraries that contain functions 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.
Installing Your Driver

Step 2: Create a Master File

Chapter 6 193

* @(#) $Revision: 1.1.106.2 $

* The following devices are those that can be specified in the
* system description file. The name specified must agree with
* the name shown, or with an alias.
* name          handle         type    mask   block    char
*
* $DEVICE
*$

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

$DRIVER_INSTALL
************************************************************
* Driver install table
*
* This table contains the name of drivers which have converged
* I/O header structures and install entry points. Drivers in
* this table should not be defined in the driver table above.
*
* Note : as of 10.30, lan0 and lan1 are obsolete
************************************************************
*
* Driver    Block major   Char major
* mydriver             -1           -1
*
*$

$ALIAS
*
* The following entries form the alias table.
* field 1: product # field 2: driver name
* $$ $TUNABLE
*
* The following entries form the tunable parameter table. Any
* of these values can be overwritten in the dfile. However,
* before changing any value, know the ramifications of your
* change.
* $$
*$

$DRIVER_DEPENDENCY
*
Installing Your Driver

Step 2: Create a Master File

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

$DRIVER_LIBRARY

* The driver/library table. This table defines which libraries a given driver depends on. If the driver is included in the dfile, then the libraries that driver depends on will be included on the ld(1) command line. Only optional libraries need to be specified in this table, (but required ones can be included, as well).
  * Driver handle <libraries>
  *
  * subsystems first $$$

$LIBRARY

* The library table. Each element in the library table describes one unique library. The flag member is a boolean value, it is initialized to 1 if the library should *always* be included on the ld(1) command line, or 0 if the library is optional (i.e. it is only included when one or more drivers require it). The order of the library table determines the order of the libraries on the ld(1) command line, (i.e. defines an implicit load order). New libraries must be added to this table.
  * Note: libhp-ux.a must be the last entry, do not place * anything after it.
  *
  * Library <required>
  *
  $$$

$SUBSYSTEMS_DEFINE

* Subsystem #define table. For any subsystem that is found in the dfile and appears below, a #define is generated in the conf.c file. This is used to include any space defined in space.h. $$$
$DRIVER_PRODUCT
*
* Driver product table. The following generates a table of
* supported products for a given interface card. The input is a
* driver handle followed by a list of product/vendor id’s. What
* is produced is a table of these product/vendor id’s to be
* used by the interface driver for further qualification during
* booting and the setting of root device location.
* $$

$$
Step 3: Modify the System File

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. There are two methods for building the kernel.

Using mk_kernel:

Use the following command to build the kernel containing your driver.

```bash
mk_kernel -s system -o vmunix
```

mk_kernel builds an executable file which can be used as a bootable kernel and kernel modules if any are configured. If the build succeeds, the newly built kernel is called `vmunix_test`. The file and directory are placed in the build directory, as defined below.

If the path used to designate the system file is `/stand/system`, the build directory is `/stand/build`. If another path is used to designate the system file, the build directory is the current working directory.

If `-o /stand/vmunix` is specified, the target kernel file and kernel function set directory are not overwritten. The new kernel file and the kernel function set directory are moved to the default path as the system shuts down or starts up. The previous versions of the file and directory are renamed `/stand/vmunix.prev` and `/stand/dlkm/vmunix.prev`. Until the system reboots, the new kernel file and the directory must be kept as `vmunix_test` and `dlkm/vmunix_test`, respectively.

If the `-o` option is specified with other than `/stand/vmunix`, the kernel file and kernel function set directory are created or updated immediately. In case the administrator needs to place these targets to the system default path, the `kmupdate` command must be used to trigger the replacement. Manually replacing the default kernel (`/stand/vmunix`) or any file under the kernel function set directory (`/stand/dlkm`) must be avoided.

Using config and make:

1. Use the following command to generate the files used to create a new kernel containing your driver.

```bash
/usr/sbin/config -s system
```
Installing Your Driver

Step 4: Build a Kernel Containing Your Driver

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 XOBJS=, 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:

  ```bash
  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_test 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:

  ```bash
  $(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:
Installing Your Driver

Step 4: Build a Kernel Containing Your Driver

$(ROOT)/libcdfs.a $(ROOT)/libhp-ux.a
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_test 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 dfile, the libraries that driver
* depends on will be included on the ld(1) command
* line. Only optional libraries *need* to be specified
* in * this table, (but required ones can be included, as
* 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). 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:

   ```bash
   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 ioscanc 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).

   ```bash
   /usr/sbin/mknod /dev/mydriver b 65 0x026000
   /usr/sbin/mknod /dev/mydriver c 234 0x026000
   ```

5. Verify the configuration by invoking ioscanc with the -fun or -fkn options. If the device-special files are created properly, ioscanc displays them beneath the configured device.
For information on packaging third-party drivers for distribution with HP-UX, see *Managing HP-UX Software with SD-UX*.
Installing Your Driver

Step 5: Create a Device-Special File
Chapter 7

Creating Networking Device Drivers
This chapter provides information for designing and writing PCI networking device drivers.

The information in this chapter is intended for developers with extensive experience in designing and writing networking device drivers for non-HP UNIX target systems.

Basic STREAMS module/driver development and general networking concepts and RFCs are not included in this documentation.

The major difference in the network device driver model from HP-UX 10.20 for third party developers is that third party developers are expected to implement their own DLPI layer and not depend on or use the HP DLPI. Also, HP supports “pure” STREAMS model drivers; BSD style drivers are not supported anymore.

The first section contains an overview of the structure of networking drivers. 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 sample driver provided in your driver development kit, to create your PCI networking device driver.
HP-UX Networking Interface Architecture

This section describes the HP-UX networking interface architecture for the PCI bus. The interface supports OSI protocols, Internet protocols, and DLPI protocols on HP-UX platforms.

The HP-UX networking subsystem comprises three logical layers, as shown in Figure 7-1, “Three layered HP-UX Interface to the PCI Bus,” and is briefly described in the following four subsections:

“Data Link Interface Layer” on page 209
“Network Protocol Layer” on page 209
“Protocol Interface Layer, Device File, and STREAMS Head” on page 209
“STREAMS Environment” on page 210
Figure 7-1 Three layered HP-UX Interface to the PCI Bus
Data Link Interface Layer

Data Link layer has STREAMS DLPI drivers. A DLPI driver interacts with STREAMS modules in the system. The network interface part of the driver is responsible for manipulating its hardware devices (e.g., Ethernet cards) and for encapsulating/decapsulating link level (e.g., SNAP) headers that are required to deliver messages to a destination. The data link layer:

- Directly connects to the network interface hardware (network interface, physical layer).
- Consists of the hardware interfaces and their respective device drivers.
- Implements DLPI Version 2.0 to interact with STREAM/UX Transport Stack.

Network Protocol Layer

The network protocol layer, above the datalink interface layer, encompasses four protocol families:

- Internet: TCP/IP, UDP/IP
- OSI
- X.25
- ARP

Each network protocol family belongs to a domain and uses the address scheme in that domain. For example, the Internet (INET) family of protocols form the Internet domain.

The network protocols of other domains, such as the OSI stack, may be functionally equivalent to the Internet stack, but are generally not compatible with Internet domain protocols.

Protocol Interface Layer, Device File, and STREAMS Head

This interface layer directly supports applications; its main functions are to:
Identify different applications on the same host (for example, a socket interface or a device file interface).

Provide services from transport layer protocols and to applications.

The interface for this layer provides the following abstract objects that applications can create, open, connect, or use for sending or receiving data:

- Sockets
- Streams
- Device files

**STREAMS Environment**

The kernel modules for the HP-UX transports (e.g. TCP/IP, UDP, OSI) are now STREAMS modules. Drivers that interface to the transport stacks must now work within this environment.

Driver writers should refer to the following documents for information concerning STREAMS modules and device drivers. Attention should be paid to the DLPI references. This document only briefly discusses the STREAMS mechanisms and concentrates on specific HP variants.

The following documents are recommended sources:

Hewlett-Packard Company:


Other References:

- *Data Link Provider Interface Specifications*, Unix International
Overview of Networking Driver Structure

The flowchart in Figure 7-2, “Steps to Develop a Networking Driver,” shows a suggested sequence to use when developing networking drivers on HP-UX systems. Step one lists the mandatory information necessary, or standard knowledge base, for a basic driver. Steps two through eight list the options available for increased network driver capabilities. The sequence of information in the flowchart closely follows the organization of this chapter. Refer to each step's description for pointers to its applicable detailed subject areas.
Creating Networking Device Drivers

Overview of Networking Driver Structure

Figure 7-2  Steps to Develop a Networking Driver

STANDARD KNOWLEDGE

1. HP-UX Network Interface Architecture
   Data Structures
   Protection & Synchronization
   Network Driver Installation

2. Aux. Code?
   Y
   Auxiliary Code

3. N/W Mgmt Spprt?
   Y
   Network Mgmt Support

4. Log & Trace Spprt?
   Y
   Logging & Tracing Support

5. Aux. Files?
   N
   Y

OPTIONS

1. Y
   N
   mblk and queue macros
   DLPI Interface
   STREAMS DLPI Network Driver

6. SAM Spprt?
   Y
   SAM Support

7.Drvradm Spprt?
   Y
   Driveradmin Support

8. Lanscan Spprt?
   Y
   Lanscan Support

DRIVER COMPLETE
1. This step in the network driver development lists the mandatory knowledge base needed to tailor the driver basic functions. The topics are:

**HP-UX Network Interface Architecture**
An overview of the STREAMS environment. Refer to “STREAMS DLPI and Network Driver Overview” on page 242.

**Data Structures**
Describes the data structures in the networking interface layer: `hw_ift_t`, `hw_dlpi_t` and device driver data structure framework. Refer to “Data Structures and Interfaces” on page 216 for detailed information about these data structures.

**Protection and Synchronization**
Describes the OSF/Encore spinlock protection model. Refer to “Protection and Synchronization for Driver and Transport” on page 222 for more detailed information about supporting the spinlock scheme in HP-UX.

**Network Driver Initialization**
Describes the `install` and initialization routines for the STREAMS DLPI driver. The `attach` routine is discussed for the driver. Refer to “Initializing Networking Device Drivers” on page 224 for detailed information about these routines.

**Protocol Configuration, Binding, and Demultiplexing**
Describes configuration of the INET stack for the STREAMS model drivers. Also, the routines for the driver to bind and demultiplex upper layer protocols to a device are explained. Refer to “Protocol Binding and Demultiplexing” on page 233 for detailed information on these routines.

**mblk and queue macros**
These are macros commonly used by STREAMS networking drivers. Refer to “Message Block and Queue Functions and Macros” on page 238.
Creating Networking Device Drivers

Overview of Networking Driver Structure

DLPI Interface
Describes how upper layers are linked to the network drivers via the DLPI. Refer to “DLPI Interface to Upper Layers” on page 239.

STREAMS DLPI Network Driver
Provides an overview of the DLPI and WSIO interface portions of the STREAMS DLPI network driver. Major driver functions are also explained. Refer to “STREAMS DLPI and Network Driver Overview” on page 242 for more detailed information.

The following steps list the options available when developing a network driver.

2. Auxiliary Code

HP customers expect to have network management and tracing and logging support in their networking products. HP recommends adding these routines to your network driver.

If selected, implement the code, then proceed to the next option. If not selected, go to step five.

3. Network Management Support

A description of the routines that support Network Management requests. Refer to “Network Management Support” on page 276 for more detailed information.

Select or go to the next option.

4. Logging and Tracing Support

A description of the routines that support Logging and Tracing. Refer to “Formatting Networking Trace/Log Messages” on page 299 for more details.

Select or go to the next option.

5. Auxiliary Files

HP customers expect to have automated configuration through the System Administration Manager (SAM) and be able to display link and encapsulation statistics and tracing and logging messages.

If Auxiliary Files are not required, the driver is complete. If they are required, go to the next option.
6. SAM Support

Describes the changes required by the driver for SAM and the configuration files necessary to support the menu driven utility when configuring a networking driver. See “Configuring a Networking Driver Through SAM” on page 303 for details.

Select or go to the next option

7. driveradmin files

Describes the shared library for the driveradmin command. This is used to display link statistics and perform administration tasks. Refer to “Network Monitoring Commands” on page 314 for detailed information.

Select or go to the next option.

8. lanscan support

Describes the shared library for the lanscan command. This is used to display link encapsulation information. Refer to “Network Monitoring Commands” on page 314 for more information.

The driver is now complete.

STREAMS Model Device Drivers

Starting with HP-UX 11.0, IHVs and ISVs are expected to write their own DLPI layer implementation in the STREAMS network interface driver. A network driver in HP-UX 11i is a native STREAMS DLPI driver. This document provides a framework that includes a native STREAMS DLPI PCI network interface driver, enet, which has a sample DLPI implementation and the device interface part, as part of the driver development kit.

NOTE

The names STREAMS DLPI driver, native STREAMS DLPI, native DLPI driver and DLPI driver are used interchangeably in this chapter.
Data Structures and Interfaces

The following data structures are used by the network interface layer:

- **hw_ift_t** (defined in `sio/lan_dlpikrn.h`)
- **hw_dlpi_t** (contained in `hw_ift_t`; defined in `sio/lan_dlpikrn.h`)

Each device driver may maintain its `hw_ift_t` and `hw_dlpi_t` structure as part of a larger structure, the driver control block `enet_ift_t` (shown in Figure 7-3 on page 217). 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 interface to the network system utilities `lanscan` (see `lanscan (1M)`), `driveradmin`, and `driverlinkloop` to display detailed information for all network devices. The `hw_ift_t` structure is described below.

```c
typedef struct hw_ift {
    hw_dlpi_t hp_dlpi;
    uint32_t mac_type;
    uint32_t llc_flags;
    uint32_t mjr_num;
    uint32_t nm_id;
    uint32_t instance_num;
    uint32_t mtu;
    char *name;
    uint8_t hdw_path[MAX_HDW_PATH_LEN];
    uint32_t hdw_state;
    uint32_t mac_addr_len;
    uint8_t mac_addr[MAX_MAC_ADDR_LEN];
    uint32_t features;
    uint8_t *arpmod_name;
    uint32_t ppa;
    uint32_t watch_timer;
    uint32_t reserved2;
    lock_t *hwift_lock;
    struct hw_ift *next;
} hw_ift_t;
```
The following fields must be properly initialized by the device driver during system initialization to support the HP-UX system utilities.

- **hp_dlpi**: Must be initialized to all zeros.
- **mac_type**: Device type; see “MAC Types and Protocol Types” in Appendix A for network media types.
- **llc_flags**: Link Level Control (LLC) encapsulation method. Defined flag values and the encapsulation method are listed in “MAC Types and Protocol Types” in Appendix A.
- **mjr_num**: Major number of the device file. The major number should be set to -1.
- **nm_id**: Network management ID; should be initialized via a call to the `get_nmid()` routine.
- **instance_num**: Device instance number; the value returned by calling the `wsio_isc_to_instance()` routine.
- **mtu**: Maximum transmission unit (number of bytes) for a particular type of link or encapsulation; see “MTU Values” in Appendix A for a list of predefined values.
Creating Networking Device Drivers
Data Structures and Interfaces

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Driver device name; used for naming shared libraries for lanscan and driveradmin.</td>
</tr>
<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; zero, if the device is OK. If the device is not available, <code>hdw_state</code> is set to <code>LAN_DEAD</code>.</td>
</tr>
<tr>
<td>mac_addr_len</td>
<td>Number of bytes of <code>mac_addr[]</code> 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 the device. The following flags are provided:</td>
</tr>
<tr>
<td></td>
<td><strong>DRV_MP</strong> Set this flag and make sure the device driver is MP scalable or MP safe; that is, uses <code>spinlock()</code> or <code>spinunlock()</code> to avoid race conditions. See “Protection and Synchronization for Driver and Transport” on page 222 for more information. When this flag is set, the driver cannot use any spl® calls.</td>
</tr>
<tr>
<td></td>
<td><strong>DRV_MBLK</strong> This flag must be set; the third party network driver is purely based on STREAMS model.</td>
</tr>
<tr>
<td></td>
<td><strong>DRV_IP_MULTICAST</strong> This flag must be set if a driver supports the IP multicast feature.</td>
</tr>
<tr>
<td></td>
<td><strong>DRV_LANC_PROMISC_SUPPORT</strong> This flag must be set if a driver supports promiscuous listening.</td>
</tr>
</tbody>
</table>

**NOTE:**
The driver names `lan` and `fddi` are reserved for HP devices.
DRV_NO_FAST_PATH
This flag must be set if a driver does not support fast path as described in “Transmission of Message Blocks” on page 254.

DRV_CKO
This flag must be set if a driver supports TCP or UDP checksum calculations in hardware.

arpmod_name
The name of ARP STREAMS helper module. This module complements the generic ARP module to resolve addresses in networks like Token Ring and Fiber Channel.

ppa
Physical Point of Attachment (PPA) number for the interface. The driver should initialize this field with hw_ift->instance_num.

watch_timer
For Hewlett-Packard internal use only. This field must be set to zero for non-Hewlett-Packard devices.

reserved2
For Hewlett-Packard internal use only. This field must be set to zero for non-Hewlett-Packard devices.

hwift_lock
Pointer to a hwift_lock spinlock structure to protect the hw_ift structure. This field is initialized in hw_ift_attach().

next
Pointer to next hw_ift structure in list. This field is set by calling the hw_ift_attach() routine during device driver initialization. See “Initializing Networking Device Drivers” on page 224 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->hdw_state = LAN_DEAD;
hw_ift_ptr->mac_addr_len = 6;
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; it 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 avoiding data corruption and race conditions when shared structures are accessed by multiple threads in MP systems. Driver data structures also need protection against interrupts. HP-UX transport networking adopted the OSF/Encore spinlock protection model to gain parallelism and provide scalable network performance. The spinlock scheme provides finer granularity locks, protecting data structures at finer levels, as opposed to grabbing a global network lock. More information on spinlocks is available in Chapter 4, Multiprocessing 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. 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 hwiflash spinlock.

hw_ift_ptr: pointer to an hw_ift structure.

NOTE

The hwiflash spinlock is allocated and initialized by the hw_ift_attach() routine. As a result, the HW_IFT_LOCK() and HW_IFT_UNLOCK macros are not available until returning from the hw_ift_attach() routine.

Driver Structure Protection

Networking drivers use spinlocks to protect their internal data structures. HP-UX predetermines the order (major order) for spinlocks for LAN and STREAMS drivers to avoid deadlock conditions when non-direct code paths are executed due to faults, traps, or interrupts.

Drivers can increase concurrency with finer granularity locks. The major lock order is predefined by HP-UX so drivers can use different minor order spinlocks to protect access to data structures. For example, a network interface driver can use one lock for transmit path and another for receive path data structures. This allows the driver to receive and transmit concurrently.

A list of the relative predefined lock orders for spinlocks used by HP-UX LAN products is shown 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 hwiflash, 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 version 11i, developing a network interface driver involves developing a STREAMS DLPI network interface driver. A DLPI driver is part of STREAMS/UX and is used by the file system for device open and close. For this reason the DLPI driver is both a STREAMS and WSIO-CDIO driver. Initialization for a DLPI network driver is described in this section.

For a detailed description of generic STREAMS driver development, refer to the STREAMS/UX for HP9000 Reference Manual. This section explains the initialization process with the help of excerpts from a sample driver 
et.

The install routine of a STREAMS DLPI driver, 
driver_install()
should call WSIO-CDIO install 
wsio_install_driver() and
STREAMS/UX install 
str_install() functions.

The WSIO-CDIO system requires the following data structures to be defined and initialized before calling wsio_install_driver() in

driver_install().

drv_ops_t enet_drv_ops;
drv_info_t enet_drv_info= {
    "enet",    /* driver name */
    "pseudo"   /* driver clas */
    DRV_CHAR|DRV_PSEUDO|DRV_MP_SAFE,
        /* type */
    -1,       /* block major number */
    -1,       /* character major number */
    NULL,NULL,NULL        /* always NULL */
};

STREAMS/UX requires that streams_info_t be initialized as shown in
the following code sample. This structure is passed in the call

str_install()

static struct module_info enet_rminfo= {
    5050, "enet", 0, 65536, 65536, 1
};
static struct module_info enet_wminfo= {
    5050, "enet", 0, 65536, 1, 1
};
static struct qinit enet_rinit= {
In addition to a `driver_install` (WSIO_DRV) routine, each HP-UX PCI networking device driver must have a `driver_attach` (WSIO_DRV) routine.

If a networking device driver interfaces with a hardware device, it is required to have a service routine to handle the device interrupts.

The following brief descriptions of the required install, attach, and initialization routines introduce the networking device driver initialization.

The following install routine for the sample driver should be prefixed with the driver name.

```c
int

`driver_install()`

An entry of `driver_install()` is called during the I/O system configuration process. When the `driver_install()` routine is called, it hooks the `driver_attach()` entry to the top of a linked list of attach routines for all of the interface drivers in the system.
Creating Networking Device Drivers
Initializing Networking Device Drivers

#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.

void driver_init(struct isc_table_type *isc_ptr)
int driver_isr(struct isc_table_type *isc_ptr, 
    caddr_t cb_ptr)

cb_ptr Pointer to the driver control block; it is driver developer 
defined and passed as a pointer through the isrlink() 
routine during the driver_attach() or 
driver_init() routines.

The driver_attach() and driver_install() initialization procedures 
are common to all HP-UX device drivers. More details of each step are 
presented in Chapter 5, Writing a Driver.

Calling driver_install()

When the HP-UX system is configured through the config command, a 
table of driver_install() entry points is created from information in 
/stand/system.

When driver_install() is called by the I/O system configuration 
process through the driver_install() entry point configured in the 
system, the driver_install() routine places the driver_attach() 
entry in a table of drivers to be called at configuration time. The 
driver_install() routine calls the wsio_install_driver() routine to 
register the driver with the I/O subsystem and returns any error.

The following is a call to driver_install():

static drv_ops_t enet_drv_ops = {
    NULL,     /* open */
    NULL,     /* close */
    NULL,     /* strategy */
    NULL,     /* dump */
    NULL,     /* psize */
    NULL,     /* reserved */
    NULL,     /* read */
    NULL,     /* write */
}
Creating Networking Device Drivers

Chapter 7 227

Initializing Networking Device Drivers

```c
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,
    DRV_MP | DRV_MODE.tech(),
    -1, /* block major number */
    -1, /* character major number */
    NULL, NULL, NULL, /* structures always set to NULL */


to attach PCI driver to system */
int (*enet_saved_attach)();

int enet_install()
{
    enet_saved_attach = pci_attach;
    /* save the current top entry */
    pci_attach = enet_attach;
    /* link attach entry to list */
    bzero((caddr_t)&enet_drv_ops, sizeof(drv_ops_t));
    msg_printf("enet:install\n");

    static wsio_drv_data_t enet_data = {
        "enet",  /* for matching probes with drivers */
        T_INTERFACE, /* type of hardware, dev or IF */
        DRV_CONVERGED, /* driver flag */
        NULL,  /* minor number build routine */
        NULL,  /* minor number decode routine */

    static wsio_drv_info_t enet_wsio_info = {
        &enet_drv_info, * driver info */
        &enet_drv_ops, /* driver ops */
        &enet_data, /* driver data */
        WSIO_DRV_CURRENT_VERSION
    };```
if(!(rv=wsio_install_driver(&enet_drv_info, 
    &enet_drv_ops)))
    if(rv=str_install(&enet_str_info)) {
        wsio_uninstall_driver(&enet_drv_info);
        msg_printf("enet:install failed\n");
    }
    return rv;
}

Calling driver_attach()

Use the driver_attach() routine to determine whether the product ID passed in matches the driver_attach device and vendor ID. If the IDs do not match, the driver_attach() routine calls the next attach routine in the chain by calling the driver_saved_attach() routine.

NOTE

The driver_attach() routine may be called many times before a match is found. For the device in the first slot, the associated driver_attach() routine is called by the number of devices in the PCI backplane. For the device in the last slot of the PCI backplane, the associated driver_attach() routine is called only once.

When the driver_attach() routine recognizes the device ID it allocates and initializes its driver control blocks and PCI I/O registers. The driver_attach() routine also sets up a driver initialization routine and calls the isc_claim() to claim the device. The following is a sample driver_attach() routine:

```
struct gfsw enet_gfsw;
...
int
#ifdef __LP64__
enet_attach( uint32_t id, struct isc_table_type *isc)
#else
enet_attach( PCI_ID id, struct isc_table_type *isc)
#endif
{
    msg_printf("enet attach id = %x\n",id);
    #ifndef __LP64__
```
/* Support for PCI only */
if (!(id.vendor_id==DEV_VENDORID &&
     id.device_id==DEV_DeviceID)) {
    return enet_saved_pci_attach(id, isc);
}
#else
if (!(id == DEV_ID)) {
    return enet_saved_pci_attach(id, isc);
}
#endif
isc->gfsw = &enet_gfsw;
CONNECT_INIT_ROUTINE(isc,enet_init);
isc->gfsw->diag = (int (*) () )NULL;
#ifdef __LP64__
    isc->if_id = (int)(id & 0x0000ffffU);
#else
    isc->if_id = (int)id.device_id;
#endif
isc_claim(isc, &enet_wsio_drv_info);
return enet_saved_pciAttach(id, isc);
}

HP-UX calls a driver_init() routine to begin driver initialization. It allocates
the driver control block and driver data structures, sets PCI configuration
information, links the driver ISR to the PCI interrupt, and initializes and resets
the controller hardware. The following is the skeleton initialization function
showing PCI configuration and linking of the driver ISR:

int
enet_init(struct isc_table_type *isc)
{
    enet_ift_t   *enet_iftp;
    size_t       size;
    u_long       phys_base;
    ...
    ubit32 base_addrp, id, revid, latency_timer, int_reg;
    ubit32 sub_id, ssid, cfda, csr6;
    BUS_TRANS_DESC desc;
    ubit32 error;
    ...
    ...
    /*
* Allocate driver control block - enet_iftp
*/
...
/

* Obtain memory for Transmit and Receive Descriptor
* Rings and any additional driver data structures */
...
/
* Get/Set PCI configuration */
pci_read_cfg_uint32_isc(isc,SSID,&ssid);
enet_iftp->sub_id = (ubit16)(ssid >> 16);  
enet_iftp->sub_vendor_id = (ubit16)(ssid & 0x0000ffff);

/* Read the Configuration ID information */
pci_read_cfg_uint32_isc(isc,CFID,&id);

/* Read the Configuration Revision information */
pci_read_cfg_uint32_isc(isc,CFRV,&revid);

/* Read the Configuration Interrupt information */
pci_read_cfg_uint32_isc(isc,CFIT,&int_reg);

/* Read the Configuration Driver Area information */
pci_read_cfg_uint32_isc(isc,CFDA,&cfda);
cfda = 0;
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
Creating Networking Device Drivers

Chapter 7

Initializing Networking Device Drivers

/*

* Perform general enet_ift initialization
*/

...

/* Setup hwift structure */

...

...

/* Attach hwift to global list */
hw_ift_attach(&enet_iftp->lanclift.hwift);

...

...

/* size: initialized to the size of enet_iftp->tdr
 (transmit descriptor ring) */

/* Allocate the DMA handle for Tx-descriptor ring */
enet_iftp->tdr_DMA_handle = wsio_allocate_dma_handle(isc);

/* Allocate shared memory for Tx-descriptor ring */
if( wsio_allocate_shared_mem(isc,
enet_iftp->tdr_DMA_handle,size,
    (caddr_t *)&enet_iftp->tdr, 0) != WSIO_MAP_OK) {
    msg_printf("enet - TDR allocation failed...\n");
    return -1;
}

...

}

If initialization is successful, the driver_init() routine proceeds with the following steps:
 Initializes the MIB structure and the `hw_dlpi` and `hw_ift` structures (see the preceding sections “hw_ift_t Structure Description and Initialization” on page 216 and “hw_dlpi Structure Description and Initialization” on page 221 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 password `hw_ift` structure.
```
Protocol Binding and Demultiplexing

This is the mechanism a networking driver uses to associate (bind) an upper layer protocol to a device. The binding ensures the driver correctly demultiplexes and delivers inbound packets to the corresponding upper layer protocol, based on the upper layer protocol's bind request.

To correctly demultiplex inbound packets, a networking driver must:

- Obtain protocol specific information during protocol binding.
- Obtain packet specific information.
- Process packets and information by the upper level protocols.

The following table summarizes the information a networking driver requires to demultiplex inbound packets for corresponding upper layer protocols. More detailed information is provided in the section “DLPI Interface to Upper Layers” on page 239.

<table>
<thead>
<tr>
<th>Packet Type</th>
<th>Protocol Kind</th>
<th>Protocol Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ethernet type</td>
<td>LAN_TYPE</td>
<td>TYPE value</td>
</tr>
<tr>
<td>IEEE 802.2 LLC type</td>
<td>LAN_SAP</td>
<td>SAP value</td>
</tr>
<tr>
<td>SNAP type</td>
<td>LAN_SNAP</td>
<td>OID + extended SNAP info</td>
</tr>
</tbody>
</table>

Protocol kind is the type of protocol to bind. Interpretation of the protocol value field depends on the protocol kind value. “Protocol Kinds and Values” in Appendix A 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_sap` field of the `DL_BIND_REQ` to pass the packet to the right Stream that is logged (see “DLPI Interface to Upper Layers” on page 239 for details).

**Protocol Binding and Unbinding**

Each upper layer protocol issues a bind request to the networking driver to affect binding. The driver is responsible for keeping track of all upper layer protocols currently bound to it. The networking driver also must have a way to unbind a protocol upon request.

**Protocol Demultiplexing**

One of the main functions of the device driver's interrupt service routine is to dispatch inbound packets to the appropriate upper layer protocol. To achieve that, the interrupt service routine in the driver must:

1. Distinguish packet protocol format and type:
   - Ethernet
   - IEEE 802.2 Link Level Control (LLC) (non-SNAP)
   - SNAP (IEEE 802.2 LLC extended)
2. Locate the proper inbound packet service routine or queue for each valid incoming packet.
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 “Protocol Kinds and Values” in Appendix A).

- If the value of the TYPE field is less than 0x600, the packet could be an IEEE 802.2 LLC type packet, SNAP or non-SNAP type.

To determine whether the packet is a SNAP type IEEE 802.2 LLC packet:

- The packet is considered to be a SNAP packet (defined in IEEE 802.1a) if both the DSAP and the SSAP values are 0xAA. The protocol kind of the packet is `LAN_SNAP`, the protocol value is 0xAA, and the protocol value extended is the five-byte SNAP protocol data specified in the SNAP header (see “Protocol Kinds and Values”, in Appendix A).

- 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”, in Appendix A).

The relationships of protocol kind, protocol value, and protocol processing for different types of packets are shown in Table on page 233.

After the device driver has found the protocol kind and value in an inbound packet, the driver locates the protocol input queue that corresponds with the bind request previously received from an upper layer protocol. This queue information is stored by the driver during binding.

If the upper layer requires header stripping, the device driver strips off the link level control (LLC) header before passing the inbound packet to the upstream queue.
Inbound Promiscuous and Outbound Promiscuous

For inbound promiscuous, a promiscuous stream receives the packets destined for other streams (protocols) and (depending on the promiscuous level enabled) other NICs.

For outbound promiscuous, the stream traces all packets on the interface (depending on the promiscuous level enabled).

Table 7-2 explains each promiscuous mode.

<table>
<thead>
<tr>
<th>Table 7-2</th>
<th>Promiscuous Mode Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PROMISC_PHY</strong></td>
<td><strong>PROMISC_MULTI</strong></td>
</tr>
<tr>
<td><strong>Unbound promiscuous stream monitors outbound traffic</strong></td>
<td>The stream gets all outbound packets transmitted on the interface. (broadcast, multicast, self addressed and non self addressed unicast packets)</td>
</tr>
<tr>
<td><strong>Unbound promiscuous stream monitors inbound traffic</strong></td>
<td>The stream gets all packets on the wire regardless of SAP or address.</td>
</tr>
</tbody>
</table>
Creating Networking Device Drivers

Inbound Promiscuous and Outbound Promiscuous

Table 7-2 Promiscuous Mode Matrix (Continued)

<table>
<thead>
<tr>
<th></th>
<th>PROMISC_PHY</th>
<th>PROMISC_MULTI</th>
<th>PROMISC_SAP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bound</strong></td>
<td>The stream gets all outbound packets that match the SAP protocols that the user has bound to on the promiscuous stream.</td>
<td>The stream gets all outbound multicast, broadcast packets that match the SAP protocol the user has bound to on the promiscuous stream. No unicast will be seen.</td>
<td>This primitive has no effect on the interface.</td>
</tr>
<tr>
<td>promiscuous stream</td>
<td><strong>outbound</strong> traffic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bound</td>
<td>The promiscuous stream gets all packets on the wire that match the SAP protocols that the user has bound to on the promiscuous stream.</td>
<td>The promiscuous stream gets all multicast, broadcast, and unicast packets that match the SAP protocol the user has bound to on the promiscuous stream.</td>
<td>This primitive has no effect on the interface.</td>
</tr>
<tr>
<td>promiscuous stream</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bound</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>in bound</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Message Block and Queue Functions and Macros

The message block and queue functions and macros are defined by STREAMS/UX. Refer to the STREAMS/UX for the HP 9000 Reference Manual for further information.

Starting with HP-UX version 11i, the header in the message block data structure mblk_t is not cacheline aligned. The area in an mblk to store data follows the header. Since the header is not cacheline aligned, part of the header shares a cacheline with the data area. If a driver purges the cache corresponding to the data area in order to read DMA data, it may corrupt the message block header since the data area and the header share the same cacheline. Therefore, drivers are required to take precautions to avoid the problem. One solution is to verify the data area and the header are in different cachelines.

The list of commonly used message block functions.

- allocb() Allocate a message block
- freemsg() Free a message block
- pullupmsg() Concatenate and align the data stored in complex message
- adjmsg() Adjust the length of the message
- dupmsg() Duplicate a simple or complex message

The following is the list of queue functions commonly used in a STREAMS driver.

- putq() Queue message to be processed by queue service procedure
- putnext() Call queue’s “put” procedure
- canput() Test whether queue can receive message
- qreply() Send the message back upstream
- OTHERQ() Other queue in the queue pair

- streams_put(), streams_put_release() Allow non-STREAMS/UX (e.g driver ICS) to “put” in a queue
**DLPI Interface to Upper Layers**

The Data Link Provider Interface (DLPI) specifies a STREAMS based kernel implementation of the ISO Data Link Service Definition (ISO 8886) and Logical Link Control (ISO 8802/2 LLC). DLPI allows a data link service user to access and use a variety of conforming data link services without special knowledge of the provider's protocol. The interface specifies access to data link service providers and does not define a specific protocol implementation.

Starting with HP-UX version 11.0, transports (e.g. TCP/IP, UDP, OSI) are now STREAMS modules. Third parties are expected to develop a STREAMS DLPI driver conforming to DLPI version 2.0 to support their network interface drivers and can not depend on the HP DLPI implementation. This section provides information about how third party drivers can integrate into a STREAMS/UX framework in HP-UX.

Two styles of DLPI provider are defined by the DLPI document, distinguished by the way they enable a DLPI user to choose a particular physical point of attachment (PPA). The Style 1 provider assigns a PPA based on the major/minor device the DLPI user opened. The Style 2 provider requires a DLPI user to explicitly identify the desired PPA by using a special attach service primitive. This document illustrates the development of a Style 2 DLPI driver.

**Device 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 device file `enet` (sample STREAMS DLPI driver). The device files created for the STREAMS DLPI driver are also shown.

```bash
# lsdev

...............................................
.............................................
239          -1         enet            lan

# ls /dev/enet*
```

```bash
# ls /dev/enet*
```
Creating Networking Device Drivers
DLPI Interface to Upper Layers

```
crw-rw-rw-   1 rootsys    72 0x0000f0 Apr 12 18:46 /dev/enet
```

`lanscan` (1M) lists all the LAN interfaces in the system from the list of `hw_if_t` structures (every network interface driver should perform `hw_if_attach()` during initialization). This list identifies the interface name and PPA numbers. Refer to “Initializing Networking Device Drivers” on page 224, for details of `hw_if_attach()`.

The following output from `lanscan` illustrates the interface name and PPA numbers for the sample WSIO network driver. The sample driver has “attached” to LAN interfaces at hardware paths 8/0/1/0 and 8/0/2/0.

**Table 7-3 lanscan Output**

<table>
<thead>
<tr>
<th>H/W Path</th>
<th>Station Address</th>
<th>Card Im. #</th>
<th>H/W State</th>
<th>Net I/F, Name PPA</th>
<th>N M ID</th>
<th>MAC Type</th>
<th>HP-DL PT Sprrt</th>
<th>DLPI Mjr#</th>
</tr>
</thead>
<tbody>
<tr>
<td>8/16/6</td>
<td>0x0060B07EDBF0</td>
<td>0</td>
<td>UP</td>
<td>lan0 snap0</td>
<td>1</td>
<td>ETHER</td>
<td>Yes</td>
<td>119</td>
</tr>
<tr>
<td>8/0/1/0</td>
<td>0x0060B07A221E</td>
<td>1</td>
<td>UP</td>
<td>enet1</td>
<td>2</td>
<td>ETHER</td>
<td>No</td>
<td>*</td>
</tr>
<tr>
<td>8/0/2/0</td>
<td>0x0060B0B2D850</td>
<td>2</td>
<td>UP</td>
<td>enet2</td>
<td>3</td>
<td>ETHER</td>
<td>No</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 set up as listed in the steps below.

1. `ifconfig` opens device file `/dev/enet` and senses PPA configured is 1
2. `ifconfig` issues an `ioctl` to push IP module to top of `enet` driver
3. `ifconfig` issues another `ioctl` to issue attach and bind requests for PPA 1
4. `ifconfig` opens device file `/dev/enet` and issues `ioctl` to push ARP to top of `enet` driver
5. `ifconfig` again performs step 3 for ARP/enet stream
6. `ifconfig` opens `/dev/ip` and uses it as dummy multiplexer; IP/enet and ARP/enet streams are linked under dummy multiplexer.
STREAMS DLPI and Network Driver Overview

The DLPI Sequence in Figure 7-4 on page 243 shows the basic structure of STREAMS DLPI driver implementation in HP-UX. There are two main data structures, enet_if_t and enet_dlpi_data_t. These two data structures establish a linkage between the DLPI specific portion and the network interface portion of the driver functionality. This is only an example implementation and is not exported by HP-UX. Third party developers may define their own interface to address their design needs. Initializing hw_if_t structure was discussed in “Initializing Networking Device Drivers” on page 224.
Figure 7-4  STREAMS DLPI Network Driver Sequence

- putmsg()
- driver_open()
- driver_install()
- driver_close()
- driver_attach()
- driver_init()
- driver_wput()
- driver_wsrv()
- _control() (INFO, ATTACH, BIND, PPA_REQ)
- _unitdata_out()
- _fast_in()
- _fast_out()
- _intr()
- _unitdata_out()
- _proc_ioctl()
- _build_hdr()
- driver_isr()

Application Layer
DLS User
getmsg()

User Space
Kernel Space
Network Protocol Layer
STREAMS Head
/dev/enet and /dev/enetX

Data Link Layer

PCI BUS

Chapter 7 243
The general STREAMS/DLPI buffer/message processing is done in the upper part of the STREAMS DLPI network driver. The lower part of the driver implements device initialization, input, output, and control functions. This section provides an overview of the synchronization of the upper and lower parts of the driver.

**Device Driver/DLPI Driver Synchronization**

For a non-STREAMS character I/O mechanism, synchronization between device driver and device can be accomplished by having the device driver sleep with the `sleep()` kernel call on a unique number, typically an object address, while waiting for the request to complete.

Upon receipt the request completion information from the device, the device driver resumes the process with the `wakeup()` kernel call. For STREAMS, however, this kind of sleep-wakeup synchronization mechanism is not permitted because STREAMS may run on either the ICS or the STREAMS scheduler context stack. Synchronization between the DLPI part of the driver and the network interface part is not defined in the DLPI 2.0 documentation.

The sample DLPI driver has an `enet_dlpi_wakeup()` routine to support the necessary synchronization between DLPI and network interface parts of the driver. This `enet_dlpi_wakeup()` routine simulates the STREAMS environment `wakeup()` kernel call.

```c
void enet_dlpi_wakeup(caddr_t addr_ptr)
addr_ptr Address of an object to wakeup. It should correspond to
the negative value returned by the
enet_dlpi_process_lock() routine.
```

The driver implements a routine `enet_dlpi_process_ioctl()` to process ioctl requests. Certain actions are required of the network device driver when device control requests passed through the `enet_dlpi_process_ioctl()` routine return a negative value.

The following rules summarize actions each networking device driver must take in dealing with such DLPI ioctl requests:
1. The control request does one of the following:
   - If the control request completes immediately with no error, the `enet_dlpi_process_ioctl()` routine immediately returns zero to DLPI.
   - If the control request completes immediately with an error, the error is returned as a positive value (from `errno.h`).
   - If the control request cannot complete immediately (that is, the driver must make a request to the hardware), the device driver must hold the `hwift_lock` and return a globally unique negative value to DLPI.

2. Some time later an interrupt or timeout occurs, and the device driver interrupt service routine determines if the interrupt is for a previously blocked and waiting request.

3. The device driver completes the previous `enet_dlpi_process_ioctl()` by placing the results in the appropriate location for that ioctl.

4. The device driver calls the `enet_dlpi_wakeup()` routine with the address of the sleep object that the `enet_dlpi_process_ioctl()` routine previously returned to DLPI.

**STREAMS Synchronization**

HP-UX STREAMS supports MP scalable drivers and modules. STREAMS/UX provides five levels of parallelism called queue, queue pair, module, elsewhere, and global. The queue synchronization level provides the most concurrency. Refer to the *STREAMS/UX for HP 9000 Reference Manual* for detailed information. The amount of parallelism for modules and drivers can be configured by specifying the synchronization level in `streams_info_t` during `str_install()`. The sample DLPI STREAMS driver uses queue synchronization level.
Entering STREAMS from ICS

When the driver is in interrupt context, it is not in STREAMS context. To enter the STREAMS framework correctly from non-STREAMS/UX code, the STREAMS/UX provides streams_put utilities. The driver ICS function can call streams_put() by passing it a function and a queue. STREAMS/UX runs the function as if it were the queue's “put” routine. The function passed in the call can safely manipulate the queue and access the same data structures as the queue's “put” routine. The streams_put_release() routine executes the streams_put functionality on a specified processor. Refer to the STREAMS/UX for HP 9000 Reference Manual for further information.

Driver Support for DLPI

This section discusses the upper portion of the STREAMS DLPI networking driver which buffers STREAMS messages, handles DLPI primitives, and passes data to the network interface part of the driver. This section's objective is to present the code flow of the sample driver enet as background to the sample driver code. Refer to the sample driver code for details. The following topics are discussed:

- DLPI driver data structures
- Open and close routines
- Control functions that describe processing of DLPI primitives such as attach/detach, bind/unbind, enable/disable multicast, enable/disable, and promiscuous
- The main I/O path
- DLPI primitives supported in the sample driver

Major Data Structures

NOTE

These data structures are part of the sample driver. They do not constitute any interface defined by HP-UX.

enet_dli_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_if_t*enetiftp;
    cred_t*cred;
    queue_t*queue_ptr;
    dev_tenet_dev;
    uint32_t dlsap_addr_length;
    uint8_t dlsap_addr[MAX_DLSAP_LEN];
    uint16_t service_mode;
    int curr_state;
    uint32_t xidtest_flag;
    int mac_type; /
    int mac_mtu;
    dlsap_t *dlsap_ptr;
    uint8_t ssap;
    uint16_t sxsap;
    enet_mcast_list_t*enet_mcast_list;
    int promiscuous_flg;
    int promisc_filter;
    uint32_t noloopback_flg;
    uint32_t no_src_routing;
    uint32_t arp_stream;
    uint32_t ip_stream;
    int fast_path;
    int fast_path_pkt_type;
    int fast_path_llc_length;
    int pre_state;
} enet_dlpi_data_t;

The following table explains the fields.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>enetiftp</td>
<td>The interface that is associated with this open stream</td>
</tr>
<tr>
<td>cred</td>
<td>Credential structure of the user who opened this stream</td>
</tr>
<tr>
<td>queue_ptr</td>
<td>Queue pointer to the read queue of the stream</td>
</tr>
<tr>
<td>enet_dev</td>
<td>enet device number</td>
</tr>
<tr>
<td>dlsap_addr_length</td>
<td>Length of DLSAP address</td>
</tr>
</tbody>
</table>
### Table 7-4  \texttt{enet_dlp}_i\_\texttt{data}_t Data Fields (Continued)

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{dlsap_addr[]}</td>
<td>MAC addr + SAP</td>
</tr>
<tr>
<td>\texttt{service_mode}</td>
<td>Only DL_CLDLS supported in the sample driver</td>
</tr>
<tr>
<td>\texttt{curr_state}</td>
<td>DLPI state</td>
</tr>
<tr>
<td>\texttt{xidtest_flag}</td>
<td>\texttt{dl_xidtest_flag 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>\texttt{mac_type}</td>
<td>Interface MAC type</td>
</tr>
<tr>
<td>\texttt{mac_mtu}</td>
<td>Interface MTU</td>
</tr>
<tr>
<td>\texttt{dlsap_ptr}</td>
<td>\texttt{dlsap_t structure list of logged SAPs}</td>
</tr>
<tr>
<td>\texttt{ssap}</td>
<td>First SAP logged on stream</td>
</tr>
<tr>
<td>\texttt{sxsap}</td>
<td>First extended SAP logged on stream</td>
</tr>
<tr>
<td>\texttt{enet_mcast_list}</td>
<td>List of multicast addresses on this stream</td>
</tr>
<tr>
<td>\texttt{promiscuous_flag}</td>
<td>Set to the promiscuous level specified in the DL_PR_\texttt{MISCON_REQ} primitive</td>
</tr>
<tr>
<td>\texttt{promisc_filter}</td>
<td>Set to one (1) if the stream has been bound with any SAP</td>
</tr>
<tr>
<td>\texttt{nolopback_flag}</td>
<td>Set when the application wants to handle loopback. This flag is set when DLPI_SET_NOLOOPBACK ioctl is issued. DLPI turns on the MSGNOLOOP flag in mblk message on every outbound message so driver won’t loop back the packet</td>
</tr>
<tr>
<td>\texttt{no_src_routing}</td>
<td>Set when DLPI_NO_SRC_ROUTING is issued</td>
</tr>
<tr>
<td>\texttt{arp_stream}</td>
<td>Set if this is ARP stream</td>
</tr>
<tr>
<td>\texttt{ip_stream}</td>
<td>Set if this is IP stream</td>
</tr>
</tbody>
</table>
enet_dlpi_data_ptr_arr[] This array holds enet_dlpi_data_t pointers to keep track of the open streams.

Opening and Closing a Driver

The DLPI driver can be accessed via either a regular device or a clone of the original device. The major number of the device file for a cloneable driver must be the clone driver’s major number, 72. (Refer to STREAMS/UX for HP 9000 Reference Manual for more details of clone driver). The minor number is set to the real major number of the device. The clone open is useful because the application does not need to keep track of which minor number is available and does not need to deal with multiple device files.

As can be seen from the following example, /dev/enet is a clone device file of the enet driver.

```
# ll /dev/enet*
   crw-rw-rw-   1 root       sys 72 0x0000ef Apr 12 18:46 /dev/enet
```

The actual major number of the enet driver is 239.

```
# lsdev
           ..............
           ..............
           239   -1 enet lan
```
However, a clone device file for the `enet` driver is created as follows:

```
# mknod /dev/enet c 72 239
```

When a clone device is opened, the clone driver invokes the DLPI driver’s open routine with the CLONEOPEN flag set. The open function `enet_open()` allocates the `enet_dlpi_data_t` for the stream being opened and initializes it. The minor number of a normal device file open is used as the index into `enet_dlpi_data_ptr_arr[]` to store and access `enet_dlpi_data_t` for the stream. The indexes 1 to 99 are reserved for regular open in the sample driver. For clone opens, an unused minor number starting from 100 is allocated. The `enet_dlpi_data_t` for the stream is stored in the `enet_dlpi_data_ptr_arr` indexed by the new minor number.

**Control Functions**

The function `enet_wput()`, the STREAMS driver’s “put” procedure, calls various control functions to service DLPI `M_PROTO` and `M_PCPROTO` messages with local management primitives (information reporting, attach, bind, and others such as multicast and promiscuous). This function consists of a switch table that calls the service function based on message `dl_primitive`. The following is a list of service functions:
### Table 7-5: Message Service Functions

<table>
<thead>
<tr>
<th>Function Name [prefixed by enet_dlpi]</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>_attach()</td>
<td>The information for PPA to be attached is 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>_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. The enet_log_sap_value() function is called. Once driver bind is successful, dlsap_t is allocated and initialized with protocol type and value of SAP. The enet_dlpi_data_t structure for this stream is updated with these bind details.</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 the driver via enet_dlpi_control(). If the request didn’t complete immediately, this routine sleeps on the address of the sleep object of the dlpi_ioctl().</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, disable it. The clean_str_spu_sw_q() routine is called to clean up any requests in the STREAMS/UX. Finally, update the state in enet_dlpi_data to DL_UNATTACHED.</td>
</tr>
</tbody>
</table>
Table 7-5 Message Service Functions (Continued)

<table>
<thead>
<tr>
<th>Function Name [prefixed by enet_dlpi]</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>_get_mib_req()</td>
<td>Services MC_GET_MIB_REQ (sys/mci.h). The driver ioctl DL_GET_STATISTICS is issued to get current MIB statistics.</td>
</tr>
<tr>
<td>_get_mibstats()</td>
<td>Calls enet_hw_req() function to get the standard MIB statistics from the driver structures.</td>
</tr>
<tr>
<td>_getphyaddr()</td>
<td>The enet_hw_req() function is called, which selects the permanent ROM physical address of the network interface, to service DL_PHYS_ADDR_REQ.</td>
</tr>
<tr>
<td>_info()</td>
<td>A service function for DL_INFO_REQ. The information is returned upstream in structure dl_info_ack_t. If the PPA is not attached yet, 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 this function being called. The hw_ifp_t list is searched for this PPA and the information from hw_ifp_t is returned.</td>
</tr>
<tr>
<td>_set_mib_req()</td>
<td>This function services MC_SET_MIB_REQ. The driver ioctl DL_HP_RESET_STATS is issued to reset the MIB statistics.</td>
</tr>
<tr>
<td>_status()</td>
<td>This function sends the hw_ifp-&gt;hdw_state upstream in response to the DL_HP_HW_STATUS_REQ request.</td>
</tr>
</tbody>
</table>
STREAMS DLPI and Network Driver Overview

**Table 7-5** Message Service Functions (Continued)

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>_subs_bind()</td>
<td>When DL_SUBS_BIND_REQ is received, this function is called. If the dl_subs_bind_class is DL_PEER_BIND, a new dlsap_t is allocated and initialized with protocol type and value of SAP. With DL_HEIRARCHICAL_BIND the dlsap_addr information in enet_dlpi_data_t is updated with bind details.</td>
</tr>
<tr>
<td>_subs_unbind()</td>
<td>For each dlsap_t bound, compare the unbind request SAP. If there is a match, routine enet_unlog_sap_value() is called.</td>
</tr>
<tr>
<td>_unbind()</td>
<td>The function enet_unlog_sap_value() is called. dlsap_t is deallocated and the bind information in enet_dlpi_data_t is set to the default value.</td>
</tr>
<tr>
<td>_xidtest_out()</td>
<td>This function services DL_TEST_REQ, DL_TEST_RES, DL_XID_REQ and DL_XID_RES. It builds the test/xid packet and sends it to the driver using dlpi_output().</td>
</tr>
</tbody>
</table>

**IOCTL Processing**

STREAMS/UX provides the capability for user processes to perform control functions by using ioctl calls on device drivers in a stream. These commands cause the stream head to create an M_IOCTL message that includes the ioctl arguments and to send the message downstream to be received and processed by a device driver. The streams “put” function calls enet_dlpi_process_ioctl() to service M_IOCTL message types. This function consists of a switch block that services various M_IOCTL messages. The IOCTL commands are defined in sys/dlpi_ext.h.

The sample driver implements DLPI_IOC_HDR_INFO, DLPI_IOC_DRIVER_OPTIONS, and DLPI_SET_NOLOOPBACK.
The application sends DLPI an M_IOCTL message with the ioctl command DLPI_IOC_HDR_INFO. The M_IOCTL message block is linked with the M_PROTO message block with the DL_UNITDATA_REQ primitive. The LLC header format is built for the specific interface in a new M_DATA message block and linked to M_PROTO; the whole complex message is sent back to the application.

The ioctl DLPI_IOC_DRIVER_OPTIONS routine is processed by sending hw_ift_t information for the request stream.

Depending on the device capabilities, the driver has to reset the device features which are assumed to be true by default by the transport stack. The features include driver checksum offload (DRIVER_CKO), copy on write (DRIVER_COW), long fat pipe (DRIVER_LFP) and long narrow pipe (DRIVER_LNP). The current version of the DDG does not provide any details on implementing support for the above listed features. So follow the implementation as given in enet_dlpi_process_ioctl() routine in the sample enet driver to inform the transport stack that the driver does not support any of these features.

DLPI_SET_NOLOOPBACK ioctl causes the enet_dlpi_data->nollopback_flg to be set to the value specified in the ioctl parameter.

Transmission of Message Blocks

The message block transmission has two paths in the sample implementation. The regular data path uses the DL_UNITDATA_REQ primitive and the fast path. The regular path is defined in the DLPI standards. The fast path uses DLPI_IOC_HDR_INFO ioctl to set up the path and is an HP extension to the DLPI standard.

Regular Data Path The regular data path message transmission works as follows. The streams “put” function enet_wput() receives the DL_UNITDATA_REQ primitive request from the application to send a message to a destination specified in the unitdata message. The enet_wput() function calls the enet_dlpi_unitdata_out() function to service the request. The enet_dlpi_unitdata_out() function applies sanity checks for the stream's DLPI state and request parameters and builds the LLC header. The LLC header message block is linked with the first M_DATA (with DL_UNITDATA_REQ) and calls the driver's output routine enet_hw_req().
Fast Path  For better performance, fast path is used to transmit and receive data. The DLPI user sends DLPI ioctl DLPI_IOC_HDR_INFO to set up the fast path on the stream. The DLPI builds an LLC header template and sends it back to the user. For an outbound packet, the user prepends the link header to the data, based on the template of the link header, and sends M_DATA messages to DLPI. DLPI passes this packet to the network driver without building the link header. For an inbound packet on the fastpath stream, DLPI strips off the LLC header and passes it to the user without building and prepending the DL_UNITDATA_IND primitive to the data.

Reception of Message Blocks

The message is received by the enet_dlpi_mblk_intr() function that was passed to the driver along with the stream queue pointer. The following sanity checks are applied:

- Drop multicast packets for which there is no enabled multicast address.
- If DL_PROMISC_MULTI is at promiscuous level and stream is in state DL_UNBOUND, discard unicast packets.
- If DL_PROMISC_SAP, discard packets not destined for stream’s network interface.

This function calls enet_dlpi_unitdata_in() or enet_dlpi_fast_in(), based on whether fast path is set or not.

The enet_dlpi_unitdata_in() routine allocates an M_PROTO message block and builds a DL_UNITDATA_IND primitive from the LLC header in the M_DATA message received from the driver. The LLC header is stripped off the M_DATA message, and this block is linked to unitdata message and sent to the application.

The function implemented in enet_dlpi_fast_in() was discussed in “Fast Path” earlier in this section.
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. The processing of most DLPI primitives and IOCTLS involves driver interaction, which is discussed in “Driving the NIC” on page 257.

Table 7-6 DLPI Primitives and IOCTLS

<table>
<thead>
<tr>
<th>DLPI Primitive or IOCTL</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DLPI PRIMITIVES DEFINED BY DLPI 2.0</strong></td>
<td></td>
</tr>
<tr>
<td>DL_ERROR_ACK</td>
<td>Information reporting</td>
</tr>
<tr>
<td>DL_INFO_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_INFO_ACK</td>
<td></td>
</tr>
<tr>
<td>DL_ATTACH_REQ</td>
<td>Attach</td>
</tr>
<tr>
<td>DL_DETACH_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_ERROR_ACK</td>
<td></td>
</tr>
<tr>
<td>DL_OK_ACK</td>
<td></td>
</tr>
<tr>
<td>DL_BIND_REQ</td>
<td>Bind</td>
</tr>
<tr>
<td>DL_BIND_ACK</td>
<td></td>
</tr>
<tr>
<td>DL_ERROR_ACK</td>
<td></td>
</tr>
<tr>
<td>DL_OK_ACK</td>
<td></td>
</tr>
<tr>
<td>DL_SUBS_BIND_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_SUBS_BIND_ACK</td>
<td></td>
</tr>
<tr>
<td>DL_SUBS_UNBIND_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_UNBIND_REQ</td>
<td>Other</td>
</tr>
<tr>
<td>DL_DISABMULTI_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_ENABMULTI_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_ERROR_ACK</td>
<td></td>
</tr>
<tr>
<td>DL_GET_STATISTICS_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_OK_ACK</td>
<td></td>
</tr>
<tr>
<td>DL_PHYS_ADDR_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_PROMISCOFF_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_PROMISCON_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_UNITDATA_IND</td>
<td>DLPI Ver 2.0 Connection less</td>
</tr>
<tr>
<td>DL_UNITDATA_REQ</td>
<td>Data transfer</td>
</tr>
</tbody>
</table>
This section briefly explains the code flow of the lower part of the driver. This portion of the driver handles device interrupts, sends and receives frames, handles control requests from the upper part that require interaction with the device, and so forth. The objective here is to present the code flow of the sample driver `enet` as a background to the sample driver code.
Data Structures

**enet_if_t** This structure holds network interface PCI information, register addresses, transmit and receive buffers and descriptors, driver state, and MIB statistics. This structure also embeds an enlan_ift structure that holds generic LAN information pertaining to this interface. The following shows the structure organization.

```c
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
     *********************************************************/
    enet_tb_t*tbr; /* Transmit buffer Ring */
    enet_td_t*tdr; /* Transmit Descriptor Ring */
    void *tdr_DMA_handle; /* DMA handle for Tx-desc ring */
    ...
    ...

    /* Receive Section - RX SECT
     *********************************************************/
    enet_rd_t*rdr; /* Receive Descriptor Ring */
    enet_rb_t*rbr; /* Receive buffer Ring */
    void *rdr_DMA_handle; /* DMA handle for Rx-desc ring */
```
...  
...  
/*************************************************************/
* Full Duplex, speed and Transmit Threshold setting - SETTINGS  
*********************************************************************/
...  
...  
/*************************************************************/
* Local Driver Receive Stats - STATS  
*********************************************************************/
rcv_stats_t rstats;/* Receive Statistics*/
/*************************************************************/
* Local Driver Transmit Stats - STATS  
*********************************************************************/
trx_stats_ttstats;/* Transmit Statistics*/
/*************************************************************/
* Mib Specific Section  
*********************************************************************/
mib_xEntrymib_xstats;
mib_Dot3StatsEntrydot3_ext_stats;
mib_Dot3CollEntrydot3_ext_coll;
/*************************************************************/
* Misc  
*********************************************************************/
...
...
/******************************/
* lock_t*enet_r_lock;  
******************************/
enet_ift_t * next; /* pointer to the next interface  
* structure */
wsio_intr_object_t enet_wsio_intr; /* Interrupt object */
/******************************/
} enet_ift_t;

Table 7-7  
enet_ift Data Fields

<table>
<thead>
<tr>
<th>Field Name/Generic Description</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>lancift</td>
<td>Contains generic LAN information</td>
</tr>
<tr>
<td>PCI INFO</td>
<td>Has PCI configuration information</td>
</tr>
</tbody>
</table>
### Table 7-7 enet_if Field Name/Generic Description

<table>
<thead>
<tr>
<th>Field Name/Generic Description</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEV REG</td>
<td>Fields have Control and Status Register addresses</td>
</tr>
<tr>
<td>DEV SPEC</td>
<td>Device specific information, such as ISC structure, serial ROM data, driver states, and cable state</td>
</tr>
<tr>
<td>TX SECT: tbr, tdr</td>
<td>This set of fields contains transmit buffers, transmit descriptors, and counters</td>
</tr>
<tr>
<td>RX SECT: rbr, rdr</td>
<td>This set of fields contains receive buffers, receive descriptors, and counters</td>
</tr>
<tr>
<td>SETTINGS</td>
<td>Full duplex, link speed, selected connection type, and transmit threshold settings</td>
</tr>
<tr>
<td>STATS</td>
<td>Driver local receiver and transmitter statistics</td>
</tr>
<tr>
<td>mib_xstats</td>
<td>MIB objects (RFC 1066/1156) and additional counters</td>
</tr>
<tr>
<td>dot3_ext_stats</td>
<td>Extended MIB statistics</td>
</tr>
<tr>
<td>dot3_ext_coll</td>
<td>Extended MIB collisions</td>
</tr>
<tr>
<td>Interrupt object</td>
<td>Contains driver interrupt information</td>
</tr>
<tr>
<td>enet_r_lock</td>
<td>Lock for accessing enet_if</td>
</tr>
</tbody>
</table>
enlan_ift  This structure holds generic LAN information for the network interface. It is shown below; the table explains the fields.

typedef struct{
  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;
  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 7-8  enlan_ift Data Fields

<table>
<thead>
<tr>
<th>Field Name/Generic Description</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>hwift</td>
<td>Generic Hardware information</td>
</tr>
<tr>
<td>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>
</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];
    caddr_t ift_ptr;
    queue_t *q_ptr;
    int flags;
};
```

Table 7-8 enlan_ift Data Fields (Continued)

<table>
<thead>
<tr>
<th>Field Name/Generic Description</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>broadcast_filter, multicast_filter, promiscuous_filter</td>
<td>Read packet filters</td>
</tr>
<tr>
<td>mcast, mcast_ref_cnt</td>
<td>Multicast addresses and their reference count</td>
</tr>
<tr>
<td>mib_xstats_ptr</td>
<td>MIB object</td>
</tr>
<tr>
<td>enlanc_lock</td>
<td>Lock to access enlanc_ift</td>
</tr>
</tbody>
</table>

**logged_info, logged_link** For each bind, the network driver keeps track of the bound SAPs and relevant information about the bind. The following structures are used to maintain this information.

Table 7-9 Bound SAP Data Fields

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>protocol_val</td>
<td>SAP, Type, or Canonical value</td>
</tr>
<tr>
<td>ift_ptr</td>
<td>Driver control block (enet_ift)</td>
</tr>
<tr>
<td>q_ptr</td>
<td>Queue pointer of the stream 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.

```c
struct logged_link{
    struct logged_link *next;
    struct logged_info log;
};
```

**Control Functions**

The function `enet_dlpi_control()` communicates the device dependent DLPI primitives to the network interface part of the driver for further processing. Essentially, the DLPI function calls the respective functions passing per instance, driver control structure `enet_if`, the `ioctl` command, and the message block with request data.

The following subsections summarize the driver control commands and the processing by the network driver.

**DL_HP_ENABMULTI** `ext_mcast_list[]` is a global array, where each element of the array is an `hw_mcast_entry_t` structure corresponding to a particular interface.

```c
typedef struct {
    mcast_list_t *hw_mcast;
    int mc_threshold;
} hw_mcast_entry_t;
```

`hw_mcast` points to a linked list of `mcast_list_t` structures which hold multicast addresses enabled on an interface.

```c
typedef struct mcast_list {
    uint8_t addr[6];
    /* Multicast address */
    int ref_cnt;
    /* Number of times the multicast address has been enabled */
    struct mcast_list *next;
    /* pointer to next structure */
} mcast_list_t;
```

`enet_dlpi_control()` calls `enet_media_control()` function to process **DL_HP_ENABMULTI** command.

`enet_media_control()` function checks validity of multicast address and calls macro `ENET_UPDATE_EXT_MCAST` to update the entry for the multicast address.
This macro checks all multicast entries for the interface. If there is already an entry of the requested multicast address, then it just updates ref_cnt and returns.

If the requested multicast address is not there in the list, then enet_media_control() calls ENET_ADD_EXT_MCAST macro.

This macro allocates an mcast_list_t structure, assigns the requested multicast addr to addr[], initializes ref_cnt to 1 and then adds mcast_list_t (multicast entry) to a linked list of multicast entries for the interface.

Then, the macro calls enet_hw_req() to enable the requested multicast address on the device.

**DL_HP_DISABMULTI**  enet_dlpi_control() calls enet_media_control() function to process the DL_HP_DISABMULTI command.

enet_media_control() function calls the ENET_DEL_EXT_MCAST macro.

This macro gets mcast_list_t structure (multicast entry) for the requested multicast address and decrements ref_cnt of the structure. If ref_cnt becomes zero, then mcast_list_t (multicast entry) is removed from the linked list.

Then, ENET_DEL_EXT_MCAST calls enet_hw_req() to remove the multicast address from the device.

**DL_HP_PROMISCON**  enet_promisc_list[] a global array; each array element is a p_entry_t structure corresponding to an interface.

typedef struct {
   int (*func_ptr) __((struct lan_ift *,
                    void *, void *, u_int));
   caddr_t data_ptr;
   /* Function to call for promiscuous packets */
   uint32_t filter_cnt;
   /* ref cnt for SAP-based request to filter */
   uint32_t no_filter_cnt;
   /* ref cnt for requests to receive all pkts */
   uint32_t phys_ref_cnt;
   /* ref cnt to enable phys promisc */
   uint32_t multi_ref_cnt;
   /* ref cnt to enable multi promisc */
   uint32_t sap_ref_cnt;
   /* ref cnt to enable sap promisc */
} p_entry_t;
enet_dlpi_control() calls enet_media_control() function to process the DL_HP_PROMISCON command.
enet_media_control() function updates related fields in the p_entry_t structure and calls enet_hw_req() to enable a specific promiscuous level on the device.

Currently only one stream can be in promiscuous mode per interface. Refer to the ENET driver example source.

**DL_HP_PROMISCOFF** enet_dlpi_control() calls enet_media_control() function to process the DL_HP_PROMISCOFF command.
enet_media_control() updates related fields in the p_entry_t structure and calls enet_hw_req() to disable promiscuous mode on the device. Refer to the ENET driver example source.

**DL_HP_SET_PHYS_ADDR** Driver calls enet_media_control() to enet_hw_req() which in turn calls enet_ctl_req() to change the local address.

**DL_HP_RESET_STATS** The functions called are enlanc_media_control(), enet_hw_req(), enet_ctl_req(), and enet_ext_clearmib() to clear MIB.

**DL_HP_HW_RESET** The following functions are called in order: enlanc_media_control(), enet_hw_req(), enet_ctl_req(), and enet_reset() to perform hardware reset.
Datapath

Outbound Path  The enet driver write path starts with the function enet_dlpi_unitdata_out.

Figure 7-5  Control Flowchart for Outbound Path
enet_dlpi_unitdata_out()  This function calls enet_hw_req() to handle the write request.

enet_hw_req()  All LAN_REQ_WRITE write requests and LAN_REQ_WRITE_L loopback write requests are processed when the driver state is ENET_ONLINE. Otherwise, only loopback write requests are processed and other write requests are discarded.

Non-loopback unicast packets are transmitted in the fast path by calling ENET_TRANSMIT_FRAME. Multicast, broadcast, self addressed frames, frames < 14 bytes, and frames with buffers > ENET_MAX_BUF_PER_FRAME are handled in the slow path by calling enet_slow_hw_req().

Non-write requests are passed on to enet_ctl_req().

enet_slow_hw_req()  Non unicast frames are handled in enet_transmit_complt(). If the number of buffers is > ENET_MAX_BUF_PER_FRAME, an attempt is made to copy all the buffers into one, to use only one transmit descriptor and fewer buffer descriptors. The frame is sent by calling ENET_TRANSMIT_FRAME.

ENET_TRANSMIT_FRAME  A check is made to see if transmit buffers are available to send the frame; if not, the frame is dropped. If transmit descriptors are unavailable the buffer is queued up for later transmission. Pending transmits are handled in the transmit complete interrupt. The enet_transmit_complt() routine is called to process transmit complete interrupts. Otherwise, the transmit descriptors are set up and a transmit poll is issued to the device to send out the frame. The device interrupts after all frames waiting transmission on the transmit descriptor list are transmitted. The enet_transmit_complt() routine is called to handle the interrupt.

enet_transmit_complt()  This routine processes transmit complete interrupts. Call enet_slow_complt() to process non-unicast frames or setup frames. Transmit error handling is done by calling the enet_trans_error() routine. If there are frames queued for transmission, call enet_transmit_pended_frames() to restart transmission.
enet_transmit_pended_frames() While there are frames pending transmission, map the frames, set up the transmit descriptors, and issue a transmit poll to the device to transmit the frames.

Inbound Path

Figure 7-6 Control Flowchart for Inbound Path
The `enet` read path is on the ICS. The `enet_isr()` routine is called when the network interface's PCI interrupt is received and the `enet_receive_frame()` routine is invoked to process received frames.

`enet_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 the receive descriptor. If the driver state is `ENET_ONLINE`, call the `enet_process_packet()` routine to process the frame. Otherwise, call the `enet_process_looper()` routine to process the frame. Replenishing the receive descriptor ring with buffers is done while doing frame receive processing.

`enet_process_packet()` This function determines the frame header is Ethernet or IEEE 802.2 and `enlanc_ether_ics()` or `enlanc_802_2_ics()` is called, accordingly.

`enet_process_looper()` This function processes the loopback packet. The current driver sub-state determines the action taken. The packet buffer is validated but not used, and discarded.

`enet_802_2_ics()` The packet type (802.2 or 802.2 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 the `enet_route_promisc()` routine. The lookup for logged DLSAPs is `enet_sap_lookup()`, and if there is a match, this routine sends the packet to the logged stream (by calling the function registered during the bind). XID and TEST packets are processed in `enet_802_2_test_ctl()`.
enet_ether_ics() Protocol kind (LAN_TYPE, LAN_SNAP or LAN_SAP) and protocol value are extracted from the received packet. If the interface supports promiscuous mode and is set, route the packet to all streams qualified for the set promiscuous level using the enet_route_promisc() routine. The lookup for logged DLSAPs is enet_lookup() and if there is a match, this routine sends the packet to the logged stream by calling the function registered during the bind.

**Inbound Promiscuous Handling** Depending on the promiscuous level set, the device receives the packets not destined to the interface.

For ex, the device receives all the packets on the wire for PROMISC_PHYS and all multicast, broadcast packets if PROMISC_MULTI is set and under normal operation for PROMISC_SAP.

**Bound Promiscuous Stream** The bound promiscuous stream sends packets through the same path as normal packets. For example: enet_ether_ics(), or enet_802_2_ics(), calls enet_sap_lookup() to look for the stream matching the destination SAP of the packet. The packet is then passed to the stream.

**Unbound Promiscuous Stream** enet_ether_ics() and enet_802_2_ics() call enet_route_promisc().

enet_route_promisc() gets the promiscuous stream’s queue pointer from the p_entry_t structure for the interface.

For PROMISC_SAP, enet_route_promisc() passes only those packets to the stream whose destination SAP matches with any SAPs enabled on the interface.

For PROMISC_PHYS and PROMISC_MULTI, enet_route_promisc() passes all the packets to the promiscuous stream because the device already has filtered the packets.

**Outbound Promiscuous Handling** The ENET_TRANSMIT_FRAME macro routes all the packets to enet_ether_ics() and enet_802_2_ics() for outbound promiscuous.

**Bound Promiscuous Stream** The bound promiscuous stream sends packets through the same path as normal packets. For example: enet_ether_ics(), or enet_802_2_ics(), calls enet_sap_lookup() to look for the stream matching the destination SAP of the packet. The packet is then passed to the stream.

**Unbound Promiscuous Stream** enet_ether_ics() and enet_802_2_ics() call enet_route_promisc().

enet_route_promisc() gets the promiscuous stream’s queue pointer from the p_entry_t structure for the interface.

For PROMISC_SAP, enet_route_promisc() passes only those packets to the stream whose source SAP matches with any SAPs enabled on the interface.
For PROMISC_MULTI, only multicast and broadcast packets are passed to the stream.

For PROMISC_PHYS, all packets are passed to the stream.

**Interrupt Service Routine - enet_isr()**

enet_isr() handles the interrupt generated by the NIC. It can also be invoked by the kernel when any other device (which shares the same interrupt resource as the NIC) generates the interrupt.

enet_isr() must check if the interrupt is generated by the NIC before processing the interrupt. If it is not generated by the NIC, then enet_isr() should return zero: The zero value indicates to the kernel that the interrupt is generated by the other device.

enet_isr() can be called even when the NIC is suspended (see Chapter 11, On-Line Addition/Replacement,), due to interrupts generated by other devices which share the same interrupt resource. Therefore, enet_isr() must verify that the NIC is online before accessing any card register (to check if the interrupt is generated by that card). If the NIC is suspended, then enet_isr() must return zero: The zero value indicates to the kernel that the interrupt is generated by the other device.

**Releasing any Pending Timeouts**

Before the driver gets suspended during an OLA/R event or before the driver is unloaded in a DLKM operation, the driver shall be free of any pending callback routines. For more information on OLA/R and DLKM, refer to Chapter 11 and Chapter 12 in the DDG.

ENET driver maintains a list of pending timeout routines. On an OLA/R suspend event or during a DLKM unload, the driver calls untimeout() on all the pending timeout entries in the timeout list.

Following enum is a field in enet_ift structure which saves this information.

```c
enum {
    ENET_SEND_LOOP_PKT_TIMEOUT = 1 << 0,
    ENET_AUTO_NEG_TIMEOUT       = 1 << 1,
    ENET_FORCE_SPEED_DUPLEX_TIMEOUT = 1 << 2,
} timeout_list;
```
Each flag in `enum` refers to a function that can be on the timeout list. The flag for the function which is called through `timeout()` is set until the function passed to timeout is called.
Platform Specifics

Interface drivers are supposed to take care of platform dependencies so one object can run on any HP platform. The CDIO in the I/O subsystem provides a consistent view of HP-UX platforms to drivers by hiding the platform dependencies as much as possible. Some newer platforms, such as the V Class, have dependencies that require special coding.

DMA Mapping

On the transmit side, packets that are passed to the driver from upper layers may cross a page boundary in virtual address space, and a page-crossing buffer may not be contiguous in physical address space. In the 'hints' argument to DMA mapping service wsio_map_dma_buffer()/wsio_map(), if WSIO_DMA_CONTIGUOUS/IO_CONTIGUOUS is specified, then the DMA mapping service tries to map the buffer to a contiguous IOVA range.

On coherent systems, it is possible to map physically non-contiguous buffers to a contiguous IOVA range. But on noncoherent systems, IO devices must directly access physical memory. Thus, it is not possible to map the non-contiguous physical buffer with hints WSIO_DMA_CONTIGUOUS/IO_CONTIGUOUS on such systems.

To use a single driver source for both coherent and noncoherent systems, WSIO_DMA_CONTIGUOUS/IO_CONTIGUOUS hint should not be specified if the driver is expected to be passed with non-contiguous buffers.

For a detailed information on cache coherence issues, refer to Chapter 3, Understanding HP-UX I/O Subsystem Features, in the Driver Development Guide.

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
Creating Networking Device Drivers

Platform Specifics

different slots will not share a DMA channel or steal resources from each other. The I/O card cannot access any non-coherent address space beyond EPIC.

The driver model for EPIC expects that all control structures are small and stored in EPIC shared memory. All application data is assumed to be in buffers in coherent system memory. These buffers are read or written as part of DMA stream. Outbound prefetch is initiated when a buffer is mapped for an I/O card's DMA access (since it is in coherent memory, it can be prefetched).

WSIO mapping calls work the same way on V Class platforms as on other platforms. EPIC CDIO (accessed via WSIO) will not reassign an IOVA range until all mappings within the channel have been released, so one must be careful with long term mappings.

The following points are useful while writing PCI network drivers for V class.

- Allocate transmit and receive descriptor memory in shared memory with the `wsio_allocate_shared_mem()` function.
- Shared memory does not need `wsio_map();` it is already both virtually and physically contiguous.

The following code examples illustrate the use of the function `wsio_allocate_shared_mem()`.

```c
/* This code illustrates the use of shared memory to allocate * a transmitter buffer ring for a V CLASS system network * controller. Refer to the sample driver enet.c for more * details. * Look for #ifdef V_CLASS or if(is_SPP()) statements. */
static int
enet_init (struct isc_table_type *isc) {
    enet_ift_t     *enet_iftp;
    size_t         size;
    u_long         phys_base;

    /* size: initialized to the size of enet_iftp->tdr
     * (transmitter descriptor ring)
     */
```
/* Allocate the DMA handle for TX-descriptor ring */
enet_iftp->tdr_DMA_handle = wsio_allocate_dma_handle(isc)

/* Allocate shared memory for Tx-descriptor ring */
if( wsio_allocate_shared_mem(isc,
enet_iftp->tdr_DMA_handle, size,
    (caddr_t *) & phys_base,
    (caddr_t *) & enet_iftp->tdr,
0) != WSIO_MAP_OK)
    msg_printf("enet - TDR allocation failed...");
    return -1;

...
Network Management Support

Hewlett-Packard’s implementation of MIBs and the access methods to MIB information from HP-UX version 10.00 and previous releases has been monolithic in nature; all MIB support was directly done in kernel. This approach forced Hewlett-Packard to constantly change the kernel to incorporate new MIB instrumentation when new links or drivers, either supplied by Hewlett-Packard or a third party, were added.

Hewlett-Packard moved from a single monolithic agent to a variable number of agents, called subagents. Whenever a new driver is added to a system, a user space subagent specific to this driver is also supplied. This subagent provides the MIB instrumentation needed to access the MIB objects associated with the driver. Figure 7-7, Master Agent/Subagents Relationship, shows the master agent/subagent relationship and partitioning of the subagents. The assumption now is that whoever supplies the new driver will also supply the subagent for that driver.

An SNMP manager only communicates with the master agent, and the master agent sends requests to the appropriate subagent(s). The subagent(s) reply to the master agent, which replies to the SNMP manager.

The new Network Management interface will be user based, contained completely within a user space library (libnm.a) and in general, will have a one-to-one mapping to the calls provided by the old Network Management Interface.
When replacing the `/dev/netman`, the following `ioctl`s 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 the following sample code for ifEntry struct in sys/mib.h.) In the 4.3 Berkeley based networking stack, the ifTable was directly tied to the global ifnet structure list. When a networking interface registered an ifnet structure via if_attach, an ifIndex value was returned which was to be used in the MIB-II ifEntry object managed by that networking interface. The ifTable was known by the TCP/IP protocol stack and all interfaces to retrieve the ifTable and a specific ifEntry was through the TCP/IP protocol stack. With the movement to a STREAMS based TCP/IP protocol stack, the global ifnet structure list no longer exists and, therefore, the global ifTable management no longer exists.

Even though in the STREAMS based networking environment the ifTable is not globally managed, each ifEntry in the ifTable must have a unique ifIndex value so the ifTable can be created. Therefore, the ifIndex values must be globally managed. Along with managing the ifIndex values, the MIB-II ifNumber object must also be managed. The Network Management services described next are for retrieving and returning a unique ifIndex value.

\[
\text{u_int32 get_nmid()}
\]

Allocates a system unique ifIndex value for use in the MIB-II ifEntry object. Any kernel entity that required an entry in the ifTable should use this service for retrieving the value of the ifIndex field.

\[> 0 \text{ indicates the call succeeded and the value returned is the ifIndex value.}\]

\[<=0 \text{ indicates the request failed to allocate an ifIndex value.}\]

Example code in enet driver:

\[
\text{enet_iftp->lancift.hwift.nm_id = get_nmid();}
\]

\[
\text{u_int32 return_nmid}
\]

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.

\[
\text{u_int32 return_nmid (u_int32 ifIndex)}
\]
ifIndex ifIndex value to be returned to the pool of available ifIndex values. <0 Indicates the ifIndex value being returned was not the previously assigned ifIndex value >=0 Indicates the ifIndex was successfully returned to the pool.

In sys/mib.h, mib_ifEntry is defined as:

typedef struct {
    int ifIndex;
    char ifDescr[64];
    int ifType;
    int ifMtu;
    gauge ifSpeed;
    mib_physaddr_t ifPhysAddress;
    int ifAdmin;
    int ifOper;
    TimeTicks ifLastChange;
    counter ifInOctets;
    counter ifInUcastPkts;
    counter ifInNUcastPkts;
    counter ifInDiscards;
    counter ifInErrors;
    counter ifInUnknownProtos;
    counter ifOutOctets;
    counter ifOutNUcastPkts;
    counter ifOutDiscards;
    counter ifOutErrors;
    gauge ifOutQlen;
    int ifSpecific;
} mib_ifEntry;

The device driver's job is to fill out the fields in the struct mib_ifEntry in the appropriate order. Any application can then retrieve information for use by the Network Management Support services interface.
Network Tracing and Logging Support for Troubleshooting

This section describes the use of the HP-UX network tracing and logging facilities. To aid in troubleshooting network problems, support for network troubleshooting must address several trends:

- The complexity of network systems is increasing.
- The number of protocols and standards is large and continues to grow.
- The possible combinations of services and applications created and used on a network is increasing.
- The 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
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 and kernel routines to access the services.

Figure 7-8, Network Tracing, 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
Creating Networking Device Drivers
Network Tracing and Logging Support for Troubleshooting

Figure 7-8  Network Tracing, Logging Elements and Data Flow I

These elements are explained in the following sections of this chapter.
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 285.

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; it starts, stops, and sets the capture criteria for tracing and logging. Specifically, `nettl` creates a port where messages can be stored while being written to the output file. The `nettl` command initializes the ktl driver, also called `netdiag1`, and `nettl` starts up the `nktl_daemon` and `ntl_reader` daemons. See the manpage for more detailed information.
Creating Networking Device Drivers

Network Tracing and Logging Support for Troubleshooting

**netfmt(1M)**

This command formats binary trace and log data into readable ASCII text. Post-filtering of the data is controlled through this command.

1. The `netfmt` command uses subsystem configuration information to identify shared libraries provided by subsystems that contain functions to parse subsystem filters and format subsystem data.

2. The `netfmt` command dynamically loads all shared libraries and finds the functions each time it is executed.

3. The command calls the functions of subsystems for which it has data.

4. The command parses the filter file if it is present. The file is sorted according to the first field, the subsystem name, in the filter file.

5. For each subsystem referenced in the file, the `subsys_N_get_options()` function for that subsystem is called with the filter data. The `subsys_N_get_options()` function is responsible for interpreting and storing the filter data.

6. The `netfmt` command reads the input file. For each record found it calls the corresponding `subsys_N_format()` function to format the record.

The subsystem will not format the record if the values in the record match the values specified in the filters. The subsystem should format the record according to the format options specified; for example, nice, terse, and raw. See the `netfmt` (1M) manpage for more detailed information.

**Using HP-UX Logging and Tracing for Troubleshooting Support**

The following guidelines may help developers to be “user friendly” when designing tracing and logging facilities to solve the troubleshooting problems of their clients:

- Log only what is needed to solve problems.

- Record all information to diagnose the problem in the log.

- Provide a hex dump to the 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 their problems, not the developer's problems.

The following information will help set up tracing and logging to support troubleshooting:

- Assign Subsystem ID
- Classify Trace Data
- Format Trace Data
- Classify Log Data
- What and When to Log

**Assign Subsystem ID**

Each networking product requires its own unique subsystem ID number, which must be assigned by the Hewlett-Packard OpenConnect Team.

To do so, Email a request for a unique subsystem ID for your product to Hewlett-Packard at nettl_support@india.hp.com. In the message identify a suggested interface subsystem name for your product. Check /usr/include/sys/subsys_id.h in your system prior to selecting the name. Do not request names such as lan, lo, ni, X25, and others that are already assigned. You will be assigned this name if it is not already being used.

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**

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 the failure of a networking operation to complete successfully. Tracing follows or records normal events and abnormal events alike and is typically used on events that occur frequently, such as connections opening and closing, or re-transmitted data.

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. Other than the items outlined in the preceding tracing or logging sections, some general guidelines include:

- If an event results or causes the product or system to be unusable by all users, it should be logged as a Disaster class log message.
- If an event affects a single application, it should be logged as an Error class log message.
- If an event 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.
Do not log “Me Too!” messages in Error or Disaster class. These are events which occur in response to an error or disaster event in another place, but aren't themselves a disaster or error. “Me Too!” messages are characterized as providing no additional information to solve the problem at hand.

Do not acquire a new log instance if one is already available for the particular event thread 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; event number, a bounded array, or a string, for example.

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.
```

```c
#include <subsys_id.h>
Contains subsystem information and definitions for log classes and trace kinds.
```

The function calls for kernel subsystems capture trace and log data.

**KTRC_CK()**

This macro is used to trace on an all interface device basis. It allows the calling process to verify if tracing is enabled for the current subsystem. The returned value is one (1) if tracing is enabled. It is defined as:

```
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 285.
- **trace_kind**: Defines trace kind; these are defined in `subsys_id.h` header file and are detailed in “Classify Trace Data” on page 286, as follows:
  - HDR_IN_BIT: Inbound header tracing mask
  - HDR_OUT_BIT: Outbound header tracing mask
  - PDU_IN_BIT: Inbound PDU tracing mask
  - PDU_OUT_BIT: Outbound PDU tracing mask
  - PROCEDURE_TRACE_BIT: Procedure entry/exit trace
  - ERROR_TRACE_BIT: Error tracing mask
  - LOGGING_TRACE_BIT: Log call tracing mask
  - LOOP_BACK_BIT: For loopback
  - PTOP_BIT: For point to point
NOTE

There are some alias or redefine the trace_kind functions in the net_diag.h header file mentioned earlier:

```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 as follows:

```c
if (KTRC_CK(ENET_ID, TR_LINK_INBOUND))
{
    ktrc_write(...);
}
```

**ktrc_write()**

This routine is used to send trace messages to the kernel trace and log facility.

Prefiltering is done at the time of the trace call, and unwanted messages are dropped. This routine always returns a success indicator of zero and is defined as:

```c
ktrc_write ( int subsys_id, int trace_kind, int path_id, 
              int device_id, caddr_t tl_packet, 
              int tl_packet_cnt)
```

**subsys_id**

Unique subsystem ID of the calling subsystem (number assigned by Hewlett-Packard; see the “Assign Subsystem ID” on page 285.

**trace_kind**

Defines the kind of trace. All kinds are defined in the header file subsys_id.h. The following is the defined trace kind values (see “Classify Trace Data” on page 286). They can be OR'ed to produce the combination of trace kinds.

- **HDR_IN_BIT** - Inbound header tracing mask
- **HDR_OUT_BIT** - Outbound header tracing mask
- **PDU_IN_BIT** - Inbound PDU tracing mask
- **PDU_OUT_BIT** - Outbound PDU tracing mask
PROCEDURE_TRACE_BIT
   Procedure entry/exit trace

STATE_TRACE_BIT
   State machine tracing mask

ERROR_TRACE_BIT
   Error tracing mask

LOGGING_TRACE_BIT
   Log call tracing mask

LOOP_BACK_BIT
   For loopback

PSTOP_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.

t1_packet
   Either a pointer to an mbuf chain or a pointer to a set of iovec structures as determined by t1_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.

t1_packet_cnt
   If −1, t1_packet points to an mbuf chain. If greater than zero, this is the number of the iovec structure to which t1_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.
Creating Networking Device Drivers

Network Tracing and Logging Support for Troubleshooting

**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
#include "../h/netdiag1.h"
#include "../h/net_diag.h"
#include "../h/subsys_id.h"
#include "../h/uio.h"

#define MAX_BUF 3 /* any number of vectors are allowed */
#define TRACE 0
#define FALSE 0

int
trace_pdu_out(pdu hdr, pdu hdr_len, pdu data, pdu data_len)
char *pdu hdr;
int pdu hdr_len;
char *pdu data;
int pdu data_len;
{
int kind;
int device_id;
int path_id;
short subsys_id;
struct iovec tl_buf[MAX_BUF];
int tl_buf_cnt;

/*
 * Set up variables for KTRC_CHECK()
 */
subsys_id = MY_SUBSYS_ID_NUMBER; /* assigned by HP */
knd = 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.
 */
```

Chapter 7

293
Creating Networking Device Drivers

Network Tracing and Logging Support for Troubleshooting

41
42 \texttt{tl\_buf[0].bufptr = pdu\_hdr;} \\
43 \texttt{tl\_buf[0].buflen = pdu\_hdr\_len} \\
44 \texttt{tl\_buf[1].bufptr = pdu\_data;} \\
45 \texttt{tl\_buf[1].buflen = pdu\_data\_len;} \\
46 \texttt{tl\_buf[2].bufptr = NULL;} \\
47 \texttt{tl\_buf[2].buflen = 0;} \\
48 \texttt{tl\_buf\_cnt = 2;} \\
49
50 \texttt{ktrc\_write(subsys\_id,} \\
51 \texttt{kind,} \\
52 \texttt{path\_id,} \\
53 \texttt{device\_id,} \\
54 \texttt{&tl\_buf,} \\
55 \texttt{tl\_buf\_cnt);} \\
56 \texttt{)} \\
57 \texttt{)} \\
58 \texttt{return(0);} \\
59 \texttt{)}

\textbf{KLOG\_CK()}

This macro allows the calling process to find out whether logging is enabled for the current subsystem. The returned value is one (1) if logging is enabled. It is defined as:

\texttt{KLOG\_CK(subsys\_id, log\_class)}

\texttt{subsys\_id} \quad \text{Unique ID number (assigned by Hewlett-Packard) of the calling subsystem.}

\texttt{log\_class} \quad \text{Defines the classification of event. All classes are defined in the header file subsys\_id.h (see also “Classify Trace Data” on page 286). Four classes are defined for logging messages:

\begin{itemize}
  \item \texttt{Informative} \quad \text{Normal messages only}
  \item \texttt{Warning} \quad \text{Warning messages}
  \item \texttt{Error} \quad \text{Error condition messages}
  \item \texttt{Disaster} \quad \text{Critical error messages}
\end{itemize}
**kget_log_instance()**

This call accepts no parameters but returns a unique log instance value. The log instance helps thread log messages together so the user can easily identify the messages that result from the same event. A change in the log instance means a new event is being logged.

The log instance value 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.)

**klogg_write()**

This routine is used to send log messages to the kernel trace and log facility.

Prefiltering is done at the time of the log call, and unwanted messages are dropped. This routine always returns a success of zero and is defined as:

```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 286). 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, tl_packet points to an mbuf chain. If the value is greater than zero, it is the number of the iovec structure (as defined in uio.h) the tl_packet points to.

Logging Code Sample

The following scenarios describe 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 allowed */
#define LOG 1
#define FALSE 0
...
extern int log_instance;
extern unsigned short kget_log_instance;
...
int
int log_disaster()
{
    int class;
    int device_id;
```
```c
23 event_data_type event_data;
24 short subsys_id;
25 struct iovec tl_buf[MAX_BUF+1];
26 int tl_buf_cnt;
27
28 /*
29 * Set up variables for call to KLOG_CK()
30 */
31 subsys_id = MY_SUBSYS_ID_NUMBER; /* assigned by HP */
32 class = DISASTER;
33 device_id = -1; /* -1 means not applicable */
34
35 if (KLOG_CK(subsys_id, class)
36 {
37 /*
38 * Logging enabled for this subsystem
39 * and class combination.
40 */
41
42 if (log_instance == 0)
43 {
44 /*
45 * There was no previous log instance
46 * associated with this event. This is
47 * the first module to encounter the
48 * problem, so it gets the log instance.
49 * Log instance should be available to
50 * all modules in the subsystem and to
51 * other subsystems.
52 */
53 log_instance = kget_log_instance();
54 }
55 }
56
57 event_data.event_number = THIS_EVENT_NUMBER;
58 event_data.event_type = THIS_EVENT_TYPE;
59 /*
60 * Additional data about the event can be
61 * placed in the data structure. This
62 * data structure is entirely up to the
63 * local developer to design. The
64 * subformatter for this subsystem must
65 * 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
* hold individual pieces of information.
*
* Callers should NOT pass strings in this
* function; the event number as shown in
* this example should be used to
* generate an NLS string from a message
* catalog in the subformatter.
* /

    tl_buf[0].bufptr = *event_data;
    tl_buf[0].buflen = sizeof(event_data_type);
    tl_buf[1].bufptr = NULL;
    tl_buf[1].buflen = 0;
    tl_buf_cnt = 1;

    klogg_write(subsys_id,
                class,
                device_id,
                log_instance,
                &tl_buf,
                tl_buf_cnt);

    return(0);
Formatting Networking Trace/Log Messages

The following sections detail facilities and network device driver developer responsibilities for formatting trace/log data output. Some sections provide code and output examples. “Designing a Subformatter” in Appendix B, shows a generic style subformatter that can handle the preceding logging and tracing examples using the basic calls of `netfmt`.

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.

The `netfmt` formatter 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; it 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 280.

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 the following sections, with recommendations for usage. Figure 7-8, “Network Tracing, Logging Elements and Data Flow I,” identifies important tracing/logging subroutines and shows their relationship to tracing/logging facilities shown in Figure 7-7, “Master Agent/Subagents Relationship,” You can add additional functions, if necessary.

Appendix B addresses what the formatted data could look like, and gives further information on this subject. To review, the formatter provides the following features:

- Loads subformatter shared libraries
- 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 B.
Figure 7-9  Network Tracing, Logging Elements and Data Flow II

- ktl driver
- Subsys Table
  - Port
- nktl daemon
- User Space
- kernel
- netfnt
- nettgen.conf
- [ASCII records]
  - Initial Log Class
  - Message Catalog
  - Subformatter Library
  - Subformatter Function
  - Subformatter Options
  - Subsystem ID
  -Subsystem Name
- ntl_reader
- Trace or Log Data File
- klogg_write() ktrc_write() KLOG_CK() KTRC_CHECK()
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. 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. Furthermore, this scheme is difficult to implement for kernel drivers.

A more sophisticated method 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()` routines) 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 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.
Configuring a Networking Driver Through SAM

System Administration Manager (SAM) is an HP-UX system administration tool. For the HP-UX system administrator, it provides both GUI and cursor based interfaces to configure the system's resources like file systems, network etc. Starting from HP-UX version 11i, SAM has included support for configuring network interface cards controlled by third party device drivers. This is a desirable feature for a third party driver since the system administration for both native HP drivers and the third party drivers falls under the same umbrella of SAM, easing the job of a system administrator.

This section presents an overview of the various components involved in lending SAM support for third party drivers, generic networking configuration supported by SAM, additional configuration and init script files required for integrating with SAM, control flow of `get/set` requests from SAM to the driver, and defines required at the driver.
SAM Interface to IHV Network Drivers

The following diagram shows various components involved in SAM support to a third party network driver.

Figure 7-10  Support Components for Third Party Network Driver

The network driver is assumed to be a STREAMS DLPI networking driver. For further information on writing a STREAMS DLPI networking driver, please refer to earlier sections in this chapter. The remaining document discusses only the issues that are relevant to modifying an existing STREAMS DLPI network driver for SAM support.

A network driver that requires SAM support has to provide a driver configuration file, a driver init script, a driveradmin tool and additional support at the driver.
To interact with a driver, first SAM needs to know the driver name and the device special file name of the driver. The driver configuration file located in the directory `/etc/rc.config.d` contains the required information. The driver configuration file also lists parameters of the driver which can be configured. Specific details on the driver configuration file along with an example are provided in later sections.

Based on the driver name, SAM locates a driver `init` script located in the directory `/usr/sbin`. The driver `init` script is a nexus between SAM and the driver and is called by SAM to modify settings of the parameters that the driver supports. SAM always calls a driver's `init` script in a particular format. It is the responsibility of the `init` script to correctly parse SAM request and issue commands to the driver to modify the NIC settings. Specific details to the driver `init` script file along with an example are provided in later sections.

A driver `init` script passes the request to the `driveradmin`, which issues a request to the driver to set a value for a parameter passed by the `init` script. For more details on writing a `driveradmin` tool for a network driver, refer to “Shared Library Examples for the `driveradmin` and `lanscan` commands” in Appendix C.

A driver has to support additional `ioctl` to process requests originated at SAM to modify settings of the driver parameters. A driver also needs to support additional HP-DLPI primitives to process requests originating from SAM to get current settings of the driver. Specific details on the changes required at a driver are given in later sections.

SAM uses an internal SAM executable `/usr/sam/lbin/laninfo` to list all the network interfaces on a host. `laninfo` issues HP-DLPI requests and `ioctl` requests to the driver to get the current settings from the driver. The control flow of `get` and `set` requests from SAM to the driver is discussed in later sections.
Generic Network Configuration Supported by SAM

SAM supports GUI based configuration of Network Interface Cards (NICs). Various card parameters of a NIC are set through SAM. Under basic configuration, SAM supports configuration of IP Address, Subnet Mask, Host Name Aliases, speed, duplex and autonegotiate. Under advanced options, it supports the configuration of Station Address, Broadcast Address and the MTU size.

SAM uses the `ifconfig` utility for setting the IP address, Subnet mask and Broadcast address. If user has given a host name and an alias, SAM updates the `/etc/hosts` files with host name and alias information.

For configuring speed, duplex, autonegotiate, station address and MTU size, SAM calls the driver `init` script.

The following is an example of how SAM calls the driver `init` script to set the various parameters:

```
/usr/sbin/hpenet_init start
-1 0 enet0 0 0x001083F60E72 1500 10hd
```

Driver Configuration File and Init Script

SAM requires a driver configuration file to obtain necessary information to access the driver, and a driver `init` script to modify the driver parameters.

Driver Configuration File

The driver specific configuration should be saved under the directory `/etc/rc.config.d` with a file name of `hp<driver_name>conf`. For example, if the driver name is `enet`, the driver configuration file name is `hpenetconf`.

SAM gets two important pieces of information from this file: configuration parameters, and the device special file name.

- Obtaining Configuration Parameters

The `HP_<DRIVER>_INIT_ARGS` statement in this file defines the parameters that are configurable on this card. SAM configures only those values that are in this statement. If a parameter does not apply here, SAM does not support its configuration. It is set as follows:
Creating Networking Device Drivers
Configuring a Networking Driver Through SAM

HP_<DRIVER>_INIT_ARGS="HP_<DRIVER>STATION_ADDRESS
HP_DRIVER_MTU
HP_<DRIVER>_SPEED"

This is a fixed setting and should be followed in the driver configuration file.

- Obtaining Device Special File Name

The configuration file should also contain the third party device special file name, which is specified as:

IHV_DLPI_DEVICE_NAME=/dev/<driver_name>

for example,

IHV_DLPI_DEVICE_NAME=/dev/enet

The hp driver_nameconf is created and owned by the driver.

SAM updates the modified configuration parameters in this driver specific configuration file by using the ch_rc command. For example, after setting the interface network speed settings to 100 full-duplex, MTU size to 1200, and MAC address to 0x000629BE051C, the following is a set of entries that SAM places after each operation on the NIC.

HP_<DRIVER>_INTERFACE_NAME[1]=enet1
HP_<DRIVER>_SPEED[1]=100FD
HP_<DRIVER>_MTU[1]=1200
HP_<DRIVER>_STATION_ADDRESS[1]=0X000629BE051C

---

**NOTE**

SAM modifies this file whenever network parameters are modified. Changes to this file should not be performed after initialization to prevent erroneous configurations.

---

Refer to the sample driver_nameconf file, hpenetconf provided in the DDK, to build a driver conf file for the third party driver.

**Driver INIT Script**

The driver specific init script should be saved under the directory /usr/sbin with a file name of hp driver_name_init. For example, if the driver name is enet the driver init script file name is hpenet_init.

This init script is for the exclusive use of SAM, and should be different from any other init scripts a driver might have.
A driver's init script is the glue between SAM and the driver. SAM sends requests to modify network configuration parameters to the driver via the init script. SAM always calls the driver init script with a fixed sequence of parameters. The calling convention of the init script by SAM is:

```
/usr/sbin/hp<driver_name>init start <major#> <instance#> <interface name> <nmid> <station address> <mtu> <interface speed>
```

The parameter definitions are:

- **major#**: Major number of the driver. Typically this is not used by the init script. Ignore this field.
- **instance#**: Instance of the NIC on which the set operation is to be performed.
- **interface name**: Network interface name, as shown in lanscan output. Typically this is not used by the init script. Ignore this field.
- **station address**: New MAC address; set the specified NIC's MAC address with this.
- **mtu**: New MTU value; set the specified NIC's MTU with this.
- **interface speed**: New link speed setting for NIC. The valid values for speed settings are:
  - 10HD: 10Mbps, half-duplex
  - 10FD: 10Mbps, full-duplex
  - 100HD: 100Mbps, half-duplex
  - 100FD: 100Mbps, full-duplex
  - 1000HD: 1000Mbps, half-duplex
  - 1000FD: 1000Mbps, full-duplex
  - AUTO_ON: Autonegotiation ON

For example, to set interface network speed settings to 100 Full-duplex, MTU size to 1200, and MAC address to 0x000629BE051C, SAM can call the corresponding driver's init script as:
Any message from the init script after successfully carrying out the set operation is to be redirected to /dev/null. If the init script fails, the error message provided by the script will be displayed by SAM.

Typically a driver init script will call the driver\_name\_admin tool to set the values at the driver.

Refer to the sample driver init script hpenet\_init in the DDK to construct the hpdriver\_name\_init script for the third party driver.

**SAM/Driver get/set Request Flow**

The following section discusses the control flow of get and set requests from SAM to the driver. Implementation details are given to process SAM requests. Where required, example code is provided from the sample driver enet for clarification.

**To get The Current Settings of the Driver**

SAM does not interact with the driver's init script to get current settings from the driver. Instead, it directly issues HP-DLPI and ioctl requests to the driver.

1. SAM issues DL_HP_PPA_REQ to the driver to get driver configuration information.

   DL_HP_PPA_REQ processing in the sample enet driver provides a good example of implementing support for it in the driver. The driver has to set the MTU field of the dl\_ppa\_info\_t structure to the current MTU size at the NIC. SAM gets the current MTU size from the field.

2. To get the speed, duplexity and autonegotiation values, SAM issues an ioctl request of command type DLPI\_LINK\_SPEED to the driver. DLPI\_LINK\_SPEED is defined in the /usr/include/sys/dlpi\_ext.h file.

   The process is:

   a. SAM allocates a variable of type struct strioctl.
Creating Networking Device Drivers
Configuring a Networking Driver Through SAM

b. SAM issues an ioctl (2) system call with the request code set to I_STR and passes the variable allocated in step a as an argument.

The following fields are passed by the ioctl:

- ic_cmd field set to the constant DLPI_LINK_SPEED
- ic_dp field points to a structure of type struct fis
- ic_len field set to sizeof(struct fis)

In the structure struct fis, the reqtype field is set to IHV_REQ_LINK_SETTING to get the current speed/duplex/autoneg settings of the link.

While processing this ioctl request, the driver sets the value.i member of the struct fis to one of the following values depending on the link status:

- IHV_SPEED_10HD (= 1) for Speed: 10 Duplex: Half
- IHV_SPEED_10FD (= 2) for Speed: 10 Duplex: Full
- IHV_SPEED_100HD (= 3) for Speed: 100 Duplex: Half
- IHV_SPEED_100FD (= 4) for Speed: 100 Duplex: Full
- IHV_SPEED_1000FD (= 5) for Speed: 1000 Duplex: Full

In addition, if autonegotiation is turned on at the NIC, the driver should bit-wise OR the above value with IHV_AUTONEG_ON (=0x10). When SAM gets this value, it performs a bit-wise AND of the returned value.i member with IHV_AUTONEG_MASK (=0xF0) to decide the status of autonegotiation.

Some devices or their drivers may or may not support autonegotiation. SAM has to know whether the device/driver supports it. Bit four in the ioctl return value indicates autonegotiation status. Bit five indicates autonegotiation support. If the support bit is not set, the autonegotiation status can be ignored by SAM. If the support bit is set, the autonegotiation status value should be considered.

SAM expects this format for the ioctl return value in the value.i member:

```plaintext
bit #  X^7  X^6  X^5  X^4  X^3  X^2  X^1  X^0
```

where X = a value associated with the bit position.
Bits 3 - 0  Indicate speed and duplexity
Bit 4  Indicates autonegotiation status (1 = ON, 0 = OFF)
Bit 5  Indicates autonegotiation support (0 = Not supported, 1 = Supported)
The following pseudo code is from the sample `enet` driver:

```c
<driver>_ioctl_function() {

....

/* Code to return speed, duplexity and autoneg values to SAM. */

case IHV_REQ_LINK_SETTING:

    datap->vtype = INTEGERTYPE;

    /* Check the current speed settings at the link and set the ioctl return value. */

    if( enet_iftp->speed == 10 )
        if( enet_iftp->full_duplex)
            datap->value.i = IHV_SPEED_10FD;
        else
            datap->value.i = IHV_SPEED_10HD;

    if( enet_iftp->speed == 100 )
        if( enet_iftp->full_duplex)
            datap->value.i = IHV_SPEED_100FD;
        else
            datap->value.i = IHV_SPEED_100HD;

    /* Check for autonegotiation status */
    if( enet_iftp->conn_type == MII_AUTOSENSE ){
        /* Autonegotiation supported */

        /* Set autonegotiation status bit to ON */
        datap->value.i |= IHV_AUTONEG_ON;

        /* Set autonegotiation suport bit to ON */
        datap->value.i |= IHV_AUTONEG_SUPPORTED;
    }

    break;

    ....
}
```
If the speed is unavailable, the ioctl for IHV_REQ_LINK_SETTING will return its error code (presently -1) and the global variable errno will be set to ENXIO. If SAM passes in any reqtype other than the above, the ioctl will return its error code and the variable will be set to EINVAL.

To Set the Current Settings of the Driver

To set modified network configuration parameters, SAM sends requests to the driver via the driver init script. The driver init script in turn calls the driveradmin to process the SAM request. Since the interface between the driveradmin and the driver is driver dependent, the way modified configuration parameters are passed to the driver can’t be described here. In a typical implementation, a driveradmin tool may issue an ioctl to the driver for each of the modified configuration parameters. The sample driver enet implementation and the sample enetadmin tool provided with the DDK will provide the required information on handling ioctl requests.

Defines Required at the Driver

The following defines have to be included in a driver header file. It is very important that the values of these defines are not modified by the driver developer. SAM uses these values, and if the driver doesn’t follow them the existing mechanism between SAM and a third party driver does not work.

```c
/*
 * The following defines are for SAM support
 */
#define IHV_SPEED_10HD  1
#define IHV_SPEED_10FD  2
#define IHV_SPEED_100HD 3
#define IHV_SPEED_100FD 4
#define IHV_SPEED_1000FD 5
#define IHV_AUTONEG_ON        0x10
#define IHV_AUTONEG_SUPPORTED 0x20
#define IHV_REQ_LINK_SETTING  0x77
```

Chapter 7
Network Monitoring Commands

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 C for the benefit of driver developers.

It is recommended the utilities be named `driveradmin` and `driverlinkloop` to avoid conflict with the default HP supplied commands. It is also 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 C, “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 216 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.
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, 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

**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 located in `/usr/lib/lanadmin`.
- The shared library function must be named `dsdriver_name`(with `driver_name` as the string stored in `name` in the `hw_ift` 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 Fiber 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 (NLS) message catalogs. Refer to the HP Native Language Support: User's Guide for further information.

Example for Writing a Shared Library

The dsenet.c file provides an example of how to write a shared library. The actual obtaining 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 C, “Shared Library Examples for the driveradmin and lanscan Commands”.

Shared Library Message Catalog

The shared library function, dsenet(), 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 the `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 216 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 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/libpedriver_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 shown above is hardcoded in `lanscan`. 
3. After loading 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:

```c
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:

```c
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:

```c
/usr/lib/nls/C/pedriver_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 */

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/dev_enet 0x0060B07EAAFD
Writing a SCSI Interface Driver
This chapter provides information on designing and developing a SCSI interface driver, also known as a Host Bus Adapter driver.

The next section of this chapter describes data structures and interfaces provided by SCSI services and WSIO to an interface driver. Included is the flow of an I/O request as it passes different layers of the SCSI subsystem.

Later in the chapter, steps involved in the interface driver development are detailed. This includes driver installation and initialization in WSIO CDIO, SCSI subsystem specific driver initialization, DMA mapping utilities, and interrupt handling.

While explaining the interface driver development, code snippets from HP-UX sample interface driver for Qlogic’s ISP12160 SCSI Ultra3 interface card are provided as an example.
Overview of HP-UX SCSI Subsystem

A SCSI interface driver is part of the SCSI subsystem in HP-UX. This is also called a SCSI Host Bus Adapter (HBA) driver in the industry.

The following figure is the Mass Storage Stack in HP-UX. It illustrates the components the SCSI subsystem interacts with.

The SCSI subsystem can be broadly divided into three layers: SCSI device drivers, SCSI services and SCSI interface drivers.

SCSI device driver refers to a driver for a SCSI device such as a disk or tape. It is responsible for implementing one or more device models; each for some set of SCSI devices. SCSI tape drivers and SCSI disk drivers are typical examples of a device driver. The device driver is responsible for all device model operations for each device in the set of supported SCSI devices. Refer to Chapter 9, Writing SCSI Device Drivers in the DDG for more details.
A SCSI pass-through driver (also referred to as pass-through driver) is a character mode device driver. It is provided for customers wishing to integrate special purpose SCSI peripherals without having to worry about the complexities of writing a kernel driver. The SCSI subsystem also uses a pass-through driver interface to discover devices off a SCSI bus during “ioscan”. No special processing is needed at the interface drivers to handle pass-through I/Os. More discussion on pass-through drivers is beyond the scope of this paper. It is mentioned here only to give a complete picture of the SCSI subsystem.

The SCSI services layer is provided to simplify the jobs of, and impose a common structure on, device drivers and interface drivers. SCSI services layer is composed of data structures and functions. All of the functionality which is consistent for all device drivers and/or all interface drivers is implemented by SCSI services. SCSI services also rely on the data specified and set by device and interface drivers on its functionality.

WSIO CDIO is Device Driver Environment (DDE) in which an interface driver is developed. Refer to Chapters 2, 3 and 5 of the DDG for detailed discussion on the WSIO and how a driver fits into the WSIO CDIO.

An interface driver is responsible for managing the SCSI bus hardware. It takes a fully specified SCSI request and manipulates the hardware (including any necessary DMA operations) to make it occur on the SCSI bus. The interface driver is responsible for managing the interconnect between the initiator and targets, I/O timeouts, and other task management functions such as bus reset, abort task, and bus device resets. When re-selection is not completely implemented in the HBA, the interface driver is also responsible for managing the data structures necessary for nexus re-establishment on re-selection.
External Interfaces to an Interface Driver

A SCSI interface driver uses services of various subsystems of HP-UX. This section describes the interfaces a SCSI interface driver interacts with.

WSIO CDIO DDE

Refer to Writing a Driver in chapter 5 of the DDG for an understanding of the data structures and WSIO services. The SCSI subsystem specific driver initialization is discussed later in this chapter.

SCSI Services Interface

SCSI Services Interface includes data structures and functions which are provided by the SCSI subsystem to ease the development of an interface driver. Since SCSI services interact with both the device driver and the interface driver, this chapter discusses only such SCSI services which are visible to an interface driver.

Functions

Of the specified functions, some are provided by the interface driver (driver entry points) for use by services and the others are provided by services (service calls) for use by the interface driver.

Interface Driver Service Entry Points The following is a description of the prototypes and usage for each.

1. Interface Driver Open

    int if_open(dev_t dev)

    The SCSI subsystem allows, but does not require, the interface driver to specify a logical unit open function. On all logical opens, the SCSI subsystem checks the if_open field of the scsi_ifsw structure for the SCSI bus. If if_open field is not NULL, the SCSI subsystem calls it with the device number of the device being opened as its sole argument.

    It is never called under interrupt context and is allowed to sleep.
The SCSI subsystem provides protection that blocks all other opens and closes to the same logical unit until it returns.

2. Interface Driver Close

```c
void
if_close(dev_t dev)
```

The SCSI subsystem allows, but does not require, the interface driver to specify a logical unit close function. On all logical unit closes, the SCSI subsystem checks the `if_close` field of the `scsi_ifsw` structure for the SCSI bus. If `if_close` field is not `NULL`, the SCSI subsystem calls it with the device number of the device being closed as its sole argument.

It is never called under interrupt context and is allowed to sleep.

The SCSI subsystem provides protection that blocks all other opens and closes to the same logical unit until it returns.

3. Interface Driver Start

```c
void
if_start(struct isc_table_type * isc)
```

The SCSI subsystem requires the interface driver to specify a start function. Using this entry point, the SCSI subsystem informs the interface driver that it has work to do.

Since `if_start` can be called on the interrupt control stack (ICS) it is not permitted to sleep under any circumstances.

The SCSI subsystem may call `if_start` at any time; for example, when the bus is dormant or not and in a process’ context, or under interrupt. In all cases, the interface driver must continue to execute I/Os that are on the select queue until the bus becomes dormant.

A bus is considered dormant if it has no active I/Os. An I/O is considered to be active from the time it is enqueued on the select queue until `scsi_cbfn` is called for the I/O.

4. Interface Driver Reset Bus

```c
int
if_reset_bus(dev_t dev)
```
The SCSI subsystem allows, but does not require, the interface driver to specify a bus reset function. When the SCSI subsystem wants to reset a bus, it checks the `if_reset_bus` field of the `scsi_ifsw` structure for the bus. If `if_reset_bus` is not NULL, it is called with a device number identifying the bus as its sole argument. When `if_reset_bus` returns, the SCSI bus should have been reset.

I/Os that are disconnected and the I/Os that are connected with the bus (if any) at the time of the reset should be returned to the SCSI subsystem with the appropriate status field set to `SCTL_INCOMPLETE`. That is, if it was the Request Sense resulting from a check condition that was terminated by the reset, then `scb->sense_action` should be set to `SCTL_INCOMPLETE`. Otherwise, `scb->cdb_status` should be set to `SCTL_INCOMPLETE`. “struct scb” is described under data structures later in this section.

The SCSI subsystem makes this call only in response to `SIOC_RESET_BUS` ioctl request.

5. Interface Driver Bus Device Reset (BDR)

```c
int
if_bdr(dev_t dev)
```

The SCSI subsystem allows, but does not require, the interface driver to specify a BDR function.

It is intended to serve as a way for the SCSI subsystem to direct the interface driver to send a SCSI BDR message to the indicated target.

The SCSI subsystem makes this call only in response to `SIOC_RESET_DEV` ioctl request.

6. Interface Driver Abort

```c
int
if_abort(dev_t dev)
```

The SCSI subsystem allows, but does not require, the interface driver to specify an abort function.

It is intended to serve as a way for the SCSI subsystem to direct the interface driver to send a SCSI ABORT message to the indicated logical unit.

The SCSI subsystem makes this call only in response to `SIOC_ABORT` ioctl request.
SCSI Service Interface Driver Service Calls  The following is a description of the prototype and usage of each.

1. SCSI Subsystem Callback

```c
void scsi_cbfn(struct buf * bp)
```

When the interface driver finishes with an I/O, it returns the I/O to the SCSI subsystem by calling `scsi_cbfn` with the `bp` as its sole argument.

The interface driver relinquishes all rights to access `bp`, `scb` and `*scb->if_scb` once it calls `scsi_cbfn()`. Of course, the `bp` may be reused later for another I/O, and similarly for the `scb` and `*scb->if_scb`, although they will not necessarily be related in subsequent I/Os.

If the interface driver has attached a sense buffer to `scb->sense_data`, the `sense_data` buffer must be valid till `scsi_cbfn()` returns. The interface driver is forbidden from accessing it until `scsi_cbfn()` returns. It is important to note that the allocation and management of this buffer for holding `sense_data` is the responsibility of the interface driver.

This can be called either in process or interrupt context. This must not be called with any locks held since the SCSI services may call the interface driver’s start entry point before it returns.

2. SCSI Subsystem Queueing Functions

The SCSI subsystem provides three simple routines for managing queues or lists of `bp`'s. They may be used by the interface driver for managing private queues, but they must be used to manage the select queue from which the interface driver gets `bp`'s for execution.

It is important to note that the driver has to acquire and release a SCSI bus lock before and after calling one of the queue routines.

```c
#define TAIL 0
#define HEAD 1

void scsi_enqueue(struct buf ** qp, struct buf * bp, int where)
```
scsi_enqueue simply enqueues bp at the HEAD or TAIL of a circular list; qp is a pointer to the list header which is a pointer to the head of the list. If “where” is HEAD, the bp is inserted ahead of the list, otherwise it is added to the tail of the list.

This must be called with scsi_bus lock held.

```c
struct buf *scsi_enqueue(struct buf ** qp, int where)
```

scsi_enqueue simply dequeues the bp at HEAD or TAIL of the list *qp based on the value of where and returns the bp. This returns NULL when the queue is empty.

This must be called with scsi_bus lock held.

```c
struct buf *scsi_dequeue(struct buf ** qp, int where)
```

scsi_dequeue_bp tries to dequeue bp from wherever it may be in the queue *qp. Returns bp when found on the queue. Returns NULL when not found on the queue.

This must be called with scsi_bus lock held.

```c
struct buf *scsi_dequeue_bp(struct buf ** qp, struct buf * bp)
```

3. Open Device Tree Access Functions

Functions for acquiring pointers to the open device tree data structures from a device number are provided by services.

```c
struct scsi_bus *m_scsi_bus(dev_t dev)
m_scsi_bus evaluates to a pointer to the scsi_bus structure for dev.
```

```c
struct scsi_tgt *m_scsi_tgt(dev_t dev)
m_scsi_tgt evaluates to a pointer to the scsi_tgt structure for dev.
```

```c
struct scsi_lun *m_scsi_lun(dev_t dev)
m_scsi_lun evaluates to a pointer to the scsi_lun structure for dev.
```

```c
struct isc_table_type *m_scsi_isc(dev_t dev)
m_scsi_isc evaluates to a pointer to the isc_table_type structure for dev.
```
4. Micro Functions

A few macro functions are provided by the SCSI subsystem for the convenience of the interface driver.

\[ \text{m_bus_id(dev_t dev)} \]

\text{m_bus_id} evaluates to the bus ID of the SCSI bus of dev. The SCSI bus ID is also same as the card’s instance number.

\[ \text{m_tgt_id(dev_t dev)} \]

\text{m_tgt_id} evaluates to the target ID, i.e. SCSI bus address of dev.

\[ \text{m_lun_id(dev_t dev)} \]

\text{m_lun_id} evaluates to the logical unit number of dev.

Data Structures

In addition to the functions, SCSI subsystem also provides data structures to pass data across different layers of the SCSI stack. Some WSIO data structures are also used to pass data across the SCSI subsystem. Of the specified data structures and fields, some are owned by the interface driver i.e., maintained only by the interface driver (after being initialized to zero by services). These exist because some part of the SCSI subsystem other than the interface driver needs access to the information, and source of this information is the interface driver. It could be services or a device driver that needs access to the information, but that is irrelevant to the interface driver.

Other specified data structures and fields are owned by some other part of the SCSI subsystem and are available for use by the interface driver. Those remaining are owned by neither the interface driver nor some other part of the SCSI subsystem, but may be accessed and modified by either.

The \( \text{bp, scb, *scb->if_scb} \) and the data buffer for an I/O are available for use by the interface driver only while the I/O is active.

In this section data structures are present in C syntax and fields that are relevant to the current discussion only are shown.

SCSI Control Block  The buf structure is not large enough to hold all state information associated with a SCSI I/O attempt. An scb is attached to a buf by SCSI services to hold the temporary state information until the I/O is completed. The bp/scb association does persist for retries. "struct buf" is described later in the section.
Some of the fields that are of interest to an interface driver writer are explained below:

```c
struct scb {
    void     *if_scb;
    struct   scsi_lun  *lp;
    ubit32   flags;
    ubit32   max_msecs;
    ubit8    cdb[SCSI_MAX_CDB_LEN];
    ubit8    cdb_len;
    ubit32   io_id;
    ubit8    tag;
    ubit32   cdb_status;
    ubit32   data_resid;
    ubit32   sense_status;
    ubit8    sense_bytes;
    ubit8    *sense_data;
} *scb;
```

```c
#define SCB_SDTR SCTL_INIT_SDTR
#define SCB_WDTR SCTL_INIT_WDTR
#define SCB_4BYTE SCTL_4BYTE
#define SCB_2BYTE SCTL_2BYTE
```

**NOTE**

Interface driver is not allowed to set the following fields:

- lp
- flags
- max_msecs
- cdb
- cdb_len
- io_id
- tag

**NOTE**

If a field is already described earlier, it will only be mentioned here. For more details on such a field, please refer to earlier sections.
Writing a SCSI Interface Driver

External Interfaces to an Interface Driver

**scb->if_scb**

Is a pointer to `ifsw->if_scb_size` bytes allocated by SCSI services and reserved for use by the interface driver. The pointer is initialized at scb creation time by services and the data area is `bzero`ed by services for each I/O attempt prior to putting the I/O on the select queue. It is not touched by services at any other time. The `if_scb` area is later freed by the SCSI services along with the scb.

**scb->lp**

Is a pointer to the `scsi_lun` structure in the open device tree with which this scb is associated. If the scb belongs to a per-lun pool of scb’s as opposed to a per-bus pool, then `scb->lp` is initialized at scb creation time by SCSI services and never changed. Otherwise, the scb belongs to a per-bus pool, and `scp->lp` is only valid while the scb is associated with a bp. Then `scb->lp` points to the `scsi_lun` structure associated with `bp->b_dev`.

**scb->flags**

The interface driver may check these bits in the flag for proper functionality. Bits in `scb->flags` that are relevant to an interface driver are:

- **SCB_NO_DISC**
  
  This bit indicates that the disconnect privilege should not be granted in the identify message.

- **SCB_SDTR**
  
  If this bit is set and **SCB_WDTR** is not set, the interface driver should initiate SDTR negotiation immediately following the Selection, Identify or tag message, whichever comes last, and before sending the CDB for the I/O.

- **SCB_WDTR**
  
  This bit directs the interface driver that a wide negotiation should be initiated immediately following the Selection, Identify or tag message, whichever comes last, and before sending any CDB for the I/O. If `(tp->state & T_ENABLE_SDTR)` or `(scb->flags & SCB_SDTR)` is also set, the interface driver should initiate
SDTR negotiation immediately following the WDTR negotiation. The wide negotiation should always precede the synchronous negotiation, since a wide negotiation resets the link to asynchronous.

**SCB_4BYTE**

This bit is a hint to the interface driver that the target will never change phase while in data phase on other than a 4-byte boundary at the beginning of the data transfer without subsequently restoring the data pointer (implicitly or explicitly) to a previously aligned value and re-transferring data up to and beyond the point of disconnection to an aligned boundary. The phase change at the end of the I/O need be considered only if the amount of data transferred may be less than that requested in `bp->b_bcount`. Note that **SCB_4BYTE** does not imply that `bp->b_count` is a multiple of four or that `bp->b_un.b_addr` is 4-byte aligned. Note also that the phase change out of data phase if all `bp->b_bcount` bytes have been transferred is not subject to the alignment restructuring.

**SCB_2BYTE**

This bit is the same as **SCB_4BYTE** except that phase changes are only restricted to even boundaries.

**SCB_ORDERED_TAG**

Denotes that ordered tags are intended to be used for this device.
scb->max_msecs Minimum number of milliseconds the interface driver is to allow for this I/O from the time of Selection until Command Complete in parallel SCSI or as close to that as possible within a given HBA architecture. If scb->max_msecs milliseconds elapses and the I/O has not completed, the interface driver is encouraged to abort the I/O with Abort or Abort Tag as appropriate. The interface driver can run a timer routine periodically to watch out for the I/Os that are timed out. A value of zero indicates the interface driver should never abort this I/O based solely on the amount of time since Selection.

scb->cdb Holds the SCSI command bytes for this I/O.

scb->cdb_len The number of bytes in the cdb. This can be a maximum of SCSI_MAX_CDB_LEN.

scb->io_id Is a unique identifier for a SCSI I/O. It is initialized when the scb is associated with a bp and is unique across all SCSI busses.

scb->tag The tag value allocated for this I/O by the SCSI subsystem in accordance with the interface driver’s direction via ifsw->if_max_tag. It is recommended that the interface driver use this value as the tag value for the I/O if the I/O will be tagged, but it is not required. Currently there can only be 256 tags per bus. The tag value may not remain same for retried I/Os.

scb->cdb_status Indicates the status of the I/O command. If the I/O attempt completes with no phase sequencing errors and without being aborted or timing out, the interface driver sets scb->cdb_status to S_GOOD. If the selection phase times out, the interface driver sets cdb_status to SCTL_SELECT_TIMEOUT. If the I/O is not even attempted because of bogus data in the bp or scb, the interface driver sets cdb_status to SCTL_INVALID_REQUEST. If the I/O is not attempted or does not complete for any other reason, cdb_status is set to SCTL_INCOMPLETE. If there is a Contingent Allegiance condition, the cdb_status is set to
S_CHECK_CONDITION to request an auto-sense request. `scb->cdb_status` must be set by the interface driver prior to returning the bp via `scsi_cbfn`.

Refer to `scsi.h` for the valid values of `cdb_status`.

`scb->data_resid`

If the I/O attempt completes with no phase sequencing errors and without being aborted or timing out, the interface driver sets `scb->data_resid` such that `bp->b_count - scb->data_resid` is the offset from `bp->b_un.b_addr` of the first byte not transferred by the target, i.e., number of bytes transferred = `bp->b_bcount - scb->data_resid`. Even if the I/O attempt is failed for some reason, it is advisable to set the `scb->data_resid` to indicate the number of bytes that are not yet transferred. Setting this field will have no adverse effect. `scb->data_resid` must be set by the interface driver prior to returning the bp via `scsi_cbfn`.

`scb->sense_status`

Represents the status of the automatic request sense that is performed if `scb->cdb_status` is `S_CHECK_CONDITION`. If the Request Sense completes with no phase sequencing errors and without being aborted or timing out, the interface driver sets `scb->sense_status`. Otherwise, `scb->sense_status` is undefined and will not be referenced by the SCSI subsystem on callback. The possible values for `scb->sense_status` are the same as those for `scb->cdb_status` except that `SCTL_INVALID_REQUEST` cannot be used. `scb->sense_status` represents the result of the automatic Request Sense in the same way that `scb->cdb_status` represents the result of attempting `scb->cdb`. It must be set by the interface driver before returning the bp via `scsi_cbfn`. If there is any sense data, the `sense_status` has to be set to `S_GOOD`.

`scb->sense_bytes`

Number of bytes of data received in response to the automatic request sense if one was performed. It is valid only if `sense_status` is valid and is neither
Writing a SCSI Interface Driver

External Interfaces to an Interface Driver

SCTL_SELECT_TIMEOUT nor SCTL_INCOMPLETE.

scb->sense_bytes is the offset from
scb->sense_data of the first byte of sense data not
transferred by the target. It must be set by the
interface driver prior to returning the bp via
scsi_cbfn.

scb->sense_data

If scb->cdb_status is Check Condition and the
resulting Request Sense completes with no phase
sequencing errors and without being aborted or timing
out, and if scb->sense_status is not zero, the
interface driver sets scb->sense_data. Otherwise,
scb->sense_data is undefined and will not be
referred to by the SCSI subsystem on callback. The
interface driver sets scb->sense_data to point to a
KERNELSPACE buffer containing the sense data; its
size must be at least scb->sense_bytes. It must be set
prior to returning the bp via scsi_cbfn and the
interface driver must not modify the buffer for the
duration of scsi_cbfn. When scsi_cbfn returns, and
not until, the interface driver can reuse the buffer.

ISC  Each instance of an interface card has an Interface Select Code
(ISC) entry that the system maintains in an internal table. Each ISC
entry, defined as an isc_table_type structure, is used by WSIO to
maintain interface driver information. Refer to Understanding HP-UX
I/O Subsystem Features in Chapter 3 of the DDG for a detailed
discussion on this structure. It is not part of the SCSI subsystem.

Some of the fields of the isc_table_type structure are reserved for use
by the interface driver as specified by the SCSI subsystem. They are
described in this section. Others are reserved for use by the interface
driver at its discretion. These reserved fields are - ppoll_flag,
ppoll_mask and ppoll_sense. However, these fields are not typically
used by the interface driver. A number of these fields have been renamed
using #defines to more accurately reflect their meaning within the
context of the SCSI subsystem.

Fields specific to a SCSI interface driver are explained here.
struct isc_table_type {
    char my_address;
    struct gfsw *gfsw;
    caddr_t *ifsw;

    unsigned char int_enabled;

    unsigned char spoll_byte;
    unsigned char tfr_control;

    struct buf *ppoll_f;
    struct buf *ppoll_l;

    int lock_count;
    struct buf *event_f;
    struct buf *event_l;
    struct buf *status_f;
    struct buf *status_l;

    char ppoll_flag;
    unsigned char ppoll_mask;
    unsigned char ppoll_sense;

    struct buf *owner;
    unsigned int state;
    int *card_ptr;

    unsigned char my_isc;
    char bus_type;
    caddr_t if_reg_ptr;
    caddr_t if_drv_data;
    void *if_isc;

    int if_id;
} *isc;

#define bus_max_width int_enabled
#define bus_min_sdtr_period spoll_byte
#define bus_max_reqack_offset tfr_control
#define tgt_wdtr_done ppoll_f
#define tgt_wdtr_width ppoll_l
#define tgt_sdtr_done lock_count
#define tgt_sdtr_period event_f

#define if_char0 ppoll_flag
#define if_uchar1 ppoll_mask
#define if_uchar2 ppoll_sense

## NOTE

Interface driver is not allowed to set if_drv_data.

isc->myaddress SCSI bus address of the initiator. It is a binary value from zero to fifteen; it is not a power of two representing the data bit used by the initiator for selection and re-selection. isc->my_address is initialized by the interface driver’s attach routine.

isc->gfsw Pointer to the interface driver’s gfsw structure. The SCSI subsystem does not require that the interface driver provide a gfsw structure.

isc->ifsw Pointer to the interface driver’s scsi_ifsw structure. It is initialized by the interface driver’s attach routine.

isc->bus_max_width Width of the SCSI data bus. Currently, reasonable values are 8 and 16. It is initialized by the interface driver during the driver’s attach routine. This field is later used by the SCSI services while probing for SCSI devices. Not setting this field results in not seeing devices on this SCSI bus.

isc->bus_min_sdtr_period Minimum synchronous data transfer period supported by the hardware. This field is expressed in units of 4 ns. It is initialized during the driver’s attach routine.

isc->bus_max_reqsck_offset Maximum synchronous data transfer REQ/ACK offset supported by the hardware. It is initialized during the driver’s attach routine.
isc->tgt_wdtr_done
Indicates whether or not a WDTR negotiation has occurred since the most recently detected event which resets the data transfer width to eight bits, i.e. bus reset or BDR. There is one bit for each target; the least significant bit is for target zero, the next least significant bit is for target one and so on. If the bit is set, a negotiation has occurred, otherwise no negotiation has occurred.

isc->tgt_wdtr_width
Indicates the current width of for data transfers. There is one bit for each target; the least significant bit is for target zero, the next least significant bit is for target one and so on. If the bit is set, sixteen-bit data transfers are in effect, otherwise eight-bit transfers are being used.

isc->tgt_sdtr_done
Indicates whether or not an SDTR negotiation has occurred since the most recently detected event which resets the data transfer parameters to asynchronous, i.e., bus reset, BDR or WDTR. There is one bit for each target; the least significant bit is for target zero, the next least significant bit is for target one and so on. If the bit is set, a negotiation has occurred, otherwise no negotiation has occurred.

isc->tgt_sdtr_period
Represents the location of an array of bytes indicating the current synchronous data transfer period as represented in an SDTR message. The address of isc->tgt_sdtr_period is the start of the array. There is one byte for each target; the byte at offset zero is for target zero, the next byte is for target one and so on.

isc->my_isc
Index into the isc_table array that will yield a pointer this structure.

isc->bus_type
Bus type of the interface card.
isc->if_reg_ptr
This is a virtual address corresponding to a base address register that has already had this virtual mapping done to make it usable by the driver and system.

isc->if_drv_data
Pointer to the scsi_bus structure. It is NULL if and only if the bus is not open. SCSI services set this field and an interface driver is not allowed to write to this field.

isc->if_isc
Pointer to private data structure for use by interface driver.

isc->if_id
Unique ID (device and vendor ID) of the interface.

The SCSI subsystem maintains an array of isc_table_type structures in scsi_isc[]. It is driver's responsibility to assign its isc structure to an element in scsi_isc[] array indexed by its instance number.

**Interface Driver Switch** This structure defines SCSI interface driver entry points and parameters as required by SCSI services. The interface driver's attach routine must initialize the ifsw field of the isc_table_type entry to point to a scsi_ifsw_structure. The contents of the scsi_ifsw structure specify the interface driver entry points and operational parameters to the SCSI subsystem. A detailed description of the fields follows.

```c
struct scsi_ifsw {
    ubit8 if_flags;
    ubit8 if_max_tag;
    unsigned int if_scb_size;
    unsigned int if_lun_size;
    unsigned int if_tgt_size;
    unsigned int if_bus_size;
    int (*if_open)(dev_t dev);
    void (*if_close)(dev_t dev);
    void (*if_start)(struct isc_table_type *isc);
    int (*if_abort)(struct buf *bp);
    int (*if_bdr)(struct buf *bp);
    int (*if_reset_bus)(dev_t dev);
} *ifsw;
```
NOTE

If a field is already described earlier, it will only be mentioned here. For more details on such a field, please refer to earlier sections.

**ifsw->if_flags**

Interface driver flags convey information to the SCSI services on what it supports and what not. The possible flags are:

- **IF_BUS_TAGS**
  
  This is a default tag.

- **IF_NO_TAGS**
  
  Interface driver does not support tags.

- **IF_B2_LIST**
  
  If set, it indicates the interface driver supports handling of disksort merge buffers.

- **IF_OWN Tags**
  
  Interface driver owns tagged queueing.

**ifsw->if_max_tag**

One less than the number of per-bus tags supported by the interface driver. A tag is used to differentiate I/O requests. The SCSI subsystem will use tags from zero through `ifsw->if_max_tag`, inclusive. Actually, the interface driver is not required to use the tags allocated by the SCSI subsystem, but the SCSI subsystem will not allow more than `ifsw->if_max_tag+1` active I/Os to the bus at any given time (this includes untagged I/Os).

**ifsw->if_scb_size**

The number of bytes the SCSI subsystem shall allocate and attach to each scb for use by the interface driver. `if_scb` field of scb structure is initialized at scb creation time by services and the data area is bzero'ed by services for each I/O attempt prior to putting the I/O on the select queue. It is not touched by services at any other time.
ifsw->if_lun_size
The number of bytes the SCSI subsystem shall allocate and attach to each scsi_lun structure for use by the interface driver. The if_lun field of scsi_lun structure is a pointer to ifsw->if_lun_size bytes for the use of the interface driver.

ifsw->if_tgt_size
The number of bytes the SCSI subsystem allocates and attaches to each scsi_tgt structure for use by the interface driver. The if_tgt field of scsi_tgt structure is a pointer to ifsw->if_tgt_size bytes for the use of the interface driver.

ifsw->if_bus_size
The number of bytes the SCSI subsystem shall allocate and attach to each scsi_bus structure for use by the interface driver. The if_bus field of scsi_bus structure is a pointer to ifsw->if_bus_size bytes for the use of the interface driver.

ifsw->if_open
Pointer to the interface driver’s logical unit close function. This is optional for an interface driver.

ifsw->if_start
Pointer to the interface driver’s start function.

ifsw->if_reset_bus
Pointer to the interface driver’s Bus Reset function. This is optional for an interface driver.

ifsw->if_bdr
Pointer to the interface driver’s Bus Device Reset function. This is optional for an interface driver.

ifsw->if_abort
Pointer to the interface driver’s Abort function. This is optional for an interface driver.

ifsw->if_io_max_size
Maximum size of I/O request supported by the interface driver. A value of zero (0) specifies no limit. If set, I/O requests for more than the supported size will be erred back by the SCSI services.
Interface driver data buffer alignment requirement. These fields must be set to \((n - 1)\) where \(n\) is a power of two (2). SCSI services will ensure that data buffer \((bp->b_un.b_addr)\) is \(n\)-byte aligned. The maximum of both the fields is used for buffer alignment.

**struct buf** This structure is the header for buffers in the buffer pool and otherwise used to describe a block I/O request. I/O requests are passed to the interface driver in the form of *buf* structure. Some of the fields that are of interest to an interface driver writer are explained here.

```c
struct buf {
    int32_t b_flags;
    struct buf *av_forw;
    struct buf *av_back;
    union { caddr_t b_addr;
        } b_un;
    int32_t b_bcount;
    dev_t b_dev;
    uint16_t b2_flags;
    struct buf * b_merge;
    uint16_t b_merge_cnt;
    space_t b_spaddr;
    long b_s2;
} *bp;
#define b_scb b_s2
```

**NOTE** Interface driver is not allowed to set the following fields:

- b_flags
- b_un.b_addr
- b_bcount
- b_dev
- b2_flags
- b_merge_cnt
- b_spaddr
- b_s2
Writing a SCSI Interface Driver

External Interfaces to an Interface Driver

bp->b_flags

_B_READ_ is the only bit bp->b_flags that is of interest to the interface driver and only if bp->b_bcount is not zero. If _B_READ_ is set, the I/O has a data in phase; if clear, the I/O has a data out of phase.

bp->av_forw

Position on free list of buffers if not busy. This field is used to save a pointer to the buf structure which is passed to scsi_cbfn() after an I/O request is completed.

bp->av_back

Position on free list of buffers if not busy.

bp->b_un.b_addr

Kernel virtual address of the data buffer for the I/O. This is passed to the DMA mapping routines in the interface driver.

**NOTE:**

This address may not be cache aligned. This has implications for a read request when part of the cacheline is modified by a processor write. The data after the I/O completion will be stale if cache flush occurs after inbound DMA.

Basically interface drivers must do I/Os to a temporary location for non-cache aligned portions and copy data from temporary buffers to actual data buffers after DMA completions. This differs from the buffer alignment requirement of the interface driver.

bp->b_count

Maximum number of bytes that should be transferred for the I/O.

bp->b_dev

Device number of the destination for the I/O. This is used to obtain a pointer to the _scsi_bus_ structure, a pointer to the _scsi_tgt_ structure, a pointer to _scsi_lun_ structure and a pointer to the _isc_ structure when only the _buf_ structure is available.
NOTE

SCSI services provide the following services to work with the device number:

- `m_scsi_dev()`
- `m_scsi_tgt`
- `m_susi_lun`
- `m_scsi_isc`

Additional flags to support B2_LIST buffers. If B2_LIST flag is set in `bp->b2_flags`, `bp` represents a disksort merge buffer. Interface driver specifies its capability of handling such buffers by setting `IF_B2_LIST` in `ifsw->if_flags`.

The `b_merge` field of the first `bp` represents a linked list of `buf` structures containing the actual data. The list itself is chained using the `b_merge` fields of subsequent `buf` structures.

The `b_count` field of the first `bp` represents the total data length in all buffers.

If this is a merge buffer, all data buffers (`bp->b_un.addr`) are page aligned and `bp->b_count` will be a multiple of page size (NBPG).

A count of merged requests. If this field is non-zero, DMA mapping is done via `bp_dma_setup()` instead of `dma_setup()`.

Space address of `b_un.b_addr`. This is passed to the DMA mapping routines in the interface driver.

This field is a pointer to the `scb` associated with this `bp`. It is used in an interface driver to obtain the `scb` struct which will have additional information on the I/O. Refer to the earlier discussion on struct `scb` for additional details.
**Logical Unit Structure**  A SCSI Logical Unit (LUN) structure is created per SCSI LUN. This structure is allocated and initialized when the LUN is first opened, and is deallocated on the last close by the SCSI services. This is owned by the SCSI services, but can be accessed by an interface driver. Only the fields relevant to an interface driver are described

```c
struct scsi_lun {
    struct scsi_tgt *tgt;
    void *if_lun;
    ubit8 lun_id;
    dev_t dev_minor;
    ubit32 open_cnt;
    ubit32 state;
} *lp;
#define L_TAGS 0x20
```

**NOTE**

The interface driver is not allowed to set the following fields:

- `tgt`
- `lun_id`
- `dev_minor`
- `open_cnt`
- `state`

- `lp->tgt`  Pointer to the `scsi_tgt` structure to which this logical unit structure belongs, i.e., it is a pointer to the logical unit structure's parent in the open device tree.
- `lp->if_lun`  Pointer to the data area allocated and bzero'ed on first open, by the SCSI subsystem, for use by the interface driver. Its size is specified by the interface driver in `isc->ifsw->if_lun_size`. This memory is allocated during the first open of the LUN and is freed during the last close of the LUN.
- `lp->lun_id`  Is the logical unit's identification number.
- `lp->dev_minor`  Minor number of the device minus the volume bits, i.e., the device number independent portion of the minor number.
Writing a SCSI Interface Driver

External Interfaces to an Interface Driver

lp->open_cnt  The number of successfully completed opens that have not had a corresponding close. It includes block, character and pass-through driver opens and closes.

lp->state  L_TAGS is the only bit of lp->state that is defined for the interface driver; all other bits are undefined. The interface driver will send a simple queue tag message when initiating an I/O to the logical unit represented by lp if (lp->state & L_TAGS) is set. The (lp->state & L_TAGS) bit will never change when there are active I/Os on the logical unit.

Target Structure  A SCSI target structure is created per SCSI target. This structure is allocated and initialized when a LUN connected to the target is first opened, and is deallocated on the last close of a LUN connected to the target by the SCSI services. This is owned by the SCSI services, but can be accessed by an interface driver.

```c
struct scsi_tgt {
    ubit32 open_cnt;
    ubit32 state;
    ubit8 tgt_id;
    u_char min_sdtr_period;
    struct scsi_bus *bus;
    void *if_tgt;
    struct scsi_lun *lun[SCSI_MAX_LUN_ID+1];
} *tp;
```

```c
#define T_ENABLE_WDTR 0x40
#define T_ENABLE_SDTR 0x20
```

NOTE  The interface driver is not allowed to set the following fields:

- `open_cnt`
- `state`
- `tgt_id`
- `min_sdtr_period`
- `bus`
- `lun`

---

Chapter 8 347
Writing a SCSI Interface Driver

External Interfaces to an Interface Driver

`tp->open_cnt` The number of successfully completed opens that have not had a corresponding close. It includes block, character and pass-through driver opens and closes for all logical units of this target.

`tp->state` Two bits are defined. T_ENABLE_WDTR directs the interface driver to initiate SDTR negotiation after any event other than WDTR negotiation that causes the data transfer width to be reset to eight buts, e.g., bus reset or BDR. If T_ENABLE_WDTR is not set, the interface driver is forbidden from initiating WDTR negotiation, but not from responding. T_ENABLE_SDTR directs the interface driver to initiate SDTR negotiation after any event other than SDTR negotiation that causes the synchronous data transfer parameters to be reset to asynchronous e.g., bus reset, BDR or WDTR negotiation. If T_ENABLE_SDTR is not set, the interface driver is forbidden from initiating SDTR negotiation, but not from responding.

`tp->tgt_id` SCSI ID of this target.

`tp->min_sdtr_period` Lower limit on the transfer period that should be used for synchronous transfers to this target. It may or may not be smaller than what is supported by the hardware. This value may change at any time and the interface driver is responsible for checking it before initiating every I/O to the target. If the currently negotiated transfer period is smaller than what is supported by the hardware. This value may change at any time and the interface driver is responsible for checking it before initiating every I/O to the target. If the currently negotiated transfer period is smaller than `tp->min_sdtr_period` and (`tp->state & T_ENABLE_SDTR`) is set, the interface driver should initiate SDTR negotiation to rectify the situation.

`tp->bus` Pointer to the target's parent `scsi_bus` structure in the open device tree.
Writing a SCSI Interface Driver

External Interfaces to an Interface Driver

**tp->if_tgt**  
Pointer to the data area allocated and bzero'ed on first open, by the SCSI subsystem for use by the interface driver. Its size is specified by the interface driver in `isc->ifsw->if_tgt_size`. This memory is allocated during the first open of a LUN on this target and is freed during the last close of a LUN on the target.

**tp->lun**  
Array of pointers to the `scsi_lun` structures for open logical units of the target. `tp->lun[x]` is `NULL` if and only if logical unit `x` is not open.

**Bus Structure**  
A SCSI bus structure is created per SCSI bus. This structure is allocated and initialized when a LUN connected to a target on the SCSI bus is first opened, and is deallocated on the last close if a LUN connected to the target on the SCSI bus by the SCSI services. This is owned by the SCSI services, but can be accessed by an interface driver.

```c
struct scsi_bus {
    u_int open_cnt;
    struct isc_table_type *isc;
    void *if_bus;
    u_char bus_id;
    scsi_lock_t *lock;
    struct scsi_tgt *tgt[SCSI_MAX_TGT_ID+1];
    struct buf *select_q;
} *busp;
```

**NOTE**  
The interface driver is not allowed to set the following fields:
- `open_cnt`
- `bus_id`
- `lock`
- `isc`

**busp->open_cnt**  
The number of successfully completed opens that have not had a corresponding close. It includes block, character and pass-through driver opens and closes for all logical units of all targets of this bus.

**busp->isc**  
Pointer to the bus' parent `isc_table_type` structure in the open device tree.
busp->if_bus  Pointer to the data area allocated and bzero'ed on first open, by the SCSI subsystem, for use by the interface driver. Its size is specified by the interface driver in isc->ifsw->if_bus_size. This memory is allocated during the first open of a LUN on this bus and is freed during the last close of a LUN on the bus.

busp->bus_id  Index into the scsi_isc array that will yield a pointer to busp->isc.

busp->lock  SCSI bus spinlock. This lock needs to be held by the interface driver when calling scsi_enqueue/scsi_dequeue/scsi_dequeue_bp. scsi_bus_lock() and scsi_bus_unlock() services are used to acquire and release the lock respectively.

busp->tgt  Array of pointers to the scsi_tgt structures for open targets of this bus. busp->tgt[x] is NULL if and only if target x is not open.

busp->select_q  A doubly linked list of buf structures onto which services places I/Os ready for selection. The interface driver picks up I/O requests from this queue. A detailed discussion of this field follows.

**Select_q**  This is the only data structure that is shared between the SCSI subsystem and the interface driver in the sense that both may change it. It is the per-bus select queue.

```c
struct scsi_bus {
    struct buf *select_q;
} *busp;
```

After initializing the bp and its scb and bzero'ing *scb->if_scb, the SCSI subsystem enqueues the bp onto the select queue using scsi_enqueue(&busp->select_q, bp, TAIL) Once the SCSI subsystem enqueues a bp onto the select queue, it will not modify the bp, its scb or the if_scb until the interface driver returns the bp via scsi_cbfn nor will the SCSI subsystem remove a bp from the queue once it has enqueued it.
The interface driver must treat the select queue as ordered for any one logical unit and execute a Request Sense during Contingent Allegiance in response to any Check Condition. That is, the interface driver must initiate all I/Os for any logical unit in the order that they were enqueued by the SCSI subsystem and it must not initiate any other command to the logical unit after a Check Condition until the associated Request Sense clears the Contingent Allegiance.

The interface driver may dequeue from and enqueue to the head of the select queue with `scsi_dequeue(&busp->select_q, HEAD)` and `scsi_enqueue(&busp->select_q, bp, TAIL)` provided it observes the ordering requirement. It may also dequeue a specific bp using `scsi_dequeue_bp(&busp->select_q, bp)` with the same restriction.

The interface driver should not access the queue in any other way than through the access functions provided by the SCSI subsystem as mentioned in this section.

The interface driver needs to hold the SCSI bus lock while calling `scsi_enqueue/scsi_dequeue/scsi_dequeue_bp`, `scsi_bus_lock()` and `scsi_bus_unlock()` services are used to acquire and release the SCSI bus lock respectively.

**Data Structure Diagram** To complete the discussion on the various data structures that are of relevance to an interface driver, a diagram is given below to illustrate the inter-relationship among different data structures in the SCSI subsystem and other kernel data structures.
I/O Path

An I/O request typically passes through different queues in the SCSI subsystem when it passes from one layer to another. The following is a brief description of these queues.

**scb queue**
- per-LUN queue defined in the device driver that contains requests that are waiting for an scb. Requests get queued into this queue by the device driver's `dd_strategy` routine and get dequeued by the PD's `dd_start` routine.

**tag queue**
- per-bus queue that contains requests that are waiting for a qtag. Requests that are ready to be retried (i.e., requests on the retry queues that have hit their “time-to-retry”) are placed on the tag queue to be restarted because they already have an scb (from before) but need a new qtag assignment.

**nexus queue**
- per-LUN queue that contains requests that have to wait to be started so as to not exceed the LUN's queue depth (`lp->max_active`).

**select queue**
- per-bus queue that contains requests that are ready to be handed off to the SCSI HBA Driver.

**retry queue**
- per-bus queue that contains requests that are needed to be retried at some point in the future. The requests are ordered in the queue in “timeout order” (i.e., they're ordered by the time when the request will be retried).

The following figure illustrates all the queues involved in a typical I/O path.
I/O Within SCSI Device Driver

The SCSI device driver is responsible for enqueuing the incoming I/O requests in its queue. Device drivers typically call `scsi_strategy()` of SCSI services for this purpose. The device driver specifies a `dd_strategy` entry point in `scsi_ddsw` structure for SCSI services to call the actual device driver routine that does the queueing.
SCSI services, in addition to calling the device driver strategy routine, provide:

- I/O buffer alignment as per the interface driver requirement
- I/O forwarding, to do further processing on the CPU slated to handle the I/O completion.

The device driver's strategy routine should minimally enqueue the buffer on lp->scb_q. The I/O start time should be recorded in bp->b_qstart if the I/O request is to be timed.

A SCSI device driver owns the following queues:

- lp->scb_q  Device driver I/O queue
- dd_lun->lun_disk_queue  sdisk driver’s queue of sorted I/O requests
- lp->special_scb_q  sctl pass-through driver I/O request queue
- lp->priority_scb_q  Priority mode I/O queue

Device driver strategy routine may return an error if an I/O request encounters any errors while queueing the I/O request. dd_strategy returns non-zero status to indicate an error. If I/O is queued successfully, SCSI services act on the I/O for further processing.

**I/O Within SCSI Service Layer**

The SCSI service layer is responsible for:

- Passing I/O requests from device driver to interface driver.
- Implement flow control policies to honor device I/O queue depth
- Ensure fair distribution of shared bus resources between different luns (tags, scbs).
- Keeping track of I/O time.
- Handle I/O completion.
- Retrying I/O requests if needed.
SCSI services allocate the required resources for the I/O. If tag resources are not available, the buffer is queued to `busp->tag_q`. If nexus resources are not available, the buffer is queued to `lp->nexus_q`. If all the resources are allocated, SCSI services place the buffers to `busp->select_q`. The Interface driver is called through the `ifsw.if_start` entry point for processing all the I/O requests queued on its `select_q`.

Interface drivers return completed I/O requests to SCSI services by calling `scsi_cbfn()`. If I/O is not completed successfully, depending on the I/O return status, a device driver can enqueue the I/O to `busp->retry_q`.

The SCSI services layer owns the following queues:

- `busp->tag_q`: SCB is allocated and I/O is initialized. Waiting for per bus resource.
- `lp->nexus_q`: I/O request got SCB and tag resources; waiting for nexus resource.
- `busp->retry_q`: I/O is being retried on failure. Tag and nexus resources are fixed. Waiting for the timeout period to be queued.
- `busp->select_q`: I/O is enqueued to interface layer. I/O is owned by interface driver until it is returned to SCSI services by calling `scsi_cbfn()`.

**I/O Within SCSI Interface Layer**

Interface driver’s `if_start` routine must dequeue each I/O on its `select_q` and do whatever is required to execute the I/O request. SCSI interface driver is responsible for:

- Executing the I/O request.
- Time the I/O request as requested by upper layers.
- Return I/O to SCSI services by calling `scsi_cbfn()`.
Interface Driver Development

Developing an interface driver includes the following steps:

1. Driver installation and initialization.
2. Register mapping and DMA.
3. Protection and synchronization.
4. Sending SCSI I/O requests to the target.
5. Processing Completed I/O requests.
6. Interrupt Handling.
7. Special Routines.

Driver Installation and Initialization

A SCSI interface driver must have a `driver_install` routine and a `driver_attach` routine. If the interface driver controls and interacts with a hardware device, the driver is also required to have an interrupt service routine to handle the device interrupts. Typically a `driver_init()` routine is assigned to the `isc->gfsw->init` field which will be called by the WSIO.

For a detailed discussion on the `driver_install()`, `driver_attach()`, `driver_init()` and `driver_isr()` routines, refer to Writing a Driver in Chapter 5 of the DDG.

The following sections explain the specific Interface Driver installation and initialization routines.

Calling `driver_install()`

When the HP-UX system is configured through the `config` command, a table of `driver_install()` entry points is created from information in `/stand/system`.

When `driver_install()` is called by the I/O system configuration process through the `driver_install()` entry point configured in the system, the `driver_install()` routine places the `driver_attach()`
entry point in the table of drivers to be called at configuration time. driver_install() then calls the wsio_install_driver() routine to register the driver with the I/O subsystem and returns any error.

**SCSI Specific Part** Every interface driver has to register a probe function with the WSIO to identify the targets connected to the interface card. This is typically the next step to attaching the driver_attach() routine. The SCSI subsystem provides the probe function parallel_scsi_probe to perform the probe and the WSIO provides wsio_register_addr_probe() to register a probe function with the WSIO. The SCSI interface driver sets the “class” field of the drv_info_t to “ext_bus”.

The following example shows a call to driver_install().

```c
static drv_ops_t qlisp_drv_ops = {
    NULL,     /* d_open */
    NULL,     /* d_close */
    NULL,     /* d_strategy */
    NULL,     /* d_dump */
    NULL,     /* d_psize */
    NULL,     /* d_mount */
    NULL,     /* d_read */
    NULL,     /* d_write */
    NULL,     /* d_ioctl */
    NULL,     /* d_select */
    NULL,     /* d_option1 */
    NULL,     /* reserved1 */
    NULL,     /* reserved2 */
    NULL,     /* reserved3 */
    NULL,     /* reserved4 */
    0         /* d_flags */
};
static drv_info_t qlisp_drv_info = {
    "qlisp",  /* name */
    "ext_bus", /* class */
    DRV_SAVE_CONF | DRV_MP_SAFE | DRV_SCAN, /* flags */
    NULL,     /* b_major */
    NULL,     /* c_major */
    NULL,     /* cdio */
    NULL,     /* gio_private */
    NULL,     /* cdio_private */
};
static wsio_drv_data_t qlisp_wsio_data = {
    "scsi",    /* drv_path */
    T_INTERFACE,  /* drv_type */
    
```
Writing a SCSI Interface Driver

Interface Driver Development

Chapter 8

WSIO allows a driver to register its own probe function that is used by the WSIO to scan for devices underneath an interface card. This is usually required if the "ioscan" can’t reach the device in question on its own. What it means is that if a device is behind a bus for which there is
Writing a SCSI Interface Driver

Interface Driver Development

no bus nexus CDIO in the OS, it is the responsibility of the driver to let WSIO inform that it can scan by setting the DRV_SCAN flag in drv_info_t structure and provide a handle to the WSIO by registering a probe function with the WSIO. For a detailed discussion on the driver probe function, refer to Writing a Driver in Chapter 5 of the DDG.

Calling driver_attach()

Use the driver_attach() routine to determine whether the product ID passed in matches the driver_attach device and vendor ID. If the IDs do not match, the driver_attach() routine calls the next attach routine in the chain by calling the *driver_saved_attach() routine.

The driver_attach() routine may be called many times before a match is found. For the device in the first slot, the associated driver_attach() routine is called by the number of devices in the PCI backplane. For the device in the last slot of the PCI backplane, the associated driver_attach() routine is called only once.

When the driver_attach() routine recognizes the device ID, it takes actions such as allocating and initializing its driver control blocks and PCI I/O registers. driver_attach() also sets up driver initialization routine and finally calls isc_claim() to claim the device.

SCSI Specific Part  A SCSI interface driver has additional requirements for hardware that it controls. It must inform the SCSI subsystem of this SCSI hardware by assigning the appropriate pointer in the global scsi_isc array to isc. The index of the appropriate pointer in scsi_isc is the bus ID for the SCSI bus. The index into this array can be obtained by calling the WSIO routine wsio_isc_to_instance(isc, NULL).

Another responsibility of the SCSI interface driver is to attach a fully initialized scsi_ifsw structure to the isc structure via isc->ifsw field. This is typically performed in the interface driver init routine. Each bus could have its own scsi_ifsw if necessary, but it's more likely that a single scsi_ifsw could be shared by all buses being managed by the same interface driver.

The interface driver init routine also sets the initiator ID, SCSI transfer rates and bus width fields in the isc structure by calling SCSI_GET_INITIATOR_PARAMS macro. The bus_max_width field in the isc structure is important. The parallel_scsi_probe() uses this field
to actually scan the SCSI bus. If this field is not set, the SCSI services do not scan on the SCSI bus behind the interface card. More information on this is provided later in the document.

Device initialization, allocating and mapping SCSI command queues, registering device interrupts etc., can be done in the driver_init() routine. In the attach routine,isc->gfsw->init field is assigned with a pointer to the driver_init() routine. This can also be done by calling the macro CONNECT_INIT_ROUTINE(isc, <driver>_init). The HP-UX configuration continues the initialization by calling the driver init routines after calling the driver attach routine.

An example driver_attach() routine and a driver_init() routine are shown below.

```c
#define QLISP_ID_SUPPORTED(x) ((x) == QLISP_ID_1020 || (x) == QLISP_ID_1040 || (x) == QLISP_ID_1240 || (x) == QLISP_ID_1080 || (x) == QLISP_ID_1280 || (x) == QLISP_ID_12160 )

static int qlisp_pci_attach(uint32_t id,
                      struct isc_table_type *isc)
{
  /* Check if it is the correct device */
  if (QLISP_ID_SUPPORTED(id)) {

    /* Check that the PCI register set is correctly
     * mapped */
    if(isc->if_reg_ptr == NULL) {
      msg_printf("qlisp:
        Mapping QLISP registers failed!
      ");
      goto recover1;
    }

    /* Allocate driver control structures */
    /* Initialize the fields in the driver control
     * structures */

    /* Connect the driver init routine */
    CONNECT_INIT_ROUTINE(isc, qlisp_init);

    /* Claim the device */
    isc_claim(isc, &qlisp_wsio_info);

    /* ... */

  }
```
return qlisp_saved_pciAttach(id, isc);

recover1:
    /* Error processing */
}

static int
qlisp_init(struct isc_table_type *isc)
{
    /* Perform SCSI specific steps */

    /* 1. Assign the scsi_isc array with the
    * interface’s isc. SCSI interface drivers must
    * check the instance number returned by
    * wsio_isc_to_instance(). If the returned value
    * is greater than SCSI_MAX_BUS_ID, the driver
    * should return WSIO_ERROR */

    if((instance = wsio_isc_to_instance(isc, NULL)) >
        SCSI_MAX_BUS_ID) return WSIO_ERROR;
    scsi_isc[instance] = isc;

    /* 2. Attach the interface driver switch */

    /* Allocate memory for scsi_ifsw used by SCSI services */

    /* Initialize the ifsw fields */

    /* Attach the struct ifsw to isc */
    isc->ifsw = (caddr_t)qlisp_ifsw;

    /* 3. Call SCSI_GET_INITIATOR_PARAMS macro to get the
    * initiator ID and the SCSI rate. */

    /* Set the following fields of the isc with the correct
    * values: isc->my_address
    *    isc->bus_max_width
    *    isc->bus_min_sdtr_period
    *    isc->bus_max_reqack_offset
    */

    /* Configure PCI config space on the interface card */
/* Allocate and map memory for SCSI command queues */

/* Reset the interface card */

/* Allocate interrupt objects and activate the interrupt */

return 0;
}

Setting SCSI Parameters

Some of the state information about the SCSI bus and targets on the bus must persist across opens, but the data structures are deallocated when a device is closed. So this information is kept in the following fields of the isc structure:

- isc->my_address
- isc->bus_max_width
- isc->bus_min_sdtr_period
- isc->bus_max_reqack_offset

For additional details on these fields, please refer to the discussion on the isc structure earlier in this document.

The driver has to set these fields in the driver’s init routine. In the current implementation of the SCSI subsystem, it is required to interact with the processor dependent code on PA-RISC platforms to obtain the SCSI parameter values. However, this may change in future platforms. In order to protect drivers from these changes, the DDK includes SCSI_GET_INITIATOR_PARAMS macro. It wraps the processor dependent calls and passes back the values of SCSI parameters. For the correct operation of the macro, it is strongly suggested not to modify anything in the macro.

NOTE

Include the file scsi_params_macro.h that is distributed with the DDK to access this macro.
SCSI_GET_INITIATOR_PARAMS This macro is called with four arguments; a pointer to the isc structure, and place holders for the initiator ID, SDTR period and the REQ/ACK offset. The last three parameters to the macro correspond to the following fields in the isc.

my_address
bus_min_sdtr_period
bus_max_reqack_offset

NOTE

SCSI_GET_INITIATOR_PARAMS does not set the fields in the isc structure. It just returns the values of the corresponding fields.

The following pseudo code of an interface driver's init routine illustrates the calling convention of SCSI_GET_INITIATOR_PARAMS macro and how to set the fields in the isc.

```c
int driver_init()
{
    /* Declare local variables to pass to the macro */
    int initiator_id, sdtr_period, reqack;
    ...

    /* Call the macro to obtain the SCSI parameters */
    SCSI_GET_INITIATOR_PARAMS(isc, initiator_id,
                                sdtr_period, reqack);

    /* Set the corresponding fields in the isc with the */
    /* values obtained above. */
    isc->my_address = initiator_id;
    isc->bus_min_sdtr_period = sdtr_period;
    isc->reqack_offset = reqack;

    /* Set the SCSI bus width */
    isc->bus_max_width = width; /* where width equals */
                          /* to 8 or 16 */
    ....
}
```
SDTR/WDTR Negotiation

SCSI host bus adapters and targets negotiate for synchronous data transfer rates and wide data transfer widths. Synchronous Data Transfer Request (SDTR) and Wide Data Transfer Request (WDTR) commands are used for this. Typically an interface driver need not do anything for this. However in some scenarios like bus reset or when a slower rate device is connected on the bus, the interface driver may force a negotiation on SDTR and WDTR.

Current Negotiated Information  The currently negotiated synchronous data transfer period for each target is stored in one byte of the tgt_sdtr_period array. A pointer to the byte associated with a particular target is kept in the scsi_tgt structure while the device is open so it can be accessed easily and in a way consistent with other target-specific state information. A value of 0xff implies asynchronous data transfers, otherwise the value is the minimum transfer period from the SDTR agreement.

The currently negotiated bus width for each target is stored in one bit of the tgt_wdtr_width bit field. As with tgt_sdtr_period, a pointer to the bit field is kept in the scsi_tgt structure. If the bit is set, 16-bit wide transfers have been agreed upon.

Two other bit fields in the isc_table_type structure associated with SDTR and WDTR are the tgt_sdtr_done and the tgt_wdtr_done bit fields. There is one bit per target in each indicating whether or not SDTR or WDTR negotiation has been done since the most recently recognized event which reset the device parameters to their default value.

When to Negotiate  The interface driver clears the appropriate tgt_sdtr_done and tgt_wdtr_done bits, sets the appropriate tgt_sdtr_period bytes to 0xff and clears the appropriate tgt_wdtr_width bits in the isc_table_type structure when it recognizes a device reset. The interface driver recognizes a device reset when it initiates or detects a SCSI bus reset or sends a Bus Device Reset (BDR) message to the device. The interface driver will also clear the appropriate tgt_sdtr_done bit when WDTR is negotiated. The interface driver will always initiate SDTR or WDTR if the corresponding “done” bit in the isc_table_type structure is clear and the T_ENABLE_SDTR or T_ENABLE_WDTR bit is set in tp->state. It will also initiate WDTR or SDTR whenever SCB_WDTR or SCB_SDTR is set in the scb independently of
the bits in `tp->state`. It will initiate WDTR and/or SDTR negotiation on every auto-sense if enabled by `tp->state`. The interface driver may also initiate negotiation at any other time it deems appropriate.

**What to Negotiate**  Three parameters are used in determining the minimum transfer period for which to negotiate:

- `tp->min_sdtr_period`
- `isc->bus_min_sdtr_period`
- the lowest period at which the hardware can operate

When initiating SDTR, the interface driver will send a period equal to the maximum of `tp->min_sdtr_period`, and the minimum period supported by the hardware. The interface driver will always negotiate for the maximum REQ/ACK offset allowed by the hardware.

Unlike for SDTR, there is no way to control the bus width for which to negotiate. It is all or nothing. Either negotiation is enabled or not. If negotiation is initiated, it will be for the maximum supported by the bus.

**Register Mapping and DMA**

A brief discussion on PCI register mapping is provided in this section. Refer to *Writing PCI Device Drivers* in Chapter 10 of the DDG for a detailed discussion on PCI register mapping services.

For WSIO drivers, the `if_reg_ptr` member of the `isc` structure is a PA virtual address corresponding to a base address register that has already had this virtual mapping done to make it usable by the driver and system. If `if_reg_ptr` is NULL, the driver needs to map the range itself.

WSIO provides a number of services for DMA mapping. Refer to the man pages on WSIO services in the DDR for a detailed discussion of each of these services. In an interface driver, `dma_setup()` or `bp_dma_setup()` is called to setup DMA mapping, depending on whether one `buf` structure or multiple `buf` structures merged together.
Fields relevant only to an interface driver:

```c
struct dma_parms {
    int channel;
    int dma_options;
    int flags;
    struct iovec *chain_ptr;
    int chain_count;
    caddr_t addr;
    space_t spaddr;
    int count;
};
```

There is a derived structure from `dma_parms`.

```c
struct bp_dma_parms {
    struct dma_parms dma_parms;
    struct dma_parms *merge_dma_parms;
};
```

The following steps set up the DMA.

1. Initialize the DMA options.
   - If the device can bus master for DMA, the channel field has to be set to `BUS_MASTER_DMA`. `dma_parms->channel = BUS_MASTER_DMA`.
   - If you don’t want to wait until the `dma_setup()` succeeds, `dma_parms->flags = NO_WAIT`.
   - Set the DMA options. `dma_parms->dma_options = ((bp->b_flags & B_READ) ? (DMA_8BYTE|DMA_READ) : (DMA_8BYTE|DMA_WRITE))`

2. If `bp->bp_merge_cnt equal 0`, call `dma_setup()`.
   - Set the address of the buffer to be mapped. `dma_parms->addr`.
   - Set the space address of the buffer to be mapped. `dma_parms->spaddr = bp->b_spaddr`. 
Set the buffer length to be mapped.

\[ \text{dma_params->count} = \text{bp->b_count}. \]

\[ \text{ret_code = dma_setup(isc, dma_params);} \]

On a successful call to \text{dma_setup()}, \text{chain_ptr} in the \text{dma_params} structure points to an \text{iovec} structure that contains the mapped address and the length of the mapping. If there are multiple elements involved, \text{chain_count} field can be used to obtain the number of DMA elements.

3. If \text{bp->b_merge_cnt} > 0, call \text{bp_dma_setup()}. Note that \text{bp_dma_setup()} takes \text{bp_dma_parms} as one argument, not \text{dma_params}.

\[ \text{ret_code = bp_dma_setup(isc, bp, bp_dma_parms);} \]

**Protection and Synchronization**

The major synchronization issue with interface 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. Spinlock scheme may be chosen to provide finer granularity locks, protecting data structures at finer levels. More information on spinlocks is available in *Using Spinlocks and Semaphores for Synchronization in MP Systems* in Chapter 4 of the DDG.

Whenever an interface driver enqueues or dequeues bp's from the \text{select_q}, it has to acquire \text{scsi_bus_lock()} and release the lock after the completion of \text{scsi_enqueue()}, \text{scsi_dequeue()} or \text{scsi_dequeue_bp()} by calling \text{scsi_bus_unlock()}.

For example:

\[
\begin{align*}
\text{scsi_bus_lock(busp);} \\
\text{scsi_enqueue(&busp->select_q, bp, HEAD);} \\
\text{scsi_bus_unlock(busp);} \\
\end{align*}
\]
Similarly,

```c
scsi_bus_lock(busp);
bp = scsi_dequeue(&busp->select_q, HEAD);
scsi_bus_unlock(busp);
```

It is advisable to protect driver control structures with a spinlock to protect across MP access.

A lock order of \((\text{SCSI\_LOCK\_ORDER\_BASE} + 2)\) can be used by the interface driver while allocating new spinlocks.

For example:

```c
typedef struct {
    uint32_t    state;
    ubit8       chip_rev;
    ....
    lock_t      *mp_lock;
    ....
} qlisp_shared_isc_t;
```

It is highly recommended that the driver does not hold spinlocks across calls to different subsystems. So, before calling the SCSI services, the driver should release the spinlocks and acquire them if necessary when the interface driver gets the control back.
Sending SCSI I/O Requests to the Target

The SCSI subsystem requires the interface driver to specify a start function. This is the entry point for an I/O request at the interface driver. The following pseudo code is for a typical interface driver start routine.

This routine is called with a pointer to the isc structure. Obtain the scsi_bus structure from the isc.

(NOTE: isc-if_drv_data field points to the scsi_bus structure)

do

acquire scsi_bus lock

Get a bp from the select_q

release scsi_bus lock

if bp is not NULL

get SCSI Control Block from the bp.

if there are bytes to transfer
(bp->b_bcount != 0)

DMA map the buffer(s)

Get the hardware I/O control block(s)

If the I/O request can’t be satisfied because of insufficient resources at the hardware, set the scb->cdb_status to SCTL_INVALID_REQUEST and call scsi_cbfn().

If the I/O request can’t be satisfied because of temporary resource shortage, then put the bp back at the HEAD of the select_q.

Build the SCSI command

Inform the hardware of a pending I/O request

done
done

until there are bp’s in the select_q

Processing Completed I/O Requests

I/O requests which are completed are processed when the I/O card either generates an interrupt or informs by some other means.

The interface driver has to inform the SCSI subsystem that an I/O request is completed so the resources allocated for that I/O can be freed. The SCSI subsystem provides the interface driver with the service `scsi_cbfn()` to accomplish this. However before calling the SCSI callback services, fields in the SCB structure have to be set to indicate the result of an I/O request. An I/O might have completed successfully, or it might have resulted in a Check Condition and an auto-sense message is returned. An I/O might have failed because of lack of resources, an invalid command, or an I/O might have timed out. In any of these scenarios, the SCSI subsystem should know the result of the I/O for further processing. It is the responsibility of an interface driver to set the fields in the SCB structure to provide the correct result to the SCSI subsystem.

For a detailed discussion on the fields of the SCB structure, refer to “SCSI Control Block” on page 330. The following pseudo code defines the steps required to set up the fields of the interface driver. In the examples, `scb` is `bp->scb`.

1. If the I/O request is completed successfully, set:

   ```c
   scb->sense_bytes = 0;
   scb->data_resid = 0;
   scb->sense_status = SCTL_INCOMPLETE
   scb->cdb_status = S_GOOD;
   ```

2. If there is a select timeout, set:

   ```c
   scb->sense_status = scb->cdb_status = SCTL_SELECT_TIMEOUT;
   scb->data_resid = <value returned from the device.>
   ```

   If this generates an auto-sense, follow step 3 to set other fields. Otherwise, set:

   ```c
   scb->sense_status = SCTL_INCOMPLETE:
   scb->sense_bytes = 0;
   ```
3. If the I/O request resulted in an auto-sense, set:

```c
scb->sense_status = S_GOOD;
scb->sense_bytes = (length of the driver buffer used for sense)
scb->sense_data = (pointer to the driver buffer used for sense)
scb->data_resid = (If data transferred and the sense data has at least
the info byte (resid) field, check resid count from
there. Else set the field with the value returned
from the device.)
```

Code example:

```c
....

scb->data_resid = iocb->stat.residual;

if (iocb->stat.state_flags & QLISP_STATE_GOTSense) {
    scb->sense_status = S_GOOD;
    scb->sense_bytes = iocb->stat.req_sense_length;
    scb->sense_data = iocb->stat.req_sense_data;
    /*
    ** If data transferred and the sense data has at
    ** least the info byte (resid) field, check resid
    ** count from there. Some devices report NO_SENSE
    ** check condition with the resid count in the
    ** sense data and the QLogic card reports zero
    ** resid bytes. Need to update scb->data_resid
    ** from there so physio() truncates the data
    ** correctly.
    */
    if (bp->b_bcount &&
        (scb->sense_bytes >= QLISP_MIN_REQ_SENSE_LEN)) {
        struct sense_2_aligned *sense = (void*)
            scb->sense_data;
        int resid;

        if ((sense->error_code == S_CURRENT_ERROR) &&
            (sense->info_valid) &&
            (scb->data_resid == 0)) {
            resid = (sense->info[0] << 24)
                + (sense->info[1] << 16)
                + (sense->info[2] << 8)
```
+ sense->info[3];

/* Negative resid values equate to a
** record size read smaller than what’s
** actually returned by the device. Set resid
** count to zero.
*/
scb->data_resid = (resid > 0) ? (uint32_t)
    resid : 0;
}
}

....

4. If SCSI command transport failed completely, set:

    /* Transport failed entirely, residual = requested length */
    scb->data_resid = bp->b_bcount;
    scb->cdb_status = SCTL_INCOMPLETE;
    scb->sense_status = SCTL_INCOMPLETE;
    scb->sense_bytes = 0;

5. If the I/O request is invalid, set:

    scb->cdb_status = S_BUSY;

6. If a call to dma_setup() / bp_dma_setup() fails, set:

    scb->cdb_status = S_BUSY;

When the control returns to the interface driver after calling
scsi_cbfn(), the interface driver should not touch any of the resources
which are part of the completed I/O. Typically, an interface driver forms a
chain of bp’s whose I/O has completed. In order to build such a chain, the
interface driver can use av_forw field of the buf structure. The following
code example clarifies this point.

    bp->av_forw = bp_chain;
    bp_chain = bp;

If initially bp_chain points to NULL, as bp’s become available the
buf_chain points to the most recent bp and the chain can be traversed
through the bp->av_forw field.

When calling scsi_cbfn(), bp’s from this list are passed to the SCSI
service.
Interrupt Processing

Writing a Driver in Chapter 5 of the DDG discusses the issues involved in writing an interrupt service routine (ISR). An ISR is a device specific routine and different drivers do different things in handling a device interrupt. Some generic steps are provided for processing a device interrupt.

1. Acquire a spinlock for protection on MP-systems
2. Read a device status register.
3. Check if it is a spurious interrupt. If so, release the lock and return.
4. If an I/O request is completed without any errors, process it.
5. If there is an error, look for the specific event that caused the interrupt.
6. Build a bp chain of all the completed I/Os.
7. Set the device register to indicate that an I/O response is processed.
8. Loop if there are some more completed commands.
9. Wakeup any I/O requests that are waiting for device resources.
10. Release the spinlock.
11. Call SCSI callback services on the I/Os that are completed.
12. Return INTR_Sserviced

Special Routines

A SCSI interface driver can optionally implement the following routines to handle special events such as bus reset, I/O abort, etc. A brief description and pseudo code is provided here for each of the routines.

Bus Device Reset

The SCSI subsystem calls the interface driver's bus device reset (bdr) function in response to SIOC_RESET_DEV ioctl request.

This function is passed with a buf structure bp as its sole argument. bp->b_dev field contains the device number of the target. As mentioned earlier, the SCSI subsystem provides m_* services to obtain required SCSI data structures to execute a BDR.
Pseudo code for a typical BDR routine:

1. Obtain device number from `bp->b_dev`.
2. Obtain `scsi_bus` structure from `m_scsi_bus(bp->b_dev)`.
3. Obtain `scsi_tgt` structure from `m_scsi_tgt(bp->b_dev)`.
4. Obtain `isc` structure from the `bp->isc`.
5. If the device is in reset state, reject the request.
6. Get the driver control structures.
7. Issue a BDR command to the device.
8. Set the device to a known state.

Aborting an I/O Request

The SCSI subsystem calls the interface driver’s abort function in response to `SIOC_ABORT` ioctl request. This function is passed with a `buf` structure `bp` as its sole argument. `bp->b_dev` field contains the device number of the destination I/O. The SCSI subsystem provides `m_*` services to obtain required SCSI data structures to execute an abort of the specified I/O request.

Pseudo code for a typical abort routine:

1. Obtain device number from `bp->bp_dev`.
2. Obtain `scsi_bus` structure from `m_scsi_bus(bp->b_dev)`.
3. Obtain `scsi_tgt` structure from `m_scsi_tgt(bp->b_dev)`.
4. Obtain `scsi_lun` structure from `m_scsi_lun(bp->b_dev)`.
5. Obtain `isc` structure from the `bp->isc`.
6. If the device is in reset state, reject the request.
7. Get the driver control structures.
8. Issue a command to abort the LUN device.
9. Set the device to a known state.

Reset Bus

The SCSI subsystem calls the interface driver’s `reset_bus` function to reset the SCSI bus. This function is passed with an `isc` structure as its sole argument.
Pseudo code for a typical reset_bus routine:

1. Get the driver control structures.
2. Acquire a driver spinlock.
3. Check if the device is under reset state. If so, release the spinlock and return. Otherwise, set the device state to RESET state.
4. Release the spinlock.
5. Issue a command to the device to reset SCSI bus.
6. Acquire a driver spinlock.
7. Set the device state to RESET_DELAY state.
8. Set a time-out after 3 seconds.
9. Release the driver lock.
10. After the time-out pops, reset the driver state.
11. Call the driver’s if_start() function to process I/O requests.

Handling Timeouts

It is a common practice in driver development to use timeouts to schedule events after a pre-determined time interval. The timeout() kernel service can be used to schedule an event to occur after a specified amount of time. An untimeout() kernel service is also provided to unschedule a timeout. It is particularly important to untimeout() any pending timeouts before unloading a DLKM driver. Refer to Kernel Services in Chapter 2 of the DDR for more information on timeout and untimeout services.
Sleep and Wakeup Mechanism

At times, an interface driver may need to wait for an event like availability of resources or until the device establishes a known state. An interface driver typically calls “sleep” if it can from the current context instead of busywaiting and monopolizing the CPU. When a corresponding event occurs, the driver calls a “wakeup” to notify any sleeping threads on this event.

To avoid any racing conditions, the kernel `spinlock()` services provide `get_sleep_lock()` call. A `get_sleep_lock()` obtains a lock before calling `sleep` and releases after it calls `sleep`. Similarly, a `wakeup()` call is also protected with the locks obtained from `get_spin_lock()`. For a more detailed explanation of this refer to Multiprocessing in Chapter 4 of the DDG.

Example code for a typical sleep/wakeup pair routine:

```c
/*
 * qlisp_sleep -
 * Sleep a process on an address
 */
STATIC void
qlisp_sleep(lock_t *wlock, caddr_t wchan)
{
    VASSERT(owns_spinlock(wlock));

    (void) get_sleep_lock(wchan);
    spinunlock(wlock); /* wlock is a driver spinlock */
    (void) sleep(wchan, PRIBIO);
    spinlock(wlock); /* wlock is a driver spinlock */
}

/*
 * qlisp_wakeup -
 * Wakeup processes sleeping on an address
 */
STATIC int
qlisp_wakeup(lock_t *wlock, caddr_t wchan)
{
    VASSERT(owns_spinlock(wlock)); /* wlock is a driver spinlock */

    return wakeup(wchan);
}
```
Sample Driver

A sample SCSI interface driver for Qlogic's ISP12160 is provided along with this document. A review of the earlier sections of this chapter will ensure a better understanding of this driver.

Description of the Device

A brief description of the device and its functionality is included to keep the driver in its correct perspective. However, this is not a complete description of the device. For that, refer to Qlogic's hardware reference manual.

The ISP12160A supports dual channel, Ultra3 (Fast-80) SCSI functionality. It interfaces the PCI bus to two Ultra3 SCSI buses and contains an on-board RISC processor. The interface between the ISP12160A firmware and drivers consists of two queues; request and response. The queues are located in host memory and are organized as circular fixed-length lists of 64-byte entries (FIFOs). They are I/O mapped and their physical addresses are stored in the device registers.

This section presents the code flow of the sample Qlogic driver. Refer to the sample driver code for details. This section is organized as follows:

- Driver Architecture
- Driver Data Structure
- qlisp_multi driver
  - Driver Installation and Initialization Routines
  - Driver entry points to SCSI Subsystem
  - The main I/O path
  - Device Access and Initialization Routines
  - Interrupt Processing Routines
  - Sending an I/O Request to the target.
  - Processing I/O responses.
  - Miscellaneous routines
Driver Architecture

The sample driver is two drivers; one for claiming the PCI interface card and probing the SCSI channels underneath the PCI card, and another for claiming the SCSI controllers. The first one is “qlisp_multi” and the second driver is “qlisp”. It is very important to understand that this kind of two-driver architecture is required only for special devices with multiple SCSI channels behind a PCI function. In such cases, since the HP-UX ioscan can’t recognize the SCSI controllers that are underneath a PCI interface, two drivers are required.

The qlisp_multi driver registers a driver specific probe function to scan for SCSI controllers during the driver installation. After the driver claims the card, it sets an WSIO_IDENTIFY_CHILD routine to correctly identify the child node. When the driver’s scan finds a SCSI controller, the second driver’s attach routine claims the SCSI channels. Now, if an interface card has each SCSI channel on a separate PCI function, this kind of two driver approach is not required.

As far as each driver’s functionality is concerned, the qlisp_multi driver is involved only in building the I/O tree with the SCSI controllers. The real work is done by the qlisp driver. If a two driver approach is not needed, qlisp driver is the one which is of interest.

Data Structures

The following are the main data structures used in the driver.

qlisp_isc_t Local interface specific control structure (lisc). This is allocated per device instance.

qlisp_shared_isc_t
Common device control structure. This is allocated when the master port is initialized.

qlisp_scb_t Interface-specific SCSI Control Block structure. This is used to keep track of the associated bp and scb.

qlisp_bus_t Interface-specific SCSI bus structure. This is used to locally associate an I/O (based on the tag) with an I/O request.
qlisp_q_ent_t This is a union of different 64-byte structures which can be entries in the request and response queues. These entries can be: Command, Command A64, Extended Command, Continuation, Continuation A64, Marker, Status and Extended Status.

The driver design facilitates traversing from one data structure to another. The inter-relationship among data structures is shown here.

Figure 8-4 Driver Data Structure Inter-relationship

qlisp_multi driver

This section briefly describes the qlisp_multi driver. The objective is to present the code flow of the qlisp_multi driver.

The qlisp_multi driver is a typical WSIO interface driver. The installation and initialization routines of the driver of the qlisp_multi driver are similar to the routines explained in Writing a Driver in Chapter 5 of the DDG. However there are some differences that are specific to the functionality of this driver and they are explained below.

The qlisp_multi driver claims the interface card and scans for SCSI controller chips underneath it. So the driver should be able scan the hardware. This requires three things:

1. It has to inform the WSIO that it can scan for the hardware.
2. It has to provide a device specific probe function to the WSIO to scan the hardware.
3. It has to inform the WSIO of a child driver identifying routine.

The last two steps are typically performed in the qlisp_multi_install() and qlisp_multi_attach() routines.

To inform the WSIO that the driver can scan, the DRV_SCAN flag is set in the drv_info_t structure.

static drv_info_t qlisp_multi_drv_info = {
    "qlisp_multi",      /* name */
    "qlispba",             /* class */
    DRV_SAVE_CONF | DRV_MP_SAFE | DRV_SCAN, /* flags */
    NULL,                  /* b_major */
    NULL,                  /* c_major */
    NULL,                  /* cdio */
    NULL,                  /* gio_private */
    NULL,                  /* cdio_private */
};

Snippets from the qlisp_multi driver’s install and attach routines are shown here to illustrate the driver steps.

int qlisp_multi_install()
{
    ....
    wsio_register_dev_probe(DRV_NAME, &qlisp_multi_dev_probe, "qlisp_multi");
    wsio_install_driver(&qlisp_multi_wsio_info);
    ....
}

The calling semantics for the device probe routine are:

qlisp_multi_dev_probe(this node,
    drv_info,
    probe_id,
    hw_path, isc,
    probe_type,
    name, desc);
In the driver attach routine `qlisp_multi_pci_attach`,

```c
int qlisp_multi_pci_attach()
{
    ....
    isc_claim(isc, &qlisp_multi_wsio_info);
    wsio_set_parm(isc, WSIO_IDENTIFY_CHILD,
                  (void *)&qlisp_multi_identify_child);
    ....
}
```

The calling semantics for the driver identify child routine is:

`qlisp_multi_identify_child(child_name);`

The driver's probe function is called by the WSIO during the hardware scan, and if a device is found the WSIO calls the driver's child identify routine to verify the driver name.

These routines are required only for a multi-port interface card. A detailed discussion on the device probe function and the `WSIO_CHILD_IDENTIFY` routines follows.

**qlisp_multi_dev_probe**

To understand a device probe routine, it is necessary to have an understanding of the hardware path.

A hardware path is a numerical string of hardware components, notated sequentially from the bus address to the device address. Typically, the initial number is appended by slash (/), to represent a bus converter (if required by your machine), and subsequent numbers are separated by periods (.). Each number represents the location of a hardware component on the path to the device. The hardware path is defined by the following structure:

```c
/* HW path structure (hw_path(GIO4) - This structure is used
 * to define a hardware path. The first/last indices define a
 * window of address elements which are meaningful. If
 * last_index == (first_index-1) then the path is NULL.
 * /
#define MAX_ELEMENTS 14
typedef struct hw_path {
    char first_index;
}
```
The device probe routine is called with the following arguments:

- a pointer to the calling I/O tree node
- a pointer to the driver info structure
- probe_id which is set by the interface driver
- hardware path of the calling device
- a pointer to the isc structure
- probe_type
- a pointer to name of the child node, and
- a pointer to description of the child node

When this routine is called for the first time by the WSIO, a probe type of PROBE_FIRST is passed and the hardware path structure has the fields set to satisfy the following condition:

\[ \text{hw_path->first_index} == \text{hw_path->last_index} + 1 \]

That way when a device is found, the driver increments the last_index and it becomes equal to the first_index and the probe function places the hardware address of first device found in the \text{hw_path->addr array} at an index of last_index. Later the probe function is called with the probe type of PROBE_NEXT and with first_index == last_index; both of which point to the hardware address of the last found device. When probe type is PROBE_ADDRESS, the probe routine should retrieve the last element of the hardware path and probe the device at that address. The last element is referenced by last_index.

If a device is found, three pieces of information are retained from the device: an “ID” (the product and vendor ID strings), a “description” (driver description) and a “name” (the expected interface/device path to this device - e.g., “scsi_disk”). The WSIO CDIO will use the “name” property to recognize a corresponding driver and match the device when the node is later claimed.

The driver should know when the address range goes out of range and should return an error in such a case.

If a device is found, the routine returns PROBE_SUCCESS or PROBE_UNSUCCESSFUL.
The following is a sample device probe routine which illustrates the concepts.

```c
static int
qlisp_multi_dev_probe(
    void *this_node,
    drv_info_t *drv_info,
    void *probe_id,
    hw_path_t *hw_path,
    struct isc_table_type *isc,
    int probe_type,
    char *name,
    char *desc)
{
    int found = NO_DEV;
    int target;

    do {
        switch (probe_type) {
            case PROBE_FIRST:
                target = 0;
                probe_type = PROBE_NEXT;
                hw_path->last_index++;
                break;
            case PROBE_NEXT:
                target = hw_path->addr[hw_path->last_index] + 1;
                break;
            case PROBE_ADDRESS: target = hw_path->addr[
                hw_path->last_index];
                break;
            default:
                found = INVAL_TGT;
        }

        if (((target < 0) || (target >= QLISP_MULTI_NUM_PORTS)) ||
            (hw_path->last_index != hw_path->first_index))
        {
            found = INVAL_TGT;
        }

        if (found == NO_DEV) {
            strcpy(name, QLISP_NAME);
            strcpy(desc, QLISP_DESC_BUS);
            *(int *)probe_id = isc->if_id;
            found = VALID_TGT;
        }
    }

    return found;
}
```
Writing a SCSI Interface Driver

Sample Driver

```c
hw_path->addr[hw_path->last_index] = target;
}
} while ((probe_type != PROBE_ADDRESS) && (found == NO_DEV));
return ((found == VALID_TGT) ? PROBE_SUCCESS :
        PROBE_UNSUCCESSFUL);
```

**qlisp driver**

This section briefly describes the qlisp driver. The objective is to present the code flow of the qlisp driver. * qlisp_multi driver is a typical WSIO interface driver. The installation and initialization routines of the driver of the qlisp_multi driver are similar to the routines explained in *Writing a Driver*, Chapter 5 of the DDG. However, qlisp driver registers an address probe function to identify all the LUNs connected to the port.

The driver should be able to scan the hardware. This requires two things:

1. It has to inform the WSIO that it can scan for the hardware.
2. It has to provide a device specific probe function to the WSIO to scan the hardware.

To inform the WSIO that the driver can scan, DRV_SCAN flag is set in the `drv_info_t` structure

```c
static drv_info_t qlisp_drv_info = {
    "qlisp",              /* name */
    "ext_bus",            /* class */
    DRV_SAVE_CONF | DRV_MP_SAFE | DRV_SCAN, /* flags */
    NULL,                 /* b_major */
    NULL,                 /* c_major */
    NULL,                 /* cdio */
    NULL,                 /* gio_private */
    NULL,                 /* cdio_private */
};
```

Snippets from the qlisp driver's install routine illustrate the previously mentioned steps in the driver.
Writing a SCSI Interface Driver

Sample Driver

```c
int qlisp_install(void)
{
 ....

    wsio_register_addr_probe(parallel_scsi_probe, "qlisp");

    wsio_install_driver(&qlisp_wsio_info);

 ....
}
```

`parallel_scsi_probe` is the address probe function for the parallel SCSI drivers. It determines the next address to be probed. Registering this probe function is required to determine the targets and LUNs attached to each port. An explanation of this probe function is given in *Writing a Driver* in Chapter 5 of the DDG.

All routines that are internal to the qlisp driver are given a brief description and for some, pseudo code is provided to facilitate easy understanding of the driver code.

**qlisp_init_pci_cfg**

Turn on the flags in the config space control registers that we care about.

**qlisp_init_reset_card**

- Soft reset the card
- Write into some registers
- Set `SHARE_DMA64` flag

**qlisp_init_attributes**

Set DMA attributes like

- pre-fetch length (in this case, 0)
- DMA width
- Coherency protocol etc.

**qlisp_init_alloc_queues**

This allocates and maps memory for request & response queues.
NOTE

Below, “share” is a pointer to `qlisp_shared_isc_t` data structure.

The “iova’s” are stored in: `share->dma_req_queue_p` &
`share->dma_res_queue_p. share->req_max_iocb is set.

qlisp_init interrupts

This allocates an interrupt object and activates the interrupt. It also
enables device interrupts by writing into the corresponding device
registers

qlisp_init_firmware

- Load the firmware in case of PA-RISC
- Set firmware features.

qlisp_init_common

- Save the isc in `scsi_isc()` maintained by the system.
- Call `SCSI_GET_INITIATOR_PARAMS` macro to obtain SCSI parameter
  values.
- Set the initiator ID.
- Set the correct transfer period and req/ack offset.
- Call `init_ifsw()`

qlisp_init_ifsw

- Allocate memory for `scsi_ifsw` structure used by `scsi_ctl`
- Initialize fields in the `if_sw` structure
- Assign `isc->ifsw` with a pointer to the `ifsw` structure allocated

qlisp_init_start_queues

- Initialize the request and response queues. These queues hold entries of
  I/O Control Blocks (iocb)
- Change the card state to `(SHARE_RSP_QUEUE | SHARE_REQ_QUEUE)`

qlisp_init_master

Call routines `qlisp_init_pci_cfg` through `qlisp_init_start_queues`
in the order set in this section of the manual.
Writing a SCSI Interface Driver

Sample Driver

qlisp_init_slave
- Save slave_isc in master_lisc
- Call qlisp_init_common()

qlisp_pci_attach
- Allocate memory for interface specific control structure
- Assign isc->if_isc with the driver control structure (lisp) allocated
- Compare the device ID to verify whether the card can be claimed or not
- Allocate memory for the shared control structure for a dual port card
- Initialize PCI config space values
- Attach the init routines to the list of init routines
- Claim the card instance

qlisp_mailbox_cmd
Mailbox commands are not done in performance path, and usually only during the driver initialization time. Send a mailbox command. (details later).

qlisp_build_cmd (and cmd 64 for __LP64__)
- Set the iocb fields
- Get the scsi_lun structure associated with the scb
- Get the scsi_tgt structure associated with the lp
- Set the target_id and lun_id fields of the command
- Set the control flags of the command
- Copy the cdb from scb into iocb
- Take the DMA parameters and sync the memory

qlisp_iocb_cnt
Calculate number of DMA command entries from DMA scatter/gather list.

qlisp_get_req_iocb
- Checks request queue for enough available iocb's for command
- Updates share->req_free and share->req_in
qlisp_if_start

Interface entry point called by SCSI services where I/Os are started. This entry point is set in the ID ifsw structure in qlisp_init_ifsw().

- Get the ID specific control structure from isc->if_isc
- Get scsi_bus structure from isc->if_drv_data (this field is set by the scsi_services)
- Get the shared control structure from the lisc

```c
do {
    Acquire scsi bus lock.

    NOTE: Once an I/O has been put on the select queue it is off limits to services until the interface driver calls scsi_cbfn with the I/O.

    Get a bp waiting for interface driver (Call scsi_dequeue, a SCSI service to get a bp from the select queue.

    Release scsi bus lock

    if bp is not NULL
        Get SCSI control block.
        Get a pointer to driver's local scb structure.
        Get a pointer to driver's local bus structure.

        if there are bytes to transfer (bp->b_bcount != 0)
            Perform mapping for dma pages
            Call get_iocb_cnt (cnt64 for LP64)
            Get required iocbs (this may not always succeed)
            if didn't get reqd iocbs, then
                if the request can't be satisfied,
                    Set the scb status to invalid
                    (scb->cdb_status = SCTL_INVALID_REQUEST)
                    and call scsi_cbfn() (SCSI service)
            else (if request can be satisfied later)
                Requeue the bp (scsi_enqueue(&busp->select_q, bp, HEAD))

        /* everything is fine */
        Call build cmd (cmd64 for LP64)
        Increase the activecnt in the interface driver local bus structure.
        /* scb->tag is used as an index into the NexusTable. The
```
Writing a SCSI Interface Driver
Sample Driver

```c
scb->tag is assigned immediately following scb association in scsi_start and is retained for all retries. */
Save the interface driver local scb in interface driver local bus structure.
Inform the device by moving in the req_in pointer.
} until bp != NULL
```

**qlisp_fast_complete**

---

**NOTE**

Used by performance status completion mechanism when status is good and only handle is returned.

---

Get local scb from the local bus structure’s NexusTable field.
Get bp from local scb.
Get scb from local scb.

if bp->b_count != 0, cleanup the DMA

The next few fields indicate the result of an I/O attempt. The appropriate fields are set by the interface driver prior to calling scsi_cbfn().
Set the following fields (as reqd.)
scb->cdb_status
scb->data_resid - number of data bytes that were transferred in response to scb->cdb, i.e.
number of bytes transferred = bp->b_bcount - scb->data_resid
scb->sense_status
scb->sense_bytes
scb->sense_data

Decrement activecnt of the outstanding I/O requests
Set the local scb structure in the Nexustable to NULL
Put the bp in the free list (bp->av_forw)
Assign the bp to the list of bps waiting for scsi_cbfn.
qlisp_post_reset_delay

Change the lisc state to ~LISC_RESET.
Call qlisp_if_start() if reset completes.

qlisp_call_cbfns

NOTE

Invoke callback functions for chain of bp's.

Get the scsi_bus structure from the isc->if_drv_ata
while there is a bp in the cbfns list
Get scb from the bp
Set cbfns to bp->av_forw
Call scsi_cbfn()
done with while

qlisp_process_rsp

NOTE

Loop through response blocks and process

Get scsi_bus structure from isc->if_drv_data.
Get the local bus structure from the scsi_bus structure.

while queue is not empty

Get the iocb from the response queue.
Move the rsp_out to the next one with a circular wrap-up if reqd.
NOTE: Performance status completion for more than 5 I/O’s at one time.

Get the number of I/O handles in done_cnt.

while the done_cnt is not 0
Call qlisp_fast_complete on the corresponding handle
done with while
Signal chip that responses have been processed by moving out pointer

**qlisp_isr**

Acquire the driver spinlock.
Get the driver local control structure (lisc).

Read the device register.

while reg val is SEMA_LOCK

    Check the register for command status
    if the command is completed,
        Read the other mailbox commands
        if the command is completed with no errors,
            just break.
        else print an error message and break.
    else /* An asynch event */
        if the mb_status is one of the port RIO_POST statuses,
            then
                Get the number of I/Os done
                Call qlisp_fast_complete() on each of them
                else check for possible errors
done with while

if the mailbox command is not complete, get the rsp_in.
else
    Check the share state if it is in SHARE_RSP_QUEUE and if the response Q is not empty,
    Call qlisp_process_rsp().

if there was a reset earlier, call qlisp_post_reset_delay()
Check for other share->state status values.

if there are any bps whose i/o is completed,
Call qlisp_call_cbfns().

**qlisp_if_lun_open**

LUN open interface entry point called by SCSI services.
qlisp_if_abort

**NOTE**
Interface entry point called by SCSI services where an I/O is aborted

- From the buf structure, get the device number, scsi_bus structure, scsi_target structure and scsi_lun structure.
- Submit a mailbox command to abort device LUN.
- If the card is not in reset state, call qlisp_send_marker.

qlisp_if_bdr

**NOTE**
Interface entry point called by SCSI services where a device is reset.

- Submit a mailbox command Bus Device Reset Target
- If the card is not in reset state, call qlisp_send_marker

qlisp_reset_bus

**NOTE**
Causes a SCSI bus reset with a mp_timeout delay before I/Os are scheduled.

- Set lisc state to LISC_RESET
- Submit a mailbox command to Reset SCSI bus
- Call qlisp_post_reset_delay after 3 secs. (via a time-out)

qlisp_if_reset_bus

**NOTE**
Interface entry point called by SCSI services where the SCSI bus is reset

- Get scsi_bus structure from the device number
- Get isc from the scsi_bus structure
- Call qlisp_reset_bus
qlisp_send_marker

This routine is used to bring the device to a known state. This re-enables the adapter/target/lun request queue.
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 82, 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/malloc.h Necessary for acquiring and releasing memory.
/usr/include/sys/wsio.h WSIO context data and macro definitions.

Header Files for SCSI Drivers

/usr/include/sys/scsi.h SCSI-specific data definitions and ioctl commands.
/usr/include/sys/scsi_ctl.h SCSI subsystem data and macro definitions.

Header Files for Device Classes

In addition to the header file created for the specific driver, 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
/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 85, 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 85, Chapter 5, for further details.
The `drv_ops_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 88, Chapter 5, for further details.

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 90, Chapter 5, for further details.
static wsio_drv_data_t mydriver_data =
{
    "scsi_disk",
    TDEVICE,
    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 91, Chapter 5, for further details.

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, &scsi_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>:
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)();
    int              (*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

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()

Points 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_DDG 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 wscio_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 */
  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 */
  NULL, /* dd_read */
};
```
NULL,  /* dd_write */
mydriver_dd_ioctl,  /* dd_ioctl */
mydriver_dd_start,  /* dd_start */
mydriver_dd_done,  /* dd_done */
mydriver_dd_pass_thru_okay,  /* dd_pass_thru_okay */
mydriver_dd_pass_thru_done,  /* dd_pass_thru_done */
mydriver_dd_ioctl_okay,  /* dd_ioctl_okay */
mydriver_dd_status_list,  /* dd_status_list */
sizeof(mydriver_dd_status_list)/
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 physio(), scsi_strategy() will be called by your driver’s driver_strategy routine. Hence, you need not create the underlying ddsw->dd_read() and ddsw->dd_write() routines. However, if your driver calls scsi_strategy(), you must specify a ddsw->dd_strategy() routine.

The scsi_strategy() routine cannot block because it can be called on the Interrupt Control Stack (ICS) by a bp->b_call routine.

driver_open() Routine

mydriver_open(dev, oflags)
dev_t dev;
int oflags;
{
    return (scsi_lun_open(dev, &mydriver_ddsw, oflags));
}

driver_close() Routine

mydriver_close(dev)
dev_t dev;
{
    return scsi_lun_close(dev);
}

driver_read() Routine

mydriver_read(dev, uio)
dev_t dev;
struct uio *uio;
{
    return scsi_read(dev, uio);
}
**driver_write() Routine**

```c
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`.

```c
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.

```c
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**

```c
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_RECOVERED_ERROR,
        SA_ANY, SA_ANY, mydriver_check_residue, SA_DONE |
        SA_LOG_IT ALWAYS, 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_INTERMEDIATE, 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()

The following is a summary of IOCTLs that the LVM uses to interface with the device driver.

<table>
<thead>
<tr>
<th>IOCTL</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIOC_RSTCLR</td>
<td>The LVM uses each of these ioctlss on all bus paths. For example, DIOC_SET_PFTIMEOUT must be repeated on each bus path for I/O requests.</td>
</tr>
<tr>
<td>DIOC_SET_PFTIMEOUT</td>
<td></td>
</tr>
<tr>
<td>SIOC_RESET_BUS</td>
<td></td>
</tr>
</tbody>
</table>
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 command's arguments
- **dev**: The device number
- **flags**: The file-access flags

Return Values

dd_ioctl() is expected to return the following values:

- **0**: Successful completion.
- **<>0**: Error. Value is expected to be an errno.
Example

```c
#include <sys/scsi.h>
#include <sys/scsi_ctl.h>

mydriver_dd_ioctl ( 
    dev_t dev,
    int cmd,
    int *data,
    int flags
);
{
    struct scsi_lun *lp = m_scsi_lun(dev);
    struct mydriver_lun *llp = lp->dd_lun;
    struct scsi_tgt *tp = lp->tgt;
    struct scsi_bus *busp = tp->bus;
    struct inquiry_2 *inq = &lp->inquiry_data.inq2;
    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 */
            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;
    }
```
Writing SCSI Device Drivers

SCSI Driver Development, Step 8: Underlying Routines

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 mydriver_sync_cache(dev);
        else
            return 0; /* IR not on, just return */
    case DIOC_CAPACITY:
        *data = (llp->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 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.

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.
Conditions

dd_ioctl_okay() is called from sctl_ioctl() in a process context. It is called within a critical section; it may not block.

NOTE

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->inquiry_sz > 0), because it does not affect the nonpass-through device driver in any way.

Declaration

```c
int dd_ioctl_okay (  
    dev_t  dev,  
    int    cmd,  
    caddr_t  data,  
    int    flags  
);
```

Parameters

- `cmd` The command word
- `data` Pointer to the commands arguments
- `dev` The device number
- `flags` The file-access flags

Return Values

dd_ioctl_okay() is expected to return the following values:

- PT_OKAY Successful completion.
- 0 Error.

Examples

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

mydriver_dd_ioctl_okay (  
    dev_t  dev,  
    int    cmd,  
    int    flags  
);  
```

Parameters

- `cmd` The command word
- `data` Pointer to the commands arguments
- `dev` The device number
- `flags` The file-access flags

Return Values

dd_ioctl_okay() is expected to return the following values:

- PT_OKAY Successful completion.
- 0 Error.

Examples

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

mydriver_dd_ioctl_okay (  
    dev_t  dev,  
    int    cmd,  
    int    flags  
);  
```
Writing SCSI Device Drivers

SCSI Driver Development, Step 8: Underlying Routines

```c
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

#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.
 */
```c
scsi_lun_lock(lp);
scsi_tgt_lock(tp);
```
```c
tp->state |= T_ENABLE_SDTR;
```

```c
scsi_tgt_unlock(tp);
scsi_lun_unlock(lp);
```

```c
bzero(cdb, sizeof(cdb));
cdb[0] = CMDtest_unit_ready;
if (scsi_cmd(dev, SCB_DONT_PRINT, 6, cdb, 0, 0,
llp->mydriver_msecs, 0,&error))
{
    /*
    * Allow an incomplete open if this is a raw device.
    */
    if (major(dev) == mydriver_ddsw.raw_major)
    {
        scsi_lun_lock(lp);
        lp->state |= L_DISABLE_OPENS;
        scsi_lun_unlock(lp);
        return 0;
    }
}
return error;
```
dd_pass_thru_done 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);
    l1p->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
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)
```

Writing SCSI Device Drivers

SCSI Driver Development, Step 8: Underlying Routines

? 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_C))
    scb->cdb[1] |= WRITE_FUA_BIT;
if (llp->state & LL_WOE && !(bp->b_flags & B_READ))
  {
    if (lp->inquiry_data.inq2.dev_type == SCSI_MO)
      scb->cdb[1] |= 0x04;
    else /* SONY */
      scb->cdb[9] |= 0x40;
  }
scb->cdb_len = 10;
scb->max_msecs = llp->mydriver_msecs;
scb->max_retries = 5;
scb->flags = SCB_4BYTE | SCB_ORDERED_TAG;
if (llp->state & LL_STINGRAY)
  scb->flags &= ~SCB_ORDERED_TAG;

/* Assume that scb->io_id will be set by caller within * this CRIT */

/* Queue this bp into llp->active_bp_list HEAD for * tracking */

if (llp->active_bp_list != NULL)
  {
    scb->io_forw = llp->active_bp_list;
    head_scb = (void *) llp->active_bp_list->b_scb;
Writing SCSI Device Drivers

SCSI Driver Development, Step 8: Underlying Routines

```c
scb->io_back = head_scb->io_back;
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()`.

---

Chapter 9 429
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));
```
Writing SCSI Device Drivers

SCSI Driver Development, Step 8: Underlying Routines

Chapter 9

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
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);
}
```
Writing SCSI Device Drivers

SCSI Driver Development, Step 8: Underlying Routines

```c
mydriver_enqueue(lp, bp)
struct scsi_lun *lp;
struct buf *bp;
{
    int x;
    struct mydriver_lun *l lp = 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

#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 9-2, “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><code>m_scsi_lun()</code></td>
<td>Returns scsi_lun pointer corresponding to the dev_t parameter passed in.</td>
</tr>
<tr>
<td><code>disksort_enqueue()</code></td>
<td>Places I/O requests on queues maintained by SCSI Services.</td>
</tr>
<tr>
<td><code>scsi_dequeue()</code></td>
<td>Removes I/O requests from queues maintained by SCSI Services.</td>
</tr>
<tr>
<td><code>scsi_dequeue_bp()</code></td>
<td>Externally available to dequeue particular bp from circular list. Intended for use with LVM's B_PFTIMEOUT.</td>
</tr>
<tr>
<td><code>scsi_ddsw_init()</code></td>
<td>Called from device driver's <code>driver_dev_init()</code> routine. Causes initialization of blk_major and raw_major fields in the driver's switch table (ddsw).</td>
</tr>
<tr>
<td><code>scsi_lun_open()</code></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>
### SCSI Services (Continued)

<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 physio().</td>
</tr>
<tr>
<td>scsi_write()</td>
<td>Synchronous write routine, which calls physio().</td>
</tr>
<tr>
<td>scsi_ioctl()</td>
<td>Ioctl 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 sclt_io and a bp, attaches the sclt_io to the bp, forwards the bp 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 dmesg buffer, retry the I/O, or disable tags.</td>
</tr>
</tbody>
</table>
### Table 9-2 SCSI Services (Continued)

<table>
<thead>
<tr>
<th>Service</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>scsi_sense_action()</code></td>
<td>Interprets sense data for SCSI, CCS, or SCSI-2 compliance. It requires that the inquiry data for the device has been initialized by <code>scsi_init_inquiry_data()</code> before it can interpret it.</td>
</tr>
<tr>
<td><code>scsi_snooze()</code></td>
<td>Performs a sleep without tying up the processor. Must not be called by a thread of execution that holds any lock. Currently, this routine is used only by <code>scsi_disk</code> to delay subsequent device access following Inquiry to a particular model of Quantum disk drive.</td>
</tr>
<tr>
<td><code>scsi_log_io()</code></td>
<td>Records the I/O attempt and its results in the <code>dmesg</code> buffer.</td>
</tr>
</tbody>
</table>
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 82, Chapter 5.
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.

- pci_read_cfg_uintN_isc()
- pci_write_cfg_uintN_isc()
These functions take a configuration space offset (0x00–0xff) as their address inputs. See “Defined Constants”. The registers at addresses 0x00–0x3f are defined in the *PCI Local Bus Specification, Revision 2.1*, but the remainder of the space can be used by the card maker for any card-specific registers it sees fit to put there. In most cases, however, card-specific registers are placed in PCI memory space or PCI I/O space instead.

**PCI Memory Space**

Most cards place their registers for control, data buffering, and status in PCI memory space. In HP-UX systems, accesses to PCI memory space have higher performance than access to PCI I/O space. Registers mapped in PCI memory space respond to memory cycles on the PCI bus.

The following PCI Services access registers in PCI memory space.

- `READ_REG_UINTn_ISC()`
- `WRITE_REG_UINTn_ISC()`

These macros take Virtual Addresses (which are mapped to PCI memory addresses) as their address inputs. They have different effects depending on whether or not `PCI_LITTLE_ENDIAN_ONLY` is defined by the driver prior to including `pci.h`. See “The PCI_LITTLE_ENDIAN_ONLY Flag” for more details.

Mapped PCI memory space, on workstations only, can also be accessed directly. In this case 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 memory space

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 CardMastered 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 its next command from cacheline Y
- Processor writes command to cacheline Y “do more work”
- Processor checks status in cacheline X

If the processor's read of cacheline X (status) returns “working”, the processor assumes that the card has not checked its command yet. Therefore it has not gone to sleep and does not need to be awakened. If the status read returns “done”, the processor wakes up the card.

The crux of the problem here is that ordering is not enforced between the two cachelines and DMA reads can pass DMA writes. Thus, both the processor and the card's reads can return the original value. This would result in the card going to sleep and the processor not waking it up.

If you cannot place the status and commands on the same cacheline, you must use some other means to ensure correct behavior. One possible workaround would be to set a timeout to ensure that the above deadlock did not occur. In most cases, commands are written to the card's register, i.e., the command is not in host memory and the above scenario would not apply.
Ordering of Interleaved Processor Mastered Writes, PCI Card Mastered Reads of Host Memory, and Processor Mastered Writes of Host Memory

The following scenario does not meet the producer-consumer transaction ordering requirements.

- Processor writes a command to the PCI card to stop processing a task list in host memory, because the processor is about to update or change the list.
- Processor begins updating the task list in main memory.
- Card does a DMA read of the next (possibly being updated) element of the task list in main memory as a part of normal processing.
- Posted processor write to the card arrives at the card telling it to stop processing the list, which unfortunately it has just done.

The problem in this case is similar to the previous problem. DMA reads by a PCI master are allowed to pass processor writes to PCI space. Since processor writes are posted, ordering is not guaranteed on the combination of the internal system bus and the PCI bus. This situation can be avoided by doing a processor read of PCI space immediately following the processor write, as shown next.

- Processor writes a command to the PCI card to stop processing the task list.
- Processor does a “dummy” read of the PCI card (perhaps a status register on the card) to make sure that the “posted” write to PCI space has completed. Perhaps a read of card status is required here in any case to ensure that the DMA engine has stopped fetching tasks.
- Processor updates the task list in main memory.
- Processor writes a command to the PCI card to resume task processing.

The above behavior can occur on all shipping PCI based systems as of the date of this document. Drivers written for workstations should probably always ensure that, where necessary, posted writes are followed by “dummy” reads to ensure ordering. This behavior will probably not occur in servers due to chipset implementation, and may not occur in future workstation products.
PCI Endian Issues

HP PA-RISC is a big-endian system; for a multibyte quantity, the most significant byte (MSB) has the lowest address, and the least significant byte (LSB) has the highest. Intel’s x86 processors, on the other hand, are little-endian. Because PCI was derived from the PC world, it, too, is little-endian.

When multibyte words are transferred between the PCI bus and the system bus (HP PA-RISC), the bytes of the word are reversed or swapped by the hardware.

This insures that the receiving system can properly interpret and store the data, from most significant byte to least significant byte. This will not happen when the data is transferred byte-by-byte, but this method is inefficient.

Byte Swapping

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 \texttt{if\_reg\_ptr} field in the ISC structure is useful.

PCI Services automatically maps one memory-space register into the \texttt{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)
int *regsToInit;
int size;
{
    int i;
    for (i=0; i<size; i++)
        *regsToInit++ = 0;
}
```

These drivers may also use the READ_REG_UINTn_ISC() and WRITE_REG_UINTn_ISC() macros with the PCI_LITTLE_ENDIAN_ONLY flag defined in the pci.h header. The choice of whether to directly access a register or to use one of the macros essentially becomes whether or not you want to swap.

The READ_REG_UINTn_ISC() and WRITE_REG_UINTn_ISC() macros are the safest accessors of PCI memory space, but what they are actually defined to do depends on whether or not the PCI_LITTLE_ENDIAN_ONLY flag was defined by the driver before the driver source code included the pci.h header.

If the driver does NOT explicitly define PCI_LITTLE_ENDIAN_ONLY before including pci.h, then the macros expand into function calls that are guaranteed to byte-swap correctly and perform the memory access. This can be considered “extra safe” mode; it will always work on all bus adapters in the future. The function calls guarantee PCI-adapter-independence. However, extra function call overhead is added to the register access, reducing its performance.
If the driver DOES explicitly define `PCI_LITTLE_ENDIAN_ONLY`, the performance loss due to the function call is taken away. In this case, the macros are expanded by the preprocessor into a series of in-line instructions that byte-swap and perform the access without a function call, under the assumption that the PCI adapter under which the card is running has directly mapped the PCI memory space into driver-accessible PA I/O space. This assumption is valid for all current and planned PCI adapters, with the single exception of a few special PA internal system bus based server PCI card projects. All regular drivers (i.e., those that are not explicitly written to drive a specially-equipped PA internal system bus based card) will benefit from defining the `PCI_LITTLE_ENDIAN_ONLY` flag and should do so before including `pci.h`.

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_isc()`) in order to allow the device to master the bus. The use of this bit is illustrated in “Sample `driver_attach()` Routine”.

PCI Masters and Coherence

It is possible for prefetching of host memory by the hardware chipset to result in a PCI master reading stale data, even though the proper `dma_sync` calls have been made. This does not occur if the mapping is done with `wsio_map()` with flags `IO_NO_SEQ` and `IO_SAFE` set. See `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_getinterrupts()` 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_hdnl_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 0x0f
#define PCI_CS_INTERRUPT_PIN 0x3d

/* masks for configuration data */
#define PCI_CS_MULT_FUNC_MASK 0x80

/* bit definitions for configuration space command *register */
#define PCI_CMD_IO_SPACE 0x001
#define PCI_CMD_MEM_SPACE 0x002
#define PCI_CMD_BUS_MASTER 0x004
#define PCI_CMD_SPEC_CYCLES 0x008
#define PCI_CMD_MEM_WR_INVAL_EN 0x010
#define PCI_CMD_VGA_PAL_SNOOP 0x020
#define PCI_CMD_PARITY_ERR_RESP 0x040
#define PCI_CMD_WAIT_CYCLE_CNTL 0x080
#define PCI_CMD_SERR_ENABLE 0x100
#define PCI_CMD_FAST_BACK_EN 0x200
A Sample PCI Driver

The following example is a skeleton that demonstrates how to write a PCI device driver in HP-UX using PCI and WSIO Services. The only part of this example that is PCI-specific is the driver_attach() routine. The other parts are typical of all WSIO drivers. They are included here for context and completeness. Chapter 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 82, 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.
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 85, 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 88, Chapter 5, for further details.
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 90, Chapter 5, for further details.

```
static wsio_drv_info_t ZZZ_info =
{
    "ZZZ",     /* name */
    "blender", /* class */
    DRV_CHAR | DRV_SAVE_CONF | DRV_MP_SAFE, /* flags */
    -1,       /* block major number (-1 for dynamic) */
    -1,       /* character major number (-1 for dynamic) */
    NULL,     /* reserved */
    NULL,     /* reserved */
    NULL,     /* reserved */
};
```

The `wsio_drv_info_t` structure 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 91, Chapter 5, for further details.

```
static wsio_drv_info_t ZZZ_wsio_info =
{
    &ZZZ_info,
    &ZZZ_ops,
    &ZZZ_data
};
```
Sample WSIO Routines

Sample driver_install() Routine

A driver's `driver_install()` routine registers the driver and its structures with WSIO Services and the I/O subsystem. It also links the driver's `driver_attach()` function into the appropriate attach chain.

(If the device had a `driver_dev_init()` function, we would link it similarly into the dev_init chain here.)

---

**NOTE**

The name 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 82, Chapter 5.

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

    /* Register our driver information with WSIO services */
    ret = wsio_install_driver(&ZZZ_wsio_info);

    if (ret) {
        /*
         * If the install worked,
         * link ourselves into the pci_attach chain
         */
        ZZZ_saved_attach = pci_attach;
        pci_attach = ZZZ_pci_attach;
    }

    /*
     * Exit, returning the value we got
     * from the wsio_install_driver() call
     */
    return ret;
}
```
Sample driver_attach() Routine

For interface and monolithic drivers, the `driver_attach()` routine is linked into the global attach list for PCI drivers in the `driver_install()` routine.

A driver's `driver_attach()` routine is called whenever the system finds a piece of hardware it thinks the driver might want to claim (this driver put its `driver_attach()` function on the `pci_attach` chain, so the system calls it every time a new PCI device is found). The `driver_attach()` routine first checks to see if this is the type of hardware it can claim, then claims it if it wants it and performs whatever initialization the card requires.

- PCI Services will NOT enable a PCI device or function response to memory accesses, I/O accesses, or PCI device or function mastering of the bus. This ensures that a PCI device or function remains completely disconnected from the bus until after driver initialization. It is the responsibility of the driver to do this, as shown in the following sample attach routine.

- The driver is responsible for ensuring that the contents of a memory or I/O base register are not zero. All zeros indicates that either the specified configuration space register is not implemented by the PCI device or function or that the system could not find the resources to map the corresponding space into the system. If alternate register mappings exist, and those base registers are not zero, it is acceptable for the driver to use those mappings instead.

The driver must enable memory access, I/O access, and DMA.

```c
int ZZZ_pci_attach(uint32_t parm, struct isc_table_type *isc)
{
    uint8_t rev_id;
    uint16_t command_reg;
    uint32_t base_addr;
    
    PCI_ID *id = (PCI_ID *)&parm /* for LP64 */

    /*
    * see if this is our card
    */
    if (!id->vendor_id == ZZZ_VEN_ID &&
        id->device_id == ZZZ_DEV_ID) {
        goto exit0;
    }
}```
/*
 * If we use a standard bus interface chip we need to
 * check subsystem vendor ID and subsystem ID here
 * to make sure that our driver should be the
 * driver claiming this device
 */

/*
 * Get the card revision
 */
pci_read_cfg_uint8_isc(isc, PCI_CS_REV_ID, &rev_id);

/*
 * We must check the isc->if_reg_ptr
 * before we use it. If it's NULL,
 * we read our base register and map it ourselves.
 * But if isc->if_reg_ptr isn't NULL, PCI
 * services already did the mapping work for us
 */
if (isc->if_reg_ptr == NULL) {
    
    /*
     * We need to map our own base address.
     * Save the value in if_reg_ptr.
     * Get our physical base memory address.
     * For ZZZ, memory is at reg 0x10
     */
    pci_read_cfg_uint32_isc(isc, 0x10, &base_addr);

    /*
     * make sure we have a memory BAR
     * instead of an IO BAR
     */
    if (base_addr & 0x01) {
        printf("ZZZ - no memory BAR\n");
        goto exit0;
    }

    /*
     * Mask off the bottom four bits of the PCI
     * memory base register (see PCI spec for
     * significance)
     */
    base_addr &= ~0xf;

exit0:
*/
/ * Ensure this base register was mapped in by the system. If base_addr is 0, then the system * was unable to allocate us PCI memory space at all. */
if (base_addr == 0) {
    goto err0;
}
/
/*
* Get a virtual translation for card registers.
* Assume there are 512 bytes of registers.
* Save the value in if_reg_ptr.
*/
if ((isc->if_reg_ptr = map_mem_to_host(isc, base_addr, 512)) == NULL){
    goto err0;
}
/
/*
* Use if_reg_ptr to access the registers.
* Enable memory access and bus mastering *(note: other bits in the register must be preserved)
*/
pci_read_cfg_uint16_isc(isc,PCI_CS_COMMAND,&command_reg);
pci_write_cfg_uint16_isc(isc, PCI_CS_COMMAND, command_reg | PCI_CMD_MEM_SPACE | PCI_CMD_BUS_MASTER);
/
* Set up our interrupt handler.
* Note that -1 is the third argument to isrlink().
*/
if (isrlink(isc, ZZZ_isr, -1, isc, 0) < 0) {
    goto err1;
}
/
* set up our init routine to be run later *
/*
CONNECT_INIT_ROUTINE(isc, ZZZ_if_init);
/
* If everything okay, claim this card *
*/
isc_claim(isc, &ZZZ_wsio_info);
/*
* Exit without error
  */
goto exit0;

err1:
  /*
   * clean up the mapping
   */
  unmap_mem_from_host(isc, isc->if_reg_ptr, 512);

err0:
  /*
   * indicate that we had an error
   */
  PCI_ATTACH_DEV_INIT_ERROR(isc);

exit0:
  /*
   * Always exit by calling rest of chain
   * Use link established in ZZZ_install()
   */
  return ZZZ_saved_attach(parm, isc);
}

Other Driver Entry Point Routines

The other routines defined by the code above must also be declared and written. These functions include the following list:

- `ZZZ_if_init()`: Initialization of the card after the `driver_attach()` routine
- `ZZZ_isr()`: The driver's interrupt service routine
- `ZZZ_open()`: The `drv_ops_t`-defined entry point for `open()`.
- `ZZZ_close()`: The `drv_ops_t`-defined entry point for `close()`.
- `ZZZ_read()`: The `drv_ops_t`-defined entry point for `read()`.
- `ZZZ_write()`: The `drv_ops_t`-defined entry point for `write()`.
- `ZZZ_ioctl()`: The `drv_ops_t`-defined entry point for `ioctl()`.

The code for these functions is driver dependent. See “Step 6: Writing Entry Point Routines” on page 125 and “Step 7: Writing Other Driver Routines” on page 157, Chapter 5. See also `close(2)`, `ioctl(2)`, `open(2)`, `read(2)`, `write(2)`.
Writing PCI Device Drivers

A Sample PCI Driver
On-Line Addition/Replacement
On-Line Addition/Replacement (OLA/R) is a required feature for high-availability servers. This chapter describes writing I/O device drivers on HP-UX 11i for OLA/R support. This chapter mainly focuses on additions to WSIO interface, modifications required to drivers for supporting OLA/R, and handling OLA/R events at the driver. Pseudo code is given wherever required for a better understanding of the concepts under discussion.

In the examples and pseudo code present in this chapter “driver_name” is used. This can be replaced with your driver name when you actually develop a driver for OLA/R support.

The reader of this chapter is assumed to have a good understanding of:

- HP-UX I/O Subsystem
- WSIO driver development environment
- Writing a WSIO driver
- High-availability issues
- Application level impact on OLA/R of a driver of interest
Introduction

The ability to insert device controller cards and replace such cards while a system is being used, without the interruption of services to users not directly affected by the device resource, is a necessary capability for high-availability machines. On PCI based HP 9000 servers, this capability is provided as On-Line Addition/Replacement (OLA/R) of I/O cards. Support for OLA/R is implemented at - hardware, firmware and software. Specified here are the implementation details to provide support for OLA/R in the driver software of an I/O card.

The major functional areas included in this chapter are:

- Overview of the required modifications to the driver for OLA/R support
- The additions to the WSIO interface for OLA/R support
- Details of the enhancement/modification steps for OLA/R
- Details of handling OLA at the driver, MP-safe issues, resource allocation failure issues, and suspend and resume event handling for OLR
- Miscellaneous code changes at the driver for OLA/R
- Performing OLA/R of PCI I/O cards

Supported Hardware and Software

Servers on which OLA/R is supported:
L-Class, N-Class

Firmware Updates:
Any required f/w updates

OS Version:
HP-UX 11i and above
Overview of Driver Modifications Required for OLA/R

A driver can be enhanced or developed with OLA/R support by following these three steps:

1. **Register a generic event handler.**
   To support OLA/R functionality, each driver has to register an event handling function with the WSIO. WSIO calls this event handler to service OLA/R events like suspend and resume on the I/O adapter card the driver controls. The generic event handling function is registered in the driver install routine.

2. **Register a capability mask.**
   The driver has to register an event mask with the WSIO. The event mask specifies the capabilities of the driver in OLA/R event handling. This allows the same driver to support various capabilities on different instances of I/O adapter. The event capability mask is registered in the driver attach routine.

3. **Driver Event Handler.**
   To support OLA/R functionality in the driver, the driver has to handle suspend and resume events that are generated by the WSIO. On-Line Addition (OLA) does not generate any event. However since OLA can be done at any time once the machine is up and running, the attach and init routines of the driver must be made MP-safe and should back out of any errors during allocation of resources.

Detailed discussion on how to implement these three steps are present in later sections.

Since the OLA/R enhancements are not in the driver’s main performance path, there should not be any performance penalty on the driver’s regular data path.
WSIO Interfaces

WSIO provides a new set of data structures and services that allow driver enhancements for OLA/R support.

**wsio_drv_info_t**

A new field is added to the `wsio_drv_info_t` structure indicating the version of WSIO. The new `wsio_drv_info_t` structure is given below:

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

A driver would then use the following definition when it calls `wsio_install_driver`, `isc_claim` and `wsio_install_drv_event_handler`.

```c
static wsio_drv_info_t driver_name_wsio_drv_info = {
    &driver_name_info,
    &driver_name_ops,
    &driver_name_wsio_drv_data,
    WSIO_DRV_CURRENT_VERSION
};
```

The macro `WSIO_DRV_CURRENT_VERSION` is defined in `wsio.h`.

**WSIO Event Handling Structures**

WSIO and the driver that supports OLA/R interact through events and event completion callbacks. A new event data structure is defined in WSIO:

```c
typedef enum {
    WSIO_EVENT_SUSPEND   = 1<<0,
    WSIO_EVENT_RESUME    = 1<<1,
    WSIO_EVENT_REMOVE    = 1<<2,
    WSIO_EVENT_DEV_ERROR = 1<<3,
    WSIO_EVENT_BUS_ERROR = 1<<4,
    WSIO_EVENT_SELF_TEST = 1<<5,
} wsio_event_t;
```
For OLA/R functionality, only the first two events `WSIO_EVENT_SUSPEND` and `WSIO_EVENT_RESUME` are of interest.

typedef unsigned int wsio_event_id_t;

`event_id` is a tag to identify a call to a driver’s event handler and its completion. A driver will return it when it calls the completion call back. It is a number that WSIO uses to match up requests with replies. The callback function is defined as:

typedef int (*generic_complete_callback_t)(
    struct isc_table_type *,
    wsio_event_id_t,
    void *);

The event status is returned to WSIO using the third argument (void *) in the above `generic_complete_callback` definition. Although it is defined as a pointer, WSIO expects only one of the WSIO status values described below:

typedef enum {
    WSIO_OK = 0,
    WSIO_ERROR = -1,
    WSIO_INFO_NULL = -2,
    WSIO_HANDLER_NULL = -3,
    WSIO_DRV_NOT_FOUND = -4,
    WSIO_INVALID_ISC = -5,
    WSIO_INVALID_EVENT = -6,
    WSIO_NO_DRV_HANDLER = -7,
    WSIO_INVALID_COMBIN_EVENTS = -8,
    WSIO_UNSUPPORTED_EVENT = -9,
    WSIO_HA_NA = -10, /* Not an OLA/R or HA capable system */
    WSIO_DRV_FUNC_NULL = -11,
    WSIO_UNKNOWN_FUNC_TYPE = -12,
} wsio_ret_code_t;
The generic event structure \texttt{wsio\_generic\_event\_t} is defined as follows:

\begin{verbatim}
typedef struct wsio\_generic\_event {
   wsio\_event\_t event; /* suspend, resume, and so on */
   /* Event\_id is a tag to identify an instance of an 
   * event. Driver just passes it back in the callback */
   wsio\_event\_id\_t event\_id;
   struct isc\_table\_type *isc;
   generic\_complete\_callback\_t wsio\_completion\_cb;
   void *arg;
} wsio\_generic\_event\_t;
\end{verbatim}

\texttt{wsio\_generic\_event\_t} is passed to the driver event handler function. A driver event handler is a function that WSIO will call when an event, such as suspend, occurs.

The driver's event handler is defined as:

\begin{verbatim}
typedef void (*wsio\_drv\_event\_handler\_t) (wsio\_generic\_event\_t *);
\end{verbatim}
Driver Modifications for OLA/R Support

The driver must perform the following four steps to support OLA/R functionality:

- Register its event handler
- Register its event capability mask
- Set the suspend and resume event timeout values
- Driver’s event handler routine

Driver Registering Its Event Handler

The driver registers a generic event handler function by calling `wsio_install_drv_event_handler`. This should be called after installing the driver by calling `wsio_install_driver` in the driver's install routine. This service is passed with two arguments:

```c
int wsio_install_drv_event_handler(
    wsio_drv_info_t * drv_info,
    wsio_drv_event_handler_t event_handler);
```
This is called in the `driver_install` routine. Pseudo code for a typical driver install routine is as follows:

```c
int driver_name_install(void)
{
    if (return_value = wsio_install_driver(&driver_name_drv_info)) {
        saved_attach = pci_attach;
        pci_attach = driver_name_attach;
    } else { /* Install Failure */
        return 0;
    }

    /* driver_name_event_handler() is the generic event * handler implemented in the driver. */
    if (wsio_install_drv_event_handler(&driver_name_drv_info, driver_name_event_handler) != WSIO_OK) {
        /* Driver not registered its event handler.
         * Driver’s normal operation may not be affected.
         */
    }

    return return_value;
}
```

**Driver Registering Its Event Mask**

This entry point is for a driver to call inside its attach routine after calling `isc_claim()`. There will be two parameters for this call; `isc` and event mask. Event mask is a `uint64_t` representing a possibility of 64 operations associated with the `isc`.

```c
typedef uint64_t wsio_event_mask_t; /* This is an OR of wsio_event_t described in last section. */

int wsio_reg_drv_capability_mask(
    struct isc_table_type *isc,
    wsio_event_mask_t event_mask);
```
The following is pseudo code for a typical driver attach routine

driver_name_attach()
{
        ....
        wsio_event_mask_t driver_name_event_mask = 0;

        /* There is no specific event for OLA. OLA does not use
           * the driver event handler. It uses the normal attach
           * and init routines. To support OLR, the driver
           * requires to handle two events - suspend and resume.
           */
        driver_name_event_mask = WSIO_EVENT_SUSPEND |
                               WSIO_EVENT_RESUME;

        .... normal attach processing ....

        isc_claim(isc, &driver_name_wsio_drv_info);

        /* Register driver event capability mask.
           * NOTE: This should be called only after
           * isc_claim() is called.
           */
        return_value = wsio_reg_drv_capability_mask(isc, 
                                          driver_name_event_mask);
        if(return_value != WSIO_OK && return_value != 
           WSIO_HA_NA) { 
                /* Registering driver event mask failed.
                   * Continue with normal processing.
                   */
        }

        .... normal processing ...
}
Driver Event Timeout Values

When WSIO issues an event request to a driver by calling the driver’s event handling function, it expects the driver to complete the event, and to call the callback function within a specified period of time. If the driver fails to complete the event within that period of time, the event goes into a timeout state. It is undesirable for any event to enter a timeout state, so a driver must set the timeout value for each event such that the timeout period will never be exceeded. However, if a driver timeouts for some reason during a suspend or resume event it still can reply anytime later to change the status.

The default timeout limit is ten seconds. If a driver needs more than this to complete an OLA/R event, it should set the timeout limit to the required value using the WSIO service wsio_set_parm. The current value of a specific parameter can be obtained from wsio_get_parm.

The data structure wsio_parm_t defines the valid parameters for wsio_set_parm() and wsio_get_parm().

typedef enum {
    WSIO_HW_SUSPEND_TIMEOUT, /* Hardware Suspend Timeout param */
    WSIO_HW_RESUME_TIMEOUT,  /* Hardware Resume Timeout param */
    WSIO_HW_REMOVE_TIMEOUT,  /* Hardware Remove Timeout param */
    WSIO_HW_ERROR_TIMEOUT,   /* Hardware Error Timeout param */
    WSIO_IDENTIFY_CHILD,     /* Function that identifies if a child is an interface */
    
    WSIO_LAST_PARM
} wsio_parm_t;

WSIO service wsio_set_parm() takes three arguments.

int wsio_set_parm (struct isc_table_type * isc,
    wsio_parm_t parm,
    void * value);

Even though the third parameter 'value' is declared as a pointer to a void, WSIO will read it as a value and NOT as a pointer. So do not use a pointer to a value; instead, use a defined value. The value specified is in microseconds.

This can be called at any place in the driver.

In the example given below, the timeout values are set in the driver attach routine after registering the driver event capability mask.
Pseudo code for a typical attach routine is shown below.

```c
{  driver_name_attach()

   ubit32_t olar_timeout = 15000000; /* timeout set to 15 sec */

   ... normal processing ...

   isc_claim();

   if(wsio_regDrv_capability_mask(isc, event_mask) == WSIO_OK) {
      /* Set the suspend and resume events * timeout period */
      if(wsio_set_parm(isc, WSIO_HW_SUSPEND_TIMEOUT,
                      (void *) olar_timeout) != WSIO_OK) {
         wsio_regDrv_capability_mask(isc, 0);
      } else if(wsio_set_parm(isc, WSIO_HW_RESUME_TIMEOUT,
                              (void *) olar_timeout) != WSIO_OK) {
         wsio_regDrv_capability_mask(isc, 0);
      }
   }

   ... normal processing ...
}
```

**Event Handling Function**

WSIO generates an event to a driver when it receives an OLA/R request (from SAM or RAD) on an I/O card the driver controls. WSIO calls the driver's event handling function which the driver has registered with the WSIO during the driver's installation time. It is the driver's responsibility to make sure that an OLA/R event is never timed out. OLA does not generate any event and follows the normal attach and init routines path. So the events supported by the driver would be for suspend and resume.
The following is pseudo code for a typical driver event handling routine:

```c
void
driver_name_event_handler(wsio_generic_event_t *handler_arg) {

    /* Switch based on the event The timeout calls
    * in the switch cases below are to allow the
    * event handler to return to WSIO immediately.
    * The timeout value of 0 is used in those cases
    * where there is no need to delay execution of the
    * individual event handler.
    * /
    switch (handler_arg->event) {

    case WSIO_EVENT_SUSPEND:
        * Save callback function and other info like
          event_id
        * Call timeout(driver_name_suspend,
                      driver_name_suspend_info, 0);
        * break;

    case WSIO_EVENT_RESUME:
        * Check if it is a like-for-like card
        * Save callback function and other info like
          event_id
        * Call timeout(driver_name_resume,
                      driver_name_resume_info, 0);
        * break;

    default:
        handler_arg->wsio_completion_cb(handler_arg->isc,
                                         handler_arg->event_id, WSIO_UNSUPPORTED_EVENT);
        break;
    }

    return;
}
```
Handling OLA/R

This section describes how On-Line Addition (OLA) and On-Line Replacement (OLR) have to be handled when designing the driver.

For detailed information on performing OLA/R operations, refer to the user manuals. The following information is only a description of the sequence of required operations necessary to write a driver with OLA/R support.

OLA

On-Line Addition of a card instance can be performed with the following steps:

1. Power-off the slot from SAM or RAD.
2. Insert the card into the slot.
3. Power-on the slot from SAM or RAD.
4. Run `ioscan`, optionally passing the H/W path of the slot.

As mentioned earlier, On-Line addition of an I/O card does not generate any WSIO event that the driver event handler will handle. Instead, the driver’s `attach` and `init` routines are called, just as they would be during the boot. There are two issues that a driver should handle to support OLA. They are:

- **Driver’s attach and init routines should be MP-safe**
- **Driver should handle resource allocation failures during the attach and init.**

MP Safe

The `attach` and `init` routines must be MP-Safe; an OLA can occur at any time once the machine is up and running. If the driver has any global resources which are common to all instances, it is advisable to allocate them during the driver install time. If required global locks can be acquired and released during `attach` and `init` routines to serialize access and/or to protect data.
### Resource Allocation Failures

Since an OLA can occur at any time once the machine is up and running, resource allocation may fail because of resource shortage problem. So the driver should be able to retrace all the steps and release all the resources that are allocated up until the failure occurred. This will take the driver back to a clean state.

To facilitate the back out of a driver because of a failure during `attach` and `init`, a list of resource allocation services and their corresponding releasing services are given below. A word of caution is, release or free the resources in LIFO order; i.e., the most recently allocated resources are released first.

<table>
<thead>
<tr>
<th>Resource Allocation/Acquire</th>
<th>Resource Free/Release</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>kmalloc</code></td>
<td><code>kfree</code></td>
</tr>
<tr>
<td><code>wsio_map</code></td>
<td><code>wsio_unmap</code></td>
</tr>
<tr>
<td><code>wsio_map_port</code></td>
<td><code>wsio_unmap_port</code></td>
</tr>
<tr>
<td><code>wsio_allocate_shared_memory</code></td>
<td><code>wsio_free_shared_memory</code></td>
</tr>
<tr>
<td><code>wsio_allocate_shared_mem</code></td>
<td><code>wsio_free_shared_mem</code></td>
</tr>
<tr>
<td><code>wsio_allocate_dma_handle</code></td>
<td><code>wsio_free_dma_handle</code></td>
</tr>
<tr>
<td><code>wsio_map_dma_buffer</code></td>
<td><code>wsio_unmap_dma_buffer</code></td>
</tr>
<tr>
<td><code>wsio_intr_alloc</code></td>
<td><code>wsio_intr_free</code></td>
</tr>
<tr>
<td><code>wsio_intr_activate</code></td>
<td><code>wsio_intr_deactivate</code></td>
</tr>
<tr>
<td><code>isrlink</code></td>
<td><code>isrunlink</code></td>
</tr>
<tr>
<td><code>hw_ift_attach</code></td>
<td><code>hw_ift_detach</code></td>
</tr>
<tr>
<td><code>get_nmid</code></td>
<td><code>return_nmid</code></td>
</tr>
<tr>
<td><code>isc_claim</code></td>
<td>WSIO will clean it up when it receives an error from driver <code>init</code>.</td>
</tr>
</tbody>
</table>
All drivers must provide a return value from the `init` functions. The values must be either:

- `WSIO_OK (0)` on success.
- `WSIO_ERROR (-1)` on failure.

Refer to the ENET sample driver for an implementation example.

**OLR**

To replace a bad I/O card when the system is on-line, OLR can be used. Usually the following steps are followed while OLR of an I/O card:

1. Suspend the driver instance of the I/O card from SAM or RAD.
2. Power-off the slot from SAM or RAD.
3. Remove the card.
4. Insert a new I/O card.
5. Power-on the slot from SAM or RAD.
6. Resume the driver instance of the I/O card from SAM or RAD.

When writing a driver to support OLR, the driver should handle the events that happen during steps 1 and 6.

WSIO guarantees that suspend and resume requests will be single-threaded. Also, the underlying OLA/R infrastructure guarantees that no suspend or resume events are triggered to the same card instance until the previous event processing is completed. For OLR, the underlying infrastructure does not trigger a resume event if the replacement card is outright incompatible. The driver is expected to do a more detailed check, like comparing the PCI sub-system ID etc., before claiming the replacement card.

The following WSIO OLA/R device driver state diagram shows the driver states from the perspective of WSIO. The internal driver states will differ from driver to driver. The timeout states in the diagram should not occur if the driver sets an appropriate value in the corresponding timeout values using `wsio_set_parm()`. Since recovery from the timeout states cannot be fully defined, it is very important that these states are never reached.
Figure 11-1 OLA/R State Diagram

The numbered state transition labels in the figure do not represent a sequence. The transitions are as follows:

- Normal suspend sequence is 1, 2.
- If a suspense fails, the sequence is 1, 5.
NOTE

A SUSPEND MUST NEVER FAIL

- If a suspend timeout occurs, the sequence is 1, 6. From the SUSPENDING TIMEOUT state, 7 can occur if the driver calls the callback function with a status of WSIO_OK after a timeout has occurred. Transition 8 can occur if the driver calls the callback function with a status of WSIO_ERROR after the timeout has occurred.

- The normal resume sequence is 3, 4.

- If a resume fails, the sequence is 3, 9. From the RESUMING TIMEOUT state, 11 can occur if the driver calls the callback function with status WSIO_OK after a timeout has occurred. Transition 12 can occur if the driver calls the callback function with a status of WSIO_ERROR after the timeout has occurred.

Suspend

The sequence of events during a suspend is as follows:

1. Wait for the correct state. Definition of the "correct state" depends on a driver. But speaking in broad terms, no critical operation should be in progress like device reset, blocked on I/O request, etc. Pseudo code to perform this follows:
driver_name_event_handler(handler_arg) {
    
    ....

    switch(handler_arg->event) {
    
    ....

    case WSIO_EVENT_SUSPEND:
        if(driver_defined_state == DRIVER_DEFINED_RESET ||
            driver_defined_state == DRIVER_DEFINED_IOBLOCK
            || ...) {
            /* Set a timeout for suspend routine */
            timeout(driver_name_suspend_handler,
                    suspend_info, PREDEFINED_WAITING_TIME);
        } else {
            /* Call suspend handler. No need to wait here */
            timeout(driver_name_suspend_handler,
                    suspend_info, 0);
            break;
        }
    
    ...

    }

    ...

}  

If the driver can not set any predetermined time as shown above, the
driver's suspend handler can timeout itself for some number of clock
ticks until the driver comes out of its critical operation state. Pseudo
code to perform this follows:
A suspend request should never occur while a driver is suspending or suspended. If one occurs, the driver should immediately call the callback function with a status of WSIO_INVALID_EVENT. While a driver is suspending or suspended, some control requests, such as a reset or abort (or any other request which would normally cause the driver to enter a different state, or interact directly with the adapter) should be rejected. In these cases, the control request path must check for the SUSPENDED state (or equivalent) as well as any state the driver may be in during suspension for certain requests. These are just guidelines and what to do while handling a control request during and after a suspend operation is driver dependent.

2. If the driver has any timers, (other than timers specifically used to time the suspension sequence), the timers must be cancelled using utimeout()

3. Quiesce the device. This is driver dependent. After this, the device is not expected to perform any I/O operations or generate interrupts to the driver.

4. Save the required device information for a following resume operation. For example, when only like-for-like replacement is allowed during resume, comparing the vendor ID, subsystem ID, etc., can be used in identifying a suitable replacement I/O card.
5. Save any other required information. This is driver dependent. Generally speaking, this information would include the required state or configuration of the device the replacement I/O card should be brought to.

6. Call the callback function with a success.

Refer to the ENET sample driver for an implementation example.

**Resume**

A resume request should only occur when the driver is suspended, and not currently in the process of resuming. If the resume event occurs at any other time, the driver should call the callback function with a status of `WSIO_INVALID_EVENT`.

If resume fails, the driver must return to the SUSPENDED state, and the callback function must be called with a status of `WSIO_ERROR`.

While a driver is resuming, some control request, such as a reset or abort (or any other request which would normally cause the driver to enter a different state, or interact directly with the adapter) should be rejected. Therefore, the control request path must check for the RESUMING state (or equivalent) as well as any state the driver may be in during resuming for certain requests. These are just guidelines and what to do while handling a control request during and after a resume operation is driver dependent. The sequence of events during a resume is described below.

1. Check for a like-for-like replacement. Currently, the PCI CDIO checks for Vendor and Device IDs. If like-for-like replacement conditions do not meet the requirements of the driver, the driver must return to the SUSPENDED state, and the callback function must be called with a status of `WSIO_ERROR`.

2. Initialize the device. Again, this is driver dependent and what is part of initialization depends on the device and the driver.

3. Restore the state of the device if required. This is driver dependent.

4. Configure the device with the information saved during the suspend. This is driver dependent.

5. Call the callback function with `WSIO_OK` status.
Pseudo code for a driver resume routine is as follows:

```c
driver_name_event_handler(handler_arg)
{
    ...

    switch(handler_arg->event) {
        ....

        case WSIO_EVENT_RESUME:
            /* Check if the driver is in suspended state*/
            /* Test of like-for-like replacement. */
            /* Call the driver resume handler */
            timeout(driver_name_resume_handler, resume_info, 0);
                break;
        ....
    }
    ....
}

driver_name_resume_handler(resume_info)
{
    ....

    /* Turn on PCI memory access and bus master capability on host, if applicable */
    /* Initialize the device */
    /* Configure the device */
    /* Call callback function with WSIO_OK status */
    return;
}
```

Refer to the ENET sample driver for an implementation example.
Miscellaneous Changes Required at the Driver

As mentioned earlier, while processing control path requests like device reset, configuring the device etc., checks are required to see whether the driver is in SUSPENDED or RESUME states, or in the process of suspending or resuming. A driver state flag can be included in the driver control structure to keep track of the driver's OLA/R states.

If the driver is in SUSPENDED or RESUME states, the request should be reject with ENXIO. A portion of pseudo code for a driver reset is given below.

```c
driver_name_reset(reset_info)
{
....
  /* Addition state check */
  if(driver_olar_state == SUSPENDED) {
    /* Return ENXIO saying that the device is busy */
    return ENXIO;
  }
  ....
}
```
Performing OLA/R of PCI I/O Cards

HP-UX provides two interfaces to perform OLA/R operations on PCI I/O cards:

- SAM (GUI)
- /sbin/rad (commandline)

Currently, only commandline tool /sbin/rad can be used to issue OLA/R requests on I/O cards that are controlled by third party drivers. SAM will be extended in later releases with support for performing OLA/R operation on I/O cards that are controlled by third party drivers.

Refer to the user documentation on "Managing PCI cards with OLA/R" for a detailed description of the use of these tools.
Chapter 12  Dynamically Loadable Kernel Modules
This chapter describes how to set up a single-user or multiuser system. The following topics are discussed:
Managing Dynamically Loadable Kernel Modules

This section presents the concepts and procedures which are necessary to understand, configure, and manage Dynamically Loadable Kernel Modules (DLKMs).

This section is divided into the following three topical sections:

Table 12-1  DLKM Topical Sections

<table>
<thead>
<tr>
<th>Topic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DLKM Concepts</td>
<td>This section provides an introduction to DLKM, important DLKM terms, and detailed technical DLKM concepts.</td>
</tr>
<tr>
<td>DLKM Tools</td>
<td>This section provides a summary of tools collectively known as the Kernel Configuration Tool Set which are used when installing, configuring, and managing DLKM modules.</td>
</tr>
<tr>
<td>DLKM Procedures</td>
<td>This sections presents the key DLKM procedures used in the three phases of managing DLKM modules: Preparation, Loading, and Maintenance.</td>
</tr>
</tbody>
</table>

This section focuses on configuring and managing loadable device drivers, as they constitute the majority of supported module types for HP-UX release 11.0 and later.

NOTE

The HP-UX kernel infrastructure provides the ability to dynamically load and unload DLKM drivers. While the base set of drivers shipped with HP-UX release 11i are not DLKM enabled, many Independent Software Vendors (ISVs) are coding DLKM enabled drivers for the hardware they provide.
Check the documentation that shipped with any 3rd-party drivers you have to determine if they are DLKM enabled.

**DLKM Concepts**

This section provides a conceptual overview of DLKM features and functionality by:

- defining DLKM at a high level
- explaining terms and concepts essential to understanding DLKM
- describing how DLKM modules are packaged in HP-UX
- identifying the types of kernel modules currently supported by DLKM
- describing the advantages of writing kernel modules in DLKM format
- examining DLKM module functions and configuration parameters

**What is DLKM?**

The *Dynamically Loadable Kernel Modules Infrastructure* is an HP-UX operating system feature that allows “DLKM Enabled” kernel modules to be dynamically loaded into, or unloaded from, the HP-UX kernel without having to re-link the entire kernel or reboot the system.

Previously, to install a new driver you had to edit the system file, run the `config` or `mk_kernel` commands to create a new kernel, shut down the system, and then bring the system back up before you could use the new driver.

The DLKM feature not only provides the infrastructure to load kernel modules into a running system, but it also allows a kernel module to be statically linked when rebuilding the kernel. Setting a flag in one of the DLKM module’s configuration files determines whether the module is to be configured as dynamically loadable or statically linked.
Important Terms and Concepts

The DLKM infrastructure allows kernel modules to be configured in a number of different ways. The following table considers the different ways a kernel module can be configured and loaded, and clearly defines each as a term. It also clarifies the relationship between each term as seen by the HP-UX kernel.

Table 12-2 Important Terms and Concepts

<table>
<thead>
<tr>
<th>Term / Concept</th>
<th>Definition</th>
</tr>
</thead>
</table>
| Kernel Module        | A Kernel Module is a section of kernel code responsible for supporting a specific capability or feature. For example, file system types and device drivers are kernel modules.  
In the kernel configuration context, a kernel module may be viewed as an object that can be installed, removed, configured or built on a system, either statically or dynamically.  
There are two categories of kernel modules:  
  • *Traditional Module*  
  • *Modularly-packaged Module* |
| Traditional Module   | A Traditional Module is a Kernel Module whose configuration data has not been modularized and can only be statically linked to the kernel.  
In the kernel configuration context, configuration information about Traditional Modules is maintained in the shared `master` and `system` files, and can only be accessed upon booting a kernel in which they have been statically-configured. |
Dynamically Loadable Kernel Modules
Managing Dynamically Loadable Kernel Modules

Table 12-2  Important Terms and Concepts (Continued)

<table>
<thead>
<tr>
<th>Term / Concept</th>
<th>Definition</th>
</tr>
</thead>
</table>
| Modularly-packaged Module | A Modularly-packaged Module is a Kernel Module whose configuration data has been modularized (not shared with other kernel modules), which is a pre-requisite for DLKM-enabling the Kernel Module. In the kernel configuration context, this means that the module uses its own `master` and `system` files (as opposed to the shared `master` and `system` files in which Traditional Modules are configured). In order to be classified as a Modularly-packaged Module, the module must contain it's own `master` and `system` files, as well as an individual object file, `mod.o`, that implements the module. A Modularly-packaged Module can be dynamically loaded into the HP-UX kernel only if that module includes the module wrapper code and additional data structures. For this reason, we place Modularly-packaged Modules in two categories:  
  - *Static Modularly-packaged Modules*  
  - *Loadable Modules (or DLKM Modules)*  
  
The terms Loadable Module and DLKM Module are interchangeable. |
| Static Modularly-packaged Module | A Static Modularly-packaged Module is a Modularly-packaged Module that can only be linked statically to the kernel. In the kernel configuration context, this means that the module uses its own `master` and `system` files but does not contain the module wrapper code and additional data structures that provide the dynamic loading and unloading ability. |
A Loadable Module (or DLKM Module) is a Modularly-packaged Module with the capability to be dynamically loaded into a running kernel.

In the kernel configuration context, this means that the DLKM module uses its own *master* and *system* files and contains the module wrapper code and additional data structures that provide the dynamic loading and unloading ability.

However, when a DLKM module is written with self-contained module wrapper code and packaged with module-specific *master* and *system* files, it can still be statically-configured into the kernel.

For this reason, we place Loadable Modules in two categories:

- *Statically-configured Loadable Module*
- *Dynamically-configured Loadable Module*

A Statically-configured Loadable Module is a DLKM module that has the capability to be dynamically loaded but instead is configured to be statically built into the kernel.

In the kernel configuration context, this means that the module-specific *system* file was updated to indicate static configuration.

Because it is now statically built into the kernel, it cannot be unloaded from or reloaded into loaded into the kernel dynamically.

<table>
<thead>
<tr>
<th>Term / Concept</th>
<th>Definition</th>
</tr>
</thead>
</table>
| Loadable Module (DLKM Module) | A Loadable Module (or DLKM Module) is a Modularly-packaged Module with the capability to be dynamically loaded into a running kernel. In the kernel configuration context, this means that the DLKM module uses its own *master* and *system* files and contains the module wrapper code and additional data structures that provide the dynamic loading and unloading ability. However, when a DLKM module is written with self-contained module wrapper code and packaged with module-specific *master* and *system* files, it can still be statically-configured into the kernel. For this reason, we place Loadable Modules in two categories:  
- *Statically-configured Loadable Module*  
- *Dynamically-configured Loadable Module* |
| Statically-Configured Loadable Module | A Statically-configured Loadable Module is a DLKM module that has the capability to be dynamically loaded but instead is configured to be statically built into the kernel. In the kernel configuration context, this means that the module-specific *system* file was updated to indicate static configuration. Because it is now statically built into the kernel, it cannot be unloaded from or reloaded into loaded into the kernel dynamically. |
Managing Dynamically Loadable Kernel Modules

Table 12-2  Important Terms and Concepts (Continued)

<table>
<thead>
<tr>
<th>Term / Concept</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamically-configured Loadable Module</td>
<td>A Dynamically-configured Loadable Module is a loadable module which has been fully configured to be dynamically loaded into or unloaded from the kernel without having to re-link the entire kernel or reboot the system. To summarize the terminology presented in this table, a Dynamically-configured Kernel Module is all of the following:</td>
</tr>
<tr>
<td></td>
<td>• a Modularly-packaged Module (Which is a Kernel Module that uses module-specific master and system files.)</td>
</tr>
<tr>
<td></td>
<td>• a Loadable Module (or DLKM Module) (Which is a Modularly-packaged Module that contains the wrapper code and additional data structures and uses module-specific master and system files, but still could be configured as dynamic or statically-linked.)</td>
</tr>
<tr>
<td></td>
<td>• a Dynamically-configured Loadable Module (Which is a DLKM Module that has been configured to be fully capable of dynamic loading into, and unloading from the running kernel.)</td>
</tr>
<tr>
<td>Module Wrapper</td>
<td>The additional code and data structures added to a kernel module which enable the DLKM mechanism to logically connect and disconnect a loadable module to and from the running kernel.</td>
</tr>
</tbody>
</table>

DLKM Module Packaging

The DLKM infrastructure specifies that:

- a kernel module must be packaged modularly with at least:
  - its own master and system files
  - its own mod.o object file that implements only that module
the mod.o object file must contain the Module Wrapper code (although full optimization is optional).

**NOTE**

See the `master` (4) manpage for descriptions of the two kinds of `master` files, and the `config` (1M) manpage for a description of the traditional and modular system files.

Kernel modules written as traditional modules are still fully supported in HP-UX. Driver developers are encouraged to re-package their static modules according to the module packaging architecture introduced with DLKM modules.

**DLKM Module Types**

The DLKM feature currently supports the following types of kernel modules:

- WSIO class drivers
- WSIO interface drivers
- STREAMS drivers
- STREAMS modules
- Miscellaneous modules—for example, modules containing support functions not required in the statically-configured kernel but shared among multiple loadable modules

**DLKM Advantages**

DLKM modules provide many advantages relative to static modules, including:

- reducing time spent on device driver development by streamlining the driver installation process
- making it easier for administrators to install device drivers from other vendors
- improving system availability by allowing device drivers and other modules to be configured into the kernel while the system is running
- conserving system resources by unloading infrequently used modules when not in use
Dynamically Loadable Kernel Modules
Managing Dynamically Loadable Kernel Modules

- providing administrators with the ability to demand load and unload modules
- providing the kernel with the ability to automatically load modules

Auto loading occurs when the kernel detects a particular loadable module is required to accomplish some task, but the module is not currently loaded. The kernel automatically loads the module.

**DLKM Driver Loading Concepts**

When a module is dynamically loaded, its object file is read from disk and loaded into newly allocated kernel memory. Once in memory, the module's symbols are relocated and any external references are resolved. Special code in the module is then executed to perform any required module-specific setup. Then the code specific to the module's type, if any, is executed, making the newly loaded module accessible to the rest of the kernel.

A module can be loaded in the following ways:

- **Demand Load**
  A demand load is a user level request for a specific module to be loaded. The load is accomplished through the kmadmin command.

- **Autoload Event**
  An autoload occurs when the kernel detects that a specific module is required to provide the functionality necessary to perform a task. The load is triggered by the initiation of the task. Once the required module is loaded, the task continues.

A loadable module's _load() function performs any initialization tasks required by the module before the module is logically connected to the kernel. Typical initialization tasks include acquiring private memory for the module and initializing devices and data structures.

- If the module is unable to initialize itself, the _load() function must free any memory that it allocated and undo any other action that it took prior to the failure including canceling all outstanding calls to timeout.

**DLKM Driver Unloading Concepts**

When the functionality provided by a module is no longer needed the module can be unloaded, thus freeing its resources for later use.
When a module is unloaded, the code specific to the module's type, if any, is executed to disconnect the module from the kernel. Then, special code in the module is executed to perform any module-specific cleanup. Finally, the memory allocated to the module is freed.

A module may be unloaded only by a user level request specifying the module to be unloaded. The unload is accomplished through the kmadmin command. This request may fail for a number of reasons, the most common being that the module is busy at the time. An example of this would be attempting to unload a device while there are outstanding opens on the device.

A loadable module's _unload() function is called by the DLKM mechanism whenever the module is about to be removed from active memory. The function may be given any name (typically module_name_unload); a pointer to the _unload() function is obtained from the module's wrapper.

The module's _unload() function cleans up any resources that were allocated to the module, and it must remove all references to the module. Typical cleanup tasks include releasing private memory acquired by the module, removing device interrupts, disabling interrupts from the device, and canceling any outstanding timeout requests made by the module.

The module's _unload() function returns 0 on success and an errno value on failure. In the event of failure, the function leaves the module in a sane state, since the module will remain loaded after the return.

The system will never attempt to unload a module that it thinks is busy. However, the system cannot determine under all cases when the module is in use. Currently, a module is considered to be busy when another module that depends on it is also loaded. In addition, WSIO class drivers and STREAMS drivers track the open() and close() calls; these types of modules are busy whenever there is at least one open on the device using the driver. Under most other circumstances, the module determines for itself whether it is appropriate for it to be unloaded. When a module is still in use, its _unload() function returns a non-zero value to cancel the unload.

The argument passed to the _unload() function is the same type-specific value that was passed to the module's _load() function. The use of this argument is described in section “STREAMS Drivers”. 
DLKM Driver Configuration Concepts

Since kernel modules written in the DLKM format can be configured as either dynamically loadable or statically-configured, DLKM-compatible device drivers must accommodate either configuration.

Through the use of configurable module attributes, System Administrators can control the various functions of a DLKM driver, including whether it is dynamically loaded or statically-configured.

This section provides attributes and keywords for:

- required components of a DLKM driver
- optional components of a DLKM driver

It also presents a brief description of STREAMS and Miscellaneous drivers. See the section DLKM Tools for detailed instructions on how to modify the configurable module attributes presented here.

NOTE

The system must be in a run-time state before dynamic module loading is available. Thus, kernel modules required during system boot must be configured as statically-configured.

master File Definition

Each DLKM module has its own master file. The format of the master file includes the following section keywords:

- $VERSION—indicates the version number for the file format. Version is defined as an integer and starts from one. A single line containing the only supported version (version 1) is entered.
- $LOADABLE—indicates that the module supports dynamic loading. If this section keyword does not exist, the module can only be statically-configured into the kernel.
- $INTERFACE—identifies the interface names and versions on which the module is built. For HP-UX, versions 11.0 and higher, a single line is entered containing the word base.
- $TYPE—indicates the module type and the type specific information. Valid types are wsio_class, wsio_intfc, streams_mod, streams_drv, and misc.
Other sections (if required)—$DRIVER_DEPENDENCY, $TUNABLE, and $DRIVER_INSTALL.

The $DRIVER_DEPENDENCY section, defines the names of all other modules that this module depends upon.

The $TUNABLE section defines the names and default values of the tunable parameters (variables) for the module. Default (and optionally minimum) values for tunable parameters are entered here.

The $DRIVER_INSTALL section defines the module’s name and associated block and/or character major device number(s).

**system File Definition**

Every DLKM module requires a system file. The system file includes the following three mandatory and one optional section keywords:

- $VERSION—indicates the version number for the file format. Version 1 is the only supported file-format.

**NOTE**

The version number for the master file and system file must be the same.

- $CONFIGURE—indicates if the module is to be configured into the system. If $CONFIGURE is Y or y, the module will be configured on the next build; if $CONFIGURE is N or n, the module will not be configured on the next build. kmsystem (1M) provides the interface to modify the flag.

- $LOADABLE—indicates how the module will be configured. If $LOADABLE is Y or y, the module will be configured as a Dynamically-configured Loadable Module; if $LOADABLE is N or n, the module will be statically configured into the kernel, requiring a reboot. kmsystem provides the interface to modify the flag.

- If $CONFIGURE is N or n, $LOADABLE is ignored.

- $TUNABLE (empty)—place holder for any tunable parameter specified in the associated master file for which you want to specify a value other than the default value. Nothing is entered here.
kmtune (1M) is the interface to modify tunable parameters in the module's system description file and the HP-UX system file (/stand/system by default).

**Modstub.o File Definition**

An optional component, the `Modstub.o` file is statically-configured into the kernel as a “place holder” for functions implemented in a loadable module that will be loaded at a later time. Its purpose is to enable the kernel to resolve references to the absent module’s functions. Configuring a module that uses stubs requires a full kernel build so that the stubs can be statically linked to the kernel.

`Modstub.o` contains stubs for entry points defined in the associated loadable module that can be referenced by other statically-configured kernel modules currently configured in the system. Access to a stub causes the kernel to auto load the associated loadable module.

**space.h File Definition**

An optional component, the `space.h` file contains storage allocations and initialization of data structures associated with a DLKM module. When the size or initial value of the data structures depend on configurable values such as tunable parameters. In order to communicate these values to the rest of the DLKM module, the values are stored in global variables and accessed by the module via external declarations in the module's `mod.o` file.

**NOTE**

All tunable parameters specified in the `master` file are defined as global variables in the `space.h` file.

**STREAMS Drivers**

Initialization of STREAMS drivers is very similar for both the loadable and statically-configured module cases. The only difference is that loadable drivers must use the `drv_info_t` structure that is passed as an argument to the `_load()` function.

STREAMS drivers, like WSIO class drivers, automatically track `open()` and `close()` system calls for the STREAMS device. The system will prevent a STREAMS driver from unloading whenever the device has one or more open file handles. Of course, the driver can still disallow an unload if this check is insufficient for its needs.
Miscellaneous Modules

Miscellaneous modules can implement any feature within the kernel. As such, a miscellaneous module's _load() function must address all of the module's specific needs. Similarly, the module's _unload() function must determine for itself if it is safe to unload. The system will not allow a module to be unloaded if other loaded modules are dependent upon the module. Other than this check, the system performs no other checks when the administrator attempts to remove a miscellaneous module from the kernel.

The argument to the _load() function is not meaningful and should be ignored.

DLKM Tools

There are a number of HP-UX commands known collectively as the kernel configuration tool set for installing, configuring, and managing DLKM modules. These commands are presented with descriptions and applicable command line options in this section.

Why you should use the kernel configuration tools instead of manually editing the system files

Although the HP-UX static kernel environment has not changed, it is affected by the configuration of kernel modules within the DLKM infrastructure. Specifically, DLKM requires that a kernel module have its own master and system files, and contain a Module Wrapper.

To the overall HP-UX kernel configuration environment this means:

1. The configurable module information is distributed among several files:
   - traditional modules use the /stand/system file
   - modularly-packaged modules use their own module-specific system file

2. The kernel structure is extended:
   - static kernel executable file /stand/vmunix
Dynamically Loadable Kernel Modules

Managing Dynamically Loadable Kernel Modules

- associated DLKM kernel components under /stand/dlkm:
  - kernel symbol table
  - dynamic loadable modules

Because of the effects that the DLKM infrastructure has on the overall kernel configuration environment, it is best to configure any type of kernel module using the tools described in this section.

_Avoid editing the system file, or replacing the kernel file manually, as doing so increases the chance of introducing configuration errors._

For more detailed information regarding the master and system files, refer to the _master_ (4) manpage and the _config_ (1M) manpages.

**Kernel Configuration Tools Description**

The system administrator uses the kernel configuration tools to install, configure, load, unload, update, or remove kernel modules from the system; and to build new kernels. You can use the commands described in this tool set to configure kernel modules of any type (static or loadable).

The action carried out by a kernel configuration tool depends upon the options you specify during the tool's invocation. This information is presented in the section _Commands and Options in the Kernel Configuration Tool Set._

The following list describes the basic function of each of the commands that make up the kernel configuration tool set.

**Tools For Building Static or Dynamic Kernels**

- _kmsystem_ (1M)
  
  Provides interface to set a module’s configurable attributes, to indicate whether a module should be configured, and whether it should be built as loadable or static.

- _kmtune_ (1M)
  
  Provides interface to set the tunable parameters
Dynamically Loadable Kernel Modules
Managing Dynamically Loadable Kernel Modules

- **kmupdate (1M)**
  Updates the system with the newly built kernel and/or associated DLKM files.

**Tools That Provide an Interface to DLKM**

- **kmininstall (1M)**
  Install, remove, or update a module’s component files on a system

- **kmadmin (1M)**
  Provides general administrative interface for DLKM. Allows administrators to load, unload, and query loadable modules.

**Commands and Options in the Kernel Configuration Tool Set**

This section the command line options with descriptions for each of the kernel configuration tools.

---

**NOTE**

If you need further information regarding the functionality, usage, or command line options for any of the kernel configuration tools, refer to their respective manpages.

---

**Table 12-3 Kernel Configuration Tool Set**

<table>
<thead>
<tr>
<th>Tool/Command</th>
<th>Action</th>
</tr>
</thead>
</table>
| config       | • First form—generates both the static kernel and associated Dynamically-configured Loadable Modules; a system reboot is necessary.  
• Second form, `-M` option—generates the specified loadable module for use with the currently running kernel. The newly configured service is available immediately, without requiring a system reboot. |
### Table 12-3   Kernel Configuration Tool Set (Continued)

<table>
<thead>
<tr>
<th>Tool/Command</th>
<th>Action</th>
</tr>
</thead>
</table>
| kmadmin      | -k option—prints a list of all statically-configured modules in the running kernel.  
|              | -L option—loads the specified loadable module into the running kernel.  
|              | -Q, -q option—prints the status of the specified loadable module.  
|              | -S, -s option—prints the status of all currently loaded or registered loadable modules.  
|              | -U, -u option—unloads the specified loadable module from the running kernel.  
| kmininstall  | -a option—adds a module’s component files to certain subdirectories of /usr/conf and /stand.  
|              | -d option—deletes a module’s component files from the subdirectories of /usr/conf and /stand.  
|              | -u option—copies a module’s updated component files into the subdirectories of /usr/conf and /stand.  |
Table 12-3 Kernel Configuration Tool Set (Continued)

<table>
<thead>
<tr>
<th>Tool/Command</th>
<th>Action</th>
</tr>
</thead>
</table>
| kmsystem     | • `-c` option—assigns a value (Y or N) to the configuration (`$CONFIGURE`) flag of the specified module in preparation for the next system configuration.  
• `-l` option—assigns a value (Y or N) to the loadable (`$LOADABLE`) flag of the specified module in preparation for the next system configuration.  
• `-q` option—prints the values of the configuration and loadable flags of the specified module. Prints a “-” (signifies “does not apply”) for the loadable flag of a static module.  
• no options or `-S` option only—prints the values of the configuration and loadable flags of all modules. Prints a “-” for the loadable flags of static modules. |
| kmtune       | • `-l` option—prints the values of all system parameters.  
• `-q` option—queries the value of the specified system parameter.  
• `-r` option—resets the value of the specified parameter to its default value in preparation for the next system configuration.  
• `-s` option—assigns a value to the specified system parameter in preparation for the next system configuration. |
Table 12-3  Kernel Configuration Tool Set (Continued)

<table>
<thead>
<tr>
<th>Tool/Command</th>
<th>Action</th>
</tr>
</thead>
</table>
| kmupdate     | • First form—prepares the system to move the specified static kernel and its associated files to the /stand/vmunix file and /stand/dlkm directory, respectively, during the next system shutdown and startup.  
|              | • Second form, -M option—moves the configured image of the specified loadable module to the location where the DLKM loader can find it, and registers the module with the kernel either (1) immediately or (2) later at system shutdown. |

DLKM Procedures for Dynamically Configured Loadable Modules

This section provides detailed procedures for configuring, loading, and unloading DLKM Enabled kernel modules. Procedural information is shown in three different ways. The first two are summary formats and the third provides detailed procedure steps.

1. DLKM Procedural Flowchart

   Use this chart as a reference to view all of the procedures and to determine the correct sequence in which to perform them.

2. Tables of Loadable Module Configuration and Management Procedures

   These tables group the procedures into 3 phases: Preparing, Loading, and Maintaining procedures. There is one table for each Loadable Module type: Dynamically-configured and Statically-configured.

3. DLKM Procedures

   This section presents step-by-step instructions for preparing, configuring, loading and unloading (or activating) loadable modules.
The detailed procedure steps are presented in two sections:

a. Dynamically-configured Loadable Module Procedures

b. Statically-configured Loadable Module Procedures
Managing Dynamically Loadable Kernel Modules

Figure 12-1  DLKM Procedural Flowchart

Start

Dynamically-configured Loadable Module

Dynamically or Statically Configured?

Statically-configured Loadable Module

Prepare module as Dynamically Configured Loadable Module using the command: `kmsystem -c Y -l Y`

Prepare module as Statically Configured Loadable Module using the command: `kmsystem -c Y -l N`

OPTIONAL: Tune system parameter(s) supplied by module or static kernel using the command: `kmtune -s`

Configure loadable module into system using command: `config -M`

Move loadable module's image into place and register module using command: `kmupdate -M`

If necessary, create device special file(s) for loadable module using command: `mknod [NTA] To manage`

Load loadable module using command: `kmadmin -l`

OPTIONAL: Query loadable module using command `kmadmin -q`

OPTIONAL: Unload loadable module using command: `kmadmin -U`

OPTIONAL: Remove module's components from system using command: `kminstall -d`

Configure statically linked module into system by building new kernel using command: `config /stand/system`

Prepare system to move new kernel into place during next system shutdown and startup using command: `kmupdate /stand/build/vmunix_test`

Activate statically linked module by booting new kernel using command: `shutdown -r`

OPTIONAL: Query statically linked module using command: `kmadmin -k`

If necessary, create device special file(s) for statically linked module using command: `mknod`

Done
### Table 12-4  Dynamically-configured Loadable Module Procedures

<table>
<thead>
<tr>
<th>Phase</th>
<th>Configuration Option</th>
<th>Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preparing</td>
<td>Prepare Loadable Module as a Dynamically-configured Loadable Module</td>
<td>Prepare a loadable module for dynamic loading into the HP-UX kernel</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Optional: Query and/or Tune the system parameters supplied by a loadable module</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Configure a loadable module for dynamic loading</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Register a Dynamically-configured Loadable Module with the kernel</td>
</tr>
<tr>
<td>Loading</td>
<td>Demand-Load</td>
<td>Load a Dynamically-configured Loadable Module into the kernel</td>
</tr>
<tr>
<td>Maintaining</td>
<td>Unload</td>
<td>Unload a Dynamically-configured Loadable Module</td>
</tr>
<tr>
<td></td>
<td>Tune</td>
<td>Tune a Dynamically-configured Loadable Module</td>
</tr>
<tr>
<td></td>
<td>Update a module</td>
<td>Update a Dynamically-configured Loadable Module’s image</td>
</tr>
<tr>
<td></td>
<td>Query a module</td>
<td>Determine which Dynamically-configured Loadable Modules are currently loaded</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Obtain information about a loaded Dynamically-configured Loadable Modules</td>
</tr>
</tbody>
</table>
All DLKM modules that are required to boot the kernel must be configured as *statically* configured modules.

If the module you are configuring is required to boot the kernel, refer to the configuration procedure in the section *Statically-configured Loadable Modules*.  

<table>
<thead>
<tr>
<th>Phase</th>
<th>Configuration Option</th>
<th>Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preparing</td>
<td>Prepare Loadable Module as a Statically-configured Loadable Module</td>
<td>Prepare a loadable module for static linking to the HP-UX kernel</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Optional: Query and/or Tune the system parameters for a Statically-configured Loadable Module present in the Static Kernel</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Configure Kernel to include Statically-configured Loadable Module</td>
</tr>
<tr>
<td>Loading</td>
<td>Activate a Statically-configured Loadable Module</td>
<td>Activate a Statically-configured Loadable Module by rebooting</td>
</tr>
<tr>
<td>Maintaining</td>
<td>Tune a module</td>
<td>Tune a loadable module</td>
</tr>
<tr>
<td></td>
<td>Query a module</td>
<td>Determine which Statically-configured Loadable Module are currently loaded</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Obtain information about a currently loaded Statically-configured Loadable Module</td>
</tr>
</tbody>
</table>

Table 12-5  
**Statically-configured Loadable Modules Procedures**
How to prepare a loadable module for dynamic loading into the HP-UX kernel

Use the `kmsystem` command to assign values (`Y` or `N`) to the configuration (`$CONFIGURATION`) and loadable (`$LOADABLE`) flags in the module's system description file. If the loadable flag is not present in the system description file and you attempt to assign it a value, `kmsystem` exits with an error.

You can use the `kmsystem` command to prepare a DLKM module for configuration as either (1) *dynamically-configured* or (2) *statically-configured*.

To prepare a loadable module to be dynamically loaded into the kernel, do the following:

**Step 1.** Execute this `kmsystem` command:

```
/usr/sbin/kmsystem -c Y -l Y module_name
```

How to query and tune the system parameters supplied by a loadable module

Use the `kmtune` command to query, set, or reset system (tunable) parameters used by the DLKM module or the static kernel. `kmtune` reads the master configuration files, the system description files, and the HP-UX system file.

For a Modularly packaged Module, `kmtune` writes any user-modified system parameter to the module's system description file. For a Traditionally-packaged module using pre-11.0 module packaging, `kmtune` writes any user-modified system parameter to the HP-UX system file.

**Step 1.** To query the value of a specific system parameter, execute this `kmtune` command:

```
/usr/sbin/kmtune -q system_parameter_name
```

**Step 2.** To set the value of a specific system parameter, execute this `kmtune` command:

```
/usr/sbin/kmtune -s system_parameter_name=value
```

**Step 3.** To reset the value of a system parameter to its default value, execute this `kmtune` command:

```
/usr/sbin/kmtune -r system_parameter_name
```
At this point, you have set the values of the module’s system parameters for the next module configuration. The values of the system parameters supplied by the module will become effective with the running kernel after the loadable module is configured and registered (see procedures on following page).

### How to configure a loadable module for dynamic loading

Upon completing the configuration procedure shown here, the dynamically-configured loadable module will be ready to load *immediately*, meaning that you do not have to wait for a reboot to be able to load it.

**Step 1.** To configure a loadable module for dynamic loading, execute this `config` command:

```
/usr/sbin/config -M module_name -u
```

This results in the generation of a loadable image. The `-u` option forces `config` to call the `kmupdate` command, which causes the system to move the newly generated image into place and register it with the running kernel.

### How to register a dynamically-configured loadable module with the HP-UX kernel.

For a DLKM module configured as dynamically loadable, you use the `kmupdate` command to update its image and register it with the kernel. Updating a dynamically-configured loadable module’s image means moving its image into place and registering it with the kernel either (1) immediately or (2) later at system shutdown.

Call `kmupdate` after first calling `config`. If you include the `-u` option in the `config` invocation, there is no need to invoke `kmupdate`. The `config -M -u` command automatically invokes `kmupdate`.

**Step 1.** To update the image of a dynamically-configured loadable module *immediately*, execute this `kmupdate` command:

```
/usr/sbin/kmupdate -M module_name -i
```

After updating the specified module and assuming the module was loaded originally, `kmupdate` will reload the module before exiting.

**Step 2.** To update the image of a dynamically-configured loadable module *at system shutdown*, execute the following `kmupdate` command:

```
/usr/sbin/kmupdate -M module_name -a
```
If you do not specify the -i or -a option, kmupdate will attempt to update the specified loadable module immediately. If the module cannot be updated immediately (for example, the current module is in use and cannot be unloaded), the module will be updated at system shutdown.

To load a dynamically-configured loadable module into the HP-UX kernel, you use the -L option of the kmadmin command. The load operation initiated by the kmadmin -L command performs all tasks associated with link editing the module to the running kernel and making the module accessible to the system.

Specifically, the load operation performs the following tasks:

- checks what other modules the loadable module depends upon and automatically loads any such module that is not currently loaded
- allocates space in active memory for the specified loadable module
- loads the specified loadable module from the disk and link-edits it into the running kernel
- relocates the loadable module’s symbols and resolves any references the module makes to external symbols
- calls the module’s _load() entry point to do any module-specific initialization and setup
- logically connects the module to the rest of the kernel, which is often accomplished with the help of module type-specific installation functions accessed through the module’s wrapper code

**Step 1.** To load a dynamically-configured loadable module into the running kernel, execute the following kmadmin command:

```
/usr/sbin/kmadmin -L module_name
```

When the loading completes, an identifier (ID) number prints on the standard output to identify the module that was loaded.

If you want the system to automatically load certain dynamically-configured loadable modules immediately after every system reboot, add the names of the modules to the /etc/loadmods file.

At boot time, the /sbin/init.d/kminit script will execute the kmadmin command and load the modules listed in /etc/loadmods.
How to unload a dynamically-configured module

Use the \(-U\) or \(-u\) 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

**Step 1.** To unload a dynamically-configured loadable module by name, execute this \texttt{kmadmin} command:

\[
/usr/sbin/kmadmin \ -U \ module\_name
\]

**Step 2.** To unload a dynamically-configured loadable module by ID number, execute this \texttt{kmadmin} command:

\[
/usr/sbin/kmadmin \ -u \ module\_id
\]

How to determine which modules are currently loaded

Use the \(-S\) or \(-s\) option of the \texttt{kmadmin} command to view detailed information about all current registered DLKM module.

**Step 1.** To print the full status for all dynamically-configured loadable modules currently registered, execute this \texttt{kmadmin} command:

\[
/usr/sbin/kmadmin \ -S
\]

**Step 2.** To print the brief status for all dynamically-configured loadable modules currently loaded, execute this \texttt{kmadmin} command:

\[
/usr/sbin/kmadmin \ -s
\]

**Step 3.** To print a list of all statically-configured modules, execute the following \texttt{kmadmin} command:

\[
/usr/sbin/kmadmin \ -k
\]

How to obtain information about a loaded module

Use the \(-Q\) or \(-q\) option of the \texttt{kmadmin} command to view detailed information about the DLKM module. For a DLKM module configured as dynamically loadable, you have the choice of displaying information for the module by its name or ID number.
Step 1. To display a dynamically-configured loadable module’s status by name, execute this `kmadmin` command:

```
/usr/sbin/kmadmin -Q module_name
```

Step 2. To display a dynamically-configured loadable module’s status by ID, execute the following `kmadmin` command:

```
/usr/sbin/kmadmin -q module_id
```

Depending on the type of module, information on the module’s block major number, character major number, and flags may also be printed.

Information returned by the `-Q` and `-q` options includes:

- the module’s name
- 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 currently depend upon this module being loaded; depended upon modules are specified in the `$DRIVER_DEPENDENCY` section of the module’s 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`)
DLKM Procedures for Statically Configured Loadable Modules

How to prepare a loadable module for static linking

You can use the `kmsystem` command to prepare a DLKM module for configuration as either (1) *dynamically loadable* or (2) *statically-configured*.

Use the `kmsystem` command to assign values (Y or N) to the configuration ($CONFIGURE) and loadable ($LOADABLE) flags in the module's system description file. If the loadable flag is not present in the system description file and you attempt to assign it a value, `kmsystem` exits with an error.

**Step 1.** To prepare a DLKM module for static linking to the HP-UX kernel, execute this `kmsystem` command:

```
/usr/sbin/kmsystem -c Y -l N module_name
```

How to query and tune the system parameters for a statically-configured loadable module present in the static kernel

Use the `kmtune` command to query, set, or reset system (tunable) parameters used by the DLKM module or the static kernel. `kmtune` reads the master configuration files, the system description files, and the HP-UX system file.

For a Modularly-packaged module or a Traditionally-packaged module using 11.0 module packaging, `kmtune` writes any user-modified system parameter to the module's system description file. For a Traditionally-packaged module using pre-11.0 module packaging, `kmtune` writes any user-modified system parameter to the HP-UX system file.

To query the value of a specific system parameter, do the following:

**Step 1.** Execute this `kmtune` command:

```
/usr/sbin/kmtune -q system_parameter_name
```

**Step 2.** To set the value of a specific system parameter, execute this `kmtune` command:

```
/usr/sbin/kmtune -s system_parameter_name=value
```

**Step 3.** To reset the value of a system parameter to its default value, execute this `kmtune` command:

```
/usr/sbin/kmtune -r system_parameter_name
```
At this point you have set the values of system parameters that will take effect after the next whole HP-UX kernel configuration, update and system reboot (see procedures below).

How to configure the HP-UX kernel to include a statically-configured loadable module

You can use the config command to configure a DLKM module into the system as either dynamically loadable or statically-configured. Use this procedure to statically link the DLKM module to a new kernel.

To configure the HP-UX kernel to include a statically-configured loadable module, do the following:

Step 1. Execute this config command:

```
/usr/sbin/config -u /stand/system
```

config builds a new kernel. The -u option forces config to call the kmupdate command, which causes the system to perform the following actions when you shutdown and restart the system:

a. save the existing kernel file and its kernel function set directory as /stand/vmunix.prev and /stand/dlkm.vmunix.prev, respectively

b. move the newly generated kernel file and its kernel function set directory to their default locations, /stand/vmunix and /stand/dlkm, respectively

After the system reboots, your DLKM module will be available as statically-configured in the new running kernel.
13 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 137, Chapter 5.
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.
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 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 137, 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:

```c
typedef int int32_t;
typedef unsigned int uint32_t;
```

Consider the following data structure from the 10.20 header file diskio.h:

```c
typedef struct {
    long lba; /* capacity in DEV_BSIZE blocks */
} capacity_type;
```

Contained in the structure is the data member lba declared as long. This is an application visible data structure that should be fixed in size to avoid compatibility problems between a 32-bit application and a 64-bit driver. For example,

```c
typedef struct {
    int32_t lba; /* capacity in DEV_BSIZE blocks */
} capacity_type;
```

Since lba is intended to store values greater than or equal to zero and negative values are never stored, the correct declaration for lba is an unsigned type. The following is the 64-bit clean version of capacity_type in release 11.0:

```c
typedef struct {
    uint32_t lba;
    /* capacity in DEV_BSIZE blocks */
} capacity_type;
```

intptr_t and uintptr_t

The data types intptr_t and uintptr_t represent storage that must scale in size with a pointer data type. They are declared in the header file _inttypes.h as:

```c
typedef long intptr_t; typedef unsigned long uintptr_t;
```

Consider the following macro from the 10.20 header file cpu.h:
#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 137, 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;
}
#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;

#define __LP64__ is used to pad the data structure so that the size is 64 bytes in both ILP32 and LP64 modes.

**Hardware Considerations**

I/O drivers often map data structures into I/O hardware registers and memory areas that are accessed by I/O hardware. Care must be exercised to ensure that these structures are fixed in size when compiled with LP64 data types.

Consider the data structure `compl_head` in the 10.20 header file `llio.h`:

```c
struct compl_head {
    int sema;
    struct compl_entry *link;
    int filler[2];
};
```

A `compl_head` structure is updated by hardware when an I/O request completes. The `link` data member is viewed by hardware as a 32-bit value. With LP64, however, `link` will be scaled to 64 bits.

The first step is to fix the size of the structure. In release 11.0, the structure is declared as:

```c
struct compl_head {
    uint32_t sema;
    uint32_t link;
    uint32_t filler[2];
};
```

The next step is to examine code that accesses the `compl_head` and make corrections as needed.
Software Considerations

While data structures that map onto hardware and must be fixed in size are easily identified, data structures with software considerations can be much more subtle. 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 size of the specific message body type being sent. In LP64 mode, the calculated size of messages are 4 bytes less than the programmer intended.

A possible solution is to explicitly pad the message header as follows:

```c
typedef struct {
    shortint   msg_descriptor;
    shortint   message_id;
    int        transaction_num;
    port_num_type from_port;

    #ifdef __LP64__
        int filler; /* pad to 64-bit boundary */
    #endif /* __LP64__ */
} llio_std_header_type;
```

**ANSI Function Prototypes**

With LP64, programmers can no longer assume that `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 `stdsyms.h` as follows:

```c
#if defined(_PROTOTYPES)
#define __ (arg) arg
#else
#define __ (arg) ()
#endif
```
To illustrate the __() macro, consider the following function prototype:

```c
void *__fl__((long arg1, int arg2));
```

Note the double parenthesis surrounding the arguments. This notation is equivalent to the cumbersome form it replaces:

```c
#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:

```
#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
14 Interrupt Migration
Interrupt Migration
Introduction

At boot time, the HP-UX kernel allocates external card interrupts to CPUs in a round-robin method. The kernel is not aware of card traffic patterns at that time. The boot allocation scheme could allow two heavily loaded cards to be mapping interrupts to the same CPU. This might lead to system performance degradation.

Interrupt Migration provides a flexible mechanism for managing CPU interrupt assignments by the movement of external I/O device interrupts from one CPU to another.

The /usr/contrib/bin/intctl command allows the user to display the interrupt configuration of the system and to migrate the interrupts between CPUs.
Overview of Interrupt Migration Impact on Driver

The interrupt migration operation is a transparent one to applications. The associated cards need not be quiesced during migration between processors. The applications utilizing the cards are not affected by these operations.

Drivers must be designed to spend minimal time in their ISR routines. They may need to quiesce the card interrupts during a migration operation, but quiescing the card DMA engines is not necessary.

In the HP-UX environment, drivers can use Line Based Interrupts (LBIs) or Transaction Based Interrupts (TBIs).

LBI Drivers

Most drivers using LBIs do not need to know which CPU is handling the interrupt. For the exceptions, WSIO provides new interrupt migration related events and informs the drivers of their use. If a driver using LBI has not registered for these interrupt event flags with WSIO, the interrupts of the associated card will be migrated to a different processor without informing the driver.

TBI Drivers

Drivers using TBIs spread the interrupt load across CPUs in the system and could interrupt more than one CPU. The drivers need to program their cards with the interrupt specifics (CPU address and data vector), so any TBI interrupt migration will be processed by the driver. The TBI drivers must first register with WSIO to utilize the new events.

WSIO provides new event masks for the drivers; they will be notified when a new CPU is enabled for interrupts in the system.

Interrupt Line Sharing

According to PCI bus specifications, a device can share an interrupt line with another device, port or function. All devices sharing the same interrupt line interrupt the same CPU. Migration of any of the interrupts sharing a line results in all interrupts on that line being migrated.
Interrupt Migration Event Masks and Registration with WSIO

WSIO defines three new event flags as part of interrupt migration. Two are for TBI based drivers, and one is for LBI based drivers.

LBI Event Flags

An optional new event flag, WSIO_EVENT_LBI_MIGR, upon registration, notifies the driver when the specified Line Based Interrupt is migrated to a new processor.

WSIO_EVENT_LBI_INTR_MIGR

Drivers of cards using LBIs may register for this event flag. They do this with the wsio_reg_drv_capability_mask() call. A driver registers for events in its _attach() routine after claiming the card it controls by calling isc_claim(). The driver also needs to register for an event handler in its _install() routine. Registration for this event flag is needed only if the driver has cached the interrupt-CPU. With an interrupt migration operation initiated, the interrupting CPU changes to a different CPU. If the driver has registered for this event flag, it is notified of this CPU change through the event handler mechanism.

Example of a driver registering for WSIO_EVENT_LBI_INTR_MIGR event:

```c
    driver_attach(....) { 
        wsio_event_mask_t newmask;
        ...
        isc_claim(isc);
        newmask = oldmask | WSIO_EVENT_LBI_INTR_MIGR
        ret = wsio_reg_drv_capability_mask( isc, newmask);
        ...
    }
```
Interrupt Migration

Interrupt Migration Event Masks and Registration with WSIO

The `wsio_reg_drv_capability_mask()` call is described in the Device Driver Reference (DDR). If the driver has registered for the event flag, its event handler is invoked twice with the `WSIO_EVENT_LBI_INTR_MIGR` flag by WSIO. The first invocation notifies the associated card that the interrupt is migrating to a different CPU. The second invocation notifies the associated card that the migration is complete. The two invocations can be differentiated by the `wsio_generic_event_t` arg structure passed to the driver event handler. The element `wsio_intr_migr_info_t` inside the `wsio_generic_event_t` structure differentiates the two notifications.

```c
typedef struct wsio_generic_event {
    wsio_event_t event; // WSIO_EVENT_LBI_INTR_MIGR
    wsio_event_id_t event_id; // wsio provided event_id
    struct isc_table_type *isc; // pointer to isc
    generic_complete_callback_t wsio_completion_cb; // wsio callback
    void *arg; // pointer to wsio_intr_migr_t
} wsio_generic_event_t;

typedef struct wsio_intr_migr {
    void *intr_object; // Associated interrupt object
    spu_t new_spu_id; // '-1' or new CPU id
    // (WSIO_LBI_INTR_MIGR_NOTIFY or WSIO_LBI_INTR_MIGR_COMPLETE)
    wsio_intr_migr_info_t info; // WSIO_LBI_INTR_MIGR_NOTIFY or
    // WSIO_LBI_INTR_MIGR_COMPLETE
    wsio_intr_ret_t ret; // return value set by the driver
    void *resvd; // reserved field.
} wsio_intr_migr_t;
```

Once the driver handles the event, it invokes the WSIO provided completion callback routine to indicate completion status of the event. The code for the completion callback routine is as follows:
typedef int (*generic_complete_callback_t)
    ((struct isc_table_type *,
      wsio_event_id_t, void *));

(*wsio_completion_callback) (isc, event_id,
   wsio_intr_migr_p);

The parameters are defined as:

isc
event_id
wsio_intr_migr_p

Pointer to the ISC structure
event id
Pointer to the wsio_intr_migr that was passed to the driver event handler through the arg field of the wsio_generic_event_t structure.
The only permissible return value from the driver for this event flag is `WSIO_OK`.

**NOTE**

The driver event handler should **not** register for a timeout call to invoke the WSIO provided callback routine.

The general flow of events is shown in Figure 14-1, LBI Interrupt Migration, and the specific steps are:

1. WSIO invokes the driver’s event handler routine with the event flag as `WSIO_EVENT_LBI_INTR_MIGR`. The `arg` field of the `wsio_generic_event_t` structure contains the interrupt object for which this event is being performed. The event `info` flag is set to `WSIO_LBI_INTR_MIGR_NOTIFY`. 
Interrupt Migration

Interrupt Migration Event Masks and Registration with WSIO

2. The driver returns the completion status by invoking the
   `wsio_completion_cb` function pointer.

3. Once the driver returns, WSIO invokes low level routines (Machine
dependent layer) to migrate the line based interrupt to the new CPU.

4. The migration causes a spurious interrupt on the CPU to which the
interrupt of this card has been migrated. Drivers **must** be capable of
handling spurious interrupts.

5. When migration is complete, WSIO invokes the driver’s event
   handler with the event flag as `WSIO_EVENT_LBI_INTR_MIGR`. The `arg`
   field of the `wsio_generic_event_t` structure contains the interrupt
   object for which this event is being performed. The event `info` flag is
   set to `WSIO_LBI_INTR_MIGR_COMPLETE`. The `new_spu_id` field in the
   `arg` structure contains the value of the new CPU.

6. The driver returns the completion status by invoking the
   `wsio_completion_cb` function pointer.

**TBI EVENT FLAGS**

Two new flags are introduced as part of interrupt migration for drivers
using TBIs. The first flag, `WSIO_EVENT_OFFLINE_CPU`, is a mandatory
event, and upon registration notifies the driver when the concerned
Transaction Based Interrupt is migrated to a new processor.

The second flag, `WSIO_EVENT_ONLINE_CPU`, is an optional event, and
upon registration notifies the driver of any processor being enabled for
interrupts.

**WSIO_EVENT_OFFLINE_CPU**

This event flag is for drivers using TBIs for their cards. All drivers using
transaction based interrupts must register for the
`WSIO_EVENT_OFFLINE_CPU` event flag. Drivers register for this event flag
with the `wsio_reg_drv_capability_mask()` call. A driver registers for
events it its `_attach` routine after claiming the card it controls by calling
`isc_claim()`. The driver also must register for an event handler in its
`_install()` routine.
Interrupt Migration

Interrupt Migration Event Masks and Registration with WSIO

Example of a driver registering for WSIO_EVENT_OFFLINE_CPU event:

```c
driver_attach(....) {
    wsio_event_mask_t newmask;

    .
    isc_claim(isc);
    newmask = oldmask | WSIO_EVENT_OFFLINE_CPU.
    ret = wsio_reg_drv_capability_mask(isc, newmask);
    ..
}
```

Refer to the Device Driver Reference for a description of the `wsio_reg_drv_capability_mask()` call.

The interrupt objects are allocated in the `init()` routine. If a driver attempts to allocate a TBI and it has not registered for the WSIO_EVENT_OFFLINE_CPU event flag, WSIO fails the TBI allocation call with WSIO_ERROR.

The WSIO subsystem invokes the driver handler with the WSIO_EVENT_OFFLINE_CPU for the following scenarios:

- A CPU is being disabled or reserved for interrupts.
- The card interrupt is being bound to a different CPU

The `wsio_generic_event_t` argument for the event handler is stated as follows:

```c
typedef struct wsio_generic_event {
    wsio_event_t event;      // WSIO_EVENT_OFFLINE_CPU
    wsio_event_id_t event_id; // wsio provided event_id
    struct isc_table_type *isc; // pointer to isc
    generic_complete_callback_t wsio_completion_cb; // wsio callback
    void *arg;               // pointer to wsio_intr_migr_t
} wsio_generic_event_t;
```

```c
typedef struct wsio_intr_migr {
    void *intr_object;      // Associated interrupt object
    spu_t new_spu_id;       // `-1’ or new CPU id
    wsio_intr_migr_info_t info;  // NULL for this event
}
```

---

Chapter 14
Interrupt Migration
Interrupt Migration Event Masks and Registration with WSIO

wsio_intr_ret_t ret;  // return value set by the driver
void *resvd;         // reserved field.
} wsio_intr_migr_t;

Once the driver handles the event, it invokes the WSIO provided
completion callback routine to indicate completion status of the event.
The code for the completion callback routine is:

typedef int (*generic_complete_callback_t)
    (struct isc_table_type *,
    wsio_event_id_t, void *);
(*wsio_completion_callback) (isc,
    event_id,wsio_intr_migr_p);

isc Pointer to the ISC structure
event_id Event identification
wsio_intr_migr_p Pointer to the wsio_intr_migr that
                   was passed to the driver event
                   handler through the arg field of the
                   wsio_generic_event_t structure.

Upon return from the driver event handler, status is set in the ret field
inside the wsio_intr_migr_t structure. The permissible return values
from the driver event handler are:

WSIO_E_DRV_DEACTIVATE_CALL The deactivate call failed
WSIO_E_DRV_SET_CPU_SPEC The wsio_set_cpu_spec call failed
WSIO_E_DRV_ACTIVATE_CALL The activate call failed
WSIO_OK Everything is OK
NOTE

The driver event handler should not register for a timeout call to invoke the WSIO provided callback routine.

The general flow of events is presented in Figure 14-2, TBI Interrupt Migration, and the specific steps are:
Interrupt Migration
Interrupt Migration Event Masks and Registration with WSIO

1. WSIO invokes the driver's event handler using the WSIO_EVENT_OFFLINE_CPU flag.

2. The driver performs the following tasks to migrate the interrupt from the CPU.
   a. Disable the card interrupts; **only** the interrupt corresponding to the interrupt object passed in the wsio_generic_event_t is disabled. It does **not** have to disable all other interrupts associated with this card.
   b. Invoke the wsio_intr_deactivate() routine for the interrupt object passed in wsio_generic_event_t structure. This disables the interrupt at the higher levels. For example, the driver's ISR will be removed from the CPU's interrupt switch table. This can cause pending interrupts to be lost. To nullify these effects, a spurious interrupt is generated when the driver calls wsio_intr_enable() (described later in this sequence).
   c. Invoke the wsio_intr_set_cpu_spec() routine. This migrates the interrupt object sent in the wsio_generic_event_t structure to the new CPU. See the wsio_intr_set_cpu_spec() call for further details.
   d. The drivers detect the new CPU to which the interrupt has been migrated with the wsio_intr_get_assigned_cpu() call, and determine the transaction address and the data with the wsio_intr_get_txn_info() call.
   e. With the interrupt migrated to a new CPU, the driver invokes the wsio_intr_enable() routine to enable the interrupt object passed in the wsio_generic_event_t structure. This interface results in a spurious interrupt being fired off to the migrated CPU. Drivers **must** be capable of handling these spurious interrupts.
   f. Program the card to use the new transaction address and transaction data obtained in step d.
   g. Return the completion status (WSIO_OK) with the wsio_completion_cb() routine.
Interrupt Migration
Interrupt Migration Event Masks and Registration with WSIO

**WSIO_EVENT_ONLINE_CPU**

This event flag is for drivers using TBIs. If the driver has registered for this event, WSIO invokes the driver event handler with the **WSIO_EVENT_ONLINE_CPU** flag when it is notified that new CPUs are enabled for interrupt processing.

Drivers register for this event flag with the `wsio_reg_drv_capability_mask()` call. A driver registers for events in its `_attach()` routine after claiming the card it controls by calling `isc_claim()`. The driver also must register for an event handler in its `_install()` routine.

Example of a driver registering for a **WSIO_EVENT_ONLINE_CPU** event:

```c
driver_attach(....)
{wsio_event_mask_t newmask;

    isc_claim(isc); newmask = oldmask |
    WSIO_EVENT_OFFLINE_CPU.
    ret = wsio_reg_drv_capability_mask(isc, newmask);
    ..
    ..
}
```

See the DDR for a description of the `wsio_reg_drv_capability_mask()` call.

The `wsio_generic_event_t` argument for the event handler is written as follows:

```c
typedef struct wsio_generic_event {
    wsio_event_t event;      // WSIO_EVENT_ONLINE_CPU
    wsio_event_id_t event_id;// wsio provided event_id
    struct isc_table_type *isc; // pointer to isc
generic_complete_callback_t wsio_completion_cb;
    // wsio callback
    void *arg;               // Number of processors enabled
} wsio_generic_event_t;
```
Interrupt Migration Flow

The HP-UX environment interrupts could be either LBIs or TBIs. From WSIO perspective, the events that can lead to interrupt migration operation are:

- Changing the interrupt state of a CPU to RESERVED or DISABLED.

  When changing the interrupt state of a CPU from ENABLED to RESERVED or DISABLED, all the card interrupts associated with the CPU are redistributed in a round-robin method to the other interrupt ENABLED CPUs.

- Changing the interrupt state of a CPU to ENABLED.

  When a CPU state is changed to interrupt ENABLED, all the drivers registered for this event are informed.

- Migration of an interrupt from one CPU to another CPU through the user level command /usr/contrib/bin/intctl.

For CPU interrupt state change details refer to subsystems such as Processor Sets, ICOD (Instant Capacity on Demand), or RTE (Real Time Extension).

The following are the steps involved in migrating an LBI and a TBI.
Interrupt Migration

Interrupt Migration Flow

- **LBI processing**

  For every Line Based Interrupt:

  **WSIO Layer**

  When the driver is registered for `WSIO_EVENT_LBI_INTR_MIGR`, the driver event handler is invoked with the event information by `WSIO_LBI_INTR_MIGR_NOTIFY`.

  The driver handles the `NOTIFY` event and returns `WSIO_OK` to WSIO.

  WSIO calls the low level machine dependant routines to migrate the interrupt to the new CPU. Later, WSIO invokes the driver event handler to notify the completion event information as `WSIO_LBI_INTR_MIGR_COMPLETE` event.

  The driver handles the `COMPLETE` event and returns `WSIO_OK` to WSIO.

- **TBI processing**

  For every Transaction Based Interrupt:

  **WSIO Layer**

  WSIO invokes the driver event handler with an event flag as `WSIO_EVENT_CPU_OFFLINE`.

  Since this is a TBI, the driver performs the necessary steps for reprogramming the card with the new CPU value. Later the driver returns `WSIO_OK` to WSIO.
Assumptions and Dependencies

- The driver **must** handle spurious interrupts. The interrupts can happen even if the card is suspended or powered off.

- The routines `isrlink()`, `isrunlink()`, `wsio_intr_alloc()` and `wsio_intr_free()` are only invoked by the driver as part of handling events initiated from WSIO. This includes card online addition and removal, ioscan, DLKM load or unload, and interrupt migration. Drivers must not invoke these routines while handling non-WSIO events such as card reset. The `wsio_intr_set_cpu_spec()` and `wsio_intr_set_irq_line()` functions are not invoked for non-WSIO events.

- The routines `wsio_intr_activate()` and `wsio_intr_deactivate()` behave in different ways.

  If drivers using LBI invoke routines `wsio_intr_activate()` or `wsio_intr_deactivate()` while interrupt migration is in progress, the routines return failure.

  If interrupt migration is invoked and driver invocation with the above calls is in progress, interrupt migration will wait for the calls to complete.

  If the driver is using a TBI, the driver carries out the migration process. Part of the migration process is disabling and later enabling card interrupts. This can involve invocation of `wsio_intr_activate()`, `wsio_intr_deactivate()` and `wsio_intr_set_cpu_spec()` calls. Since the driver initiates all the processing, the drivers ensure there are no other threads calling these routines and the interrupt migration thread at the same time. WSIO does not be ensure this synchronization.

- When invoking the `wsio_intr_get_assigned_cpu()`, `wsio_intr_get_txn_info()` and `wsio_intr_get_irq_line()` routines, and interrupt migration is in progress, it is not guaranteed the values returned by these routines are the same as those assigned when interrupt migration is complete.

- Only one high availability event, such as card OLAR/D, error handling, processor OL*, ioscan, or interrupt migration will be in progress in the system at any given point in time.
Interrupt Migration

Assumptions and Dependencies
A 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 IEEE802.2 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 276, Chapter 7, 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

/* Types for 802.3 extended MIB */

typedef struct {
    int dot3StatsIndex;
    counter dot3StatsAlignmentErrors;
    counter dot3StatsFCSErrors;
    counter dot3StatsSingleCollisionFrames;
    counter dot3StatsMultipleCollisionFrames;
    counter dot3StatsSQETestErrors;
    counter dot3StatsDeferredTransmissions;
    counter dot3StatsLateCollisions;
    counter dot3StatsExcessiveCollisions;
    counter dot3StatsInternalMacTransmitErrors;
    counter dot3StatsCarrierSenseErrors;
    counter dot3StatsFrameTooLongs;
    counter dot3StatsInternalMacReceiveErrors;
} 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;
} mib_Dot5StatsEntry;
counter dot5StatsACErrors;
counter dot5StatsAbortTransErrors;
counter dot5StatsInternalErrors;
counter dot5StatsLostFrameErrors;
counter dot5StatsReceiveCongestions;
counter dot5StatsFrameCopiedErrors;
counter dot5StatsTokenErrors;
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 216, Chapter 7.
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 256, Chapter 7.
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 1</td>
<td>497 bytes, max IEEE 802.3 packet size.</td>
</tr>
<tr>
<td>SNAP8023_MTU 1</td>
<td>492 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];
}
These structures are contained in the `netinet/if_ether.h`, `netinet/if_ieee.h`, and `slio/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_PK    IEEE 802.3 packet.
SNAP8025_PKT  SNAP packet over IEEE 802.5 media.
IEEE8025_PK    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
enum 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 “STREAMS DLPI and Network Driver Overview” on page 242, chapter 7.
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_SNAP 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
Appendix B

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.
Designing a Subformatter

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 7-8 on page 282, Chapter 7, 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()`

Use of the functions above ensures that all headers look alike, thus helping users find useful landmarks to guide them through the output.

- `tl_format_write()`
- `tl_format_fprintf()`

Subformatters are free to call the output functions above for each line or for several lines of output, as appropriate.

- `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 `tl_format_write()` and `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.
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

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 594, for the format_link_nice, format_link_terse, and format_link_raw routines.

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 “AssignSubsystem ID” on page 285, Chapter 7). 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:

```c
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:

```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;
```

**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).
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

**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 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 ktrace_write or klogg_write):

```c
typedef struct {
    unsigned short hdr_len;
    short subsystemid;
    int device_id;
    tl_msg_flag_type flags;
    set_of_32 kind;
    set_of_32 class;
} tl_msg_hdr;
```
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

**Designing a Subformatter**

```c
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;
```
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

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 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.

`susbys_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 `susbys_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)
```

`get_opt parms_type` Defined in `/usr/include/fmt.h` as:

```c
typedef struct {
    int *status_ptr;
    FILE *subsys_strm;
    FILE *error_strm;
    FILE *log_strm;
    int ss_id;
    char *ss_name;
} get_opt_parms_type;
```
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

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.
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

```c
ss_n_get_opt_flag
Type of flag is defined as:

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.

```c
ss_options_ptr
 Pointer to a pointer to a data structure containing the 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 depend on the load order of subsystem shared libraries.

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 7-9 on page 301, Chapter 7, 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.

```c
void set_up_8022(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)
    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).

```
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.
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

**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.

- **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.

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.
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

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).

line1

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, -).

ReturnValue

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.
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

The following shows an example output:

[The linktype parameter goes here]-vvvvv
------------------------ 802.3 ------------------------
[The line1 parameter goes here]-vvvvvvvvvvvvv
Source : 00-00-0c-00-06-31 [I] [Cisco] LENGTH: 26
Dest : 09-00-09-00-00-01 [M] [HP Probe] TRACED LENGTH: 60
Date : Mon Dec 02 09:22:04:33390 PST 1991
< The addlinfo parameter info goes here
< 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)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hdr</td>
<td>Pointer to the standard nettl message header.</td>
</tr>
<tr>
<td>buffer</td>
<td>Pointer to data beginning at the 802.2 level. The upper layer routines typically will not format data straight from this buffer.</td>
</tr>
<tr>
<td>len</td>
<td>Length of the buffer (including 802.2, excluding any lower layer data).</td>
</tr>
<tr>
<td>linktype</td>
<td>String describing the type of link this information is carried over, plus any other MAC layer information appropriate for terse mode. For 802.3 the linktype is simply “S”; for Ethernet (not SNAP) it is “E”.</td>
</tr>
<tr>
<td>addlinfo</td>
<td>String giving other MAC or upper layer information to be displayed (blank for 802.3 and Ethernet).</td>
</tr>
</tbody>
</table>
The following shows and explains an example output:

8m probe vna request for: 15.13.106.63 from: 00-00-c0-06-31
  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 interface=[interface] field. Pass a NULL pointer to suppress displaying any interface information.
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

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 “”.

**addinfo**

(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:

```
vvvv[The linktype parameter goes here]
Received 60 bytes via 802.3 Mon Dec 02 09:22:04:33390 PST 1991
vvvv-[The interface parameter goes here]
    pid=[ICS] interface=[lan0]
[The line3 parameter goes here]-vvvvvvvvvvvvv
Dest: 09-00-09-00-00-01 Source: 00-00-0c-00-06-31 Length: 00-1a
< [The addinfo parameter info goes here]
14: fc fc 03 00 00 00 05 03 05 03 03 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 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.

```c
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 */
    ....

    /* 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){
    ret = format_link_nice(hdr, buffer, size, "802.2", temp_buf,
        ",", "");

Appendix B 603
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

/* 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,
    "802.2", "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.

#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.
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

flags

Type of flag 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;
```

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.
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

time_buffer_length
Contains the length of time_buffer not counting the null terminator byte.

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:

```c
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,
```
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.

user_acct_result  users[],
err_num *status_ptr)
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:

```c
int tl_format_fprintf(FILE *stream,
    fmt_wrt_flag_type flags,
    error_num *status_ptr,
    char *format, /* [,...] */ ...)
```

- **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().
- **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).
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

**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 non-printing 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.
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

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.

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_get_parms() 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;
} ss_N_fmt_parms_type;
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

```c
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.
- `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 588.
- `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,
                        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.
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.

char *tl_log_class(unsigned int class)

class Log class of the message.

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,
    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 OrgLine.

- **LineNo**  
  Stores the line number of OrgLine 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.
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

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.
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

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:

- Subsystem ID: Assigned to your subsystem by Hewlett-Packard (see “Assign Subsystem ID” on page 285, Chapter 7).
- 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.
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

**Designing a Subformatter**

- **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 `/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
#
# (c) Copyright Hewlett-Packard Company 1993
#
set -a    # Export all vars
exitval=0  # Anticipate success
```

618   Appendix B
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

```bash
: ${UTILS:="/usr/lbin/sw/control_utils"}
if [ ! -f $UTILS ]
then
    echo "ERROR: Cannot find $UTILS"
    exit 1
fi
.
$UTILS
set_env

: ${FILESET:="NETTL-MIN"}
: ${NETTLCONF:="$(SW_ROOT_DIRECTORY)usr/sbin/nettlconf"}
: ${NETFMT:="$(SW_ROOT_DIRECTORY)usr/sbin/netfmt"}

if [ ! -x "$NETTLCONF" ]; then
    echo "ERROR: Cannot find $NETTLCONF"
    exit 1
fi

# Subsystem A
$NETTLCONF -S -id 0 -name SUBSYSTEM_A -class 12 -kernel\  
    -lib libsubsystem_A.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`
```

Appendix B 619
How to Design a Networking Trace/Log Subformatter and a Sample Subformatter

Designing a Subformatter

```bash
62  cmd_result=$?
63  if [ $cmd_result -ne 0 ]
64    then
65      # The configuration file caused an error
66      echo "ERROR  The $NETFMT command produced following error"
67      echo "  messages while verifying configuration:"
68      echo "$cmd_output"
69    fi
70  fi
71
72  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 285.)
Shared Library Examples for the driveradmin and lanscan Commands
This appendix contains examples of the \texttt{lanscan} and \texttt{driveradmin}
shared library code and the \texttt{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 an enetlink.
*****************************************************************

#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stdsysms.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>
Shared Library Examples for the driveradmin and lanscan Commands

The lanscan's shared library sample code

```c
#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 */
    {"ETHER",       3},/* llc_flags 0x0008 */
    {"NOVELL",     4}/* llc_flags 0x0010 */
};

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.
**************************************************************************
******************************************************************************
***
*** dseret.c
******************************************************************************

* This file implements the shared library using by the enet driver.
*
* Modification History
*
* -- ------  ---------------------------------------------
* *
******************************************************************************

#if 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;
t ierrbase;
char errbuf[128];

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},
{"lapb (%d)\n",4116},
{"sd1c (%d)\n",4117},
{"t1-carrier (%d)\n",4118},
{"cept (%d)\n",4119},
{"basicIsdn (%d)\n",4120},
{"primaryIsdn (%d)\n",4121},
{"propPointToPointSerial (%d)\n",4122},
{"INVALID (%d)\n",4100}
};
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin’s shared library sample code

```c
int nlix;
} adopstatus[] = {
    "INVALID(%d)\n",4200,
    "up(%d)\n",4201,
    "down(%d)\n",4202,
    "testing(%d)\n",4203
};

/******************************************************************
global areas for sending and receiving streams 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 */
};

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;
int 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;
```
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin’s shared library sample code

```c

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;
}

miblp = (u_char *) ctrl_area + get_statistics_ack->di_stat_offset;
printf((catgets(catfd, 1, 105, "Description                     = %s
")), miblp->ifDescr);
printf((catgets(catfd, 1, 106, "Type (value)                    = 
")), iftypes[miblp->ifType].nlix, iftypes[miblp->ifType].s);
printf((catgets(catfd, 1, 107, "MTU Size                        = %u
")), miblp->ifMtu);
printf((catgets(catfd, 1, 108, "Speed                           = %u
")), miblp->ifSpeed);
addrintp = miblp->ifPhysAddress.o_bytes;
addrshortp = &miblp->ifPhysAddress.o_bytes[4];
printf((catgets(catfd, 1, 109, "Station Address                 = 0x%hx%04x\n")), *addrintp, *addrshortp);
printf((catgets(catfd, 1, 110, "Administration Status (value)  = 
")), adopstatus[miblp->ifAdmin].nlix, adopstatus[miblp->ifAdmin].s);
printf((catgets(catfd, 1, 111, "Operation Status (value)        = 
")), adopstatus[miblp->ifOper].nlix, adopstatus[miblp->ifOper].s);
printf((catgets(catfd, 1, 112, "Last Change                     = %u
")), miblp->ifLastChange);
```
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin's shared library sample code

```c
printf((catgets(catfd,1,113, "Inbound Octets                  = %u\n")), miblp->ifInOctets);
printf((catgets(catfd,1,114, "Inbound Unicast Packets         = %u\n")), miblp->ifInUcastPkts);
printf((catgets(catfd,1,115, "Inbound Non-Unicast Packets     = %u\n")), miblp->ifInNUcastPkts);
printf((catgets(catfd,1,116, "Inbound Discards                = %u\n")), miblp->ifInDiscards);
printf((catgets(catfd,1,117, "Inbound Errors                  = %u\n")), miblp->ifInErrors);
printf((catgets(catfd,1,118, "Inbound Unknown Protocols       = %u\n")), miblp->ifInUnknownProtos);
printf((catgets(catfd,1,119, "Outbound Octets                 = %u\n")), miblp->ifOutOctets);
printf((catgets(catfd,1,120, "Outbound Unicast Packets        = %u\n")), miblp->ifOutUcastPkts);
printf((catgets(catfd,1,121, "Outbound Non-Unicast Packets    = %u\n")), miblp->ifOutNUcastPkts);
printf((catgets(catfd,1,122, "Outbound Discards               = %u\n")), miblp->ifOutDiscards);
printf((catgets(catfd,1,123, "Outbound Errors                 = %u\n")), miblp->ifOutErrors);
printf((catgets(catfd,1,124, "Outbound Queue Length           = %u\n")), miblp->ifOutQlen);
printf((catgets(catfd,1,125, "Specific                        = %u\n")), miblp->ifSpecific);

/* Are there Extended MIB statistics to be displayed? */
if (!miblp->ifSpecific)
/* No */
{
    catclose(catfd);
    return;
}

/* Display "Continue" message and wait for user input */
disp_continue();

mib2p = (int)miblp + sizeof(mib_ifEntry);
printf((catgets(catfd,1,150, "\nEthernet-like Statistics Group\n\n")), mib2p->dot3StatsIndex);
```

Appendix C  631
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);
printf((catgets(catfd,1,162, "Internal MAC Receive Errors     = %u\n")), mib2p->dot3StatsInternalMacReceiveErrors);

#ifdef 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, "Ethernet-like Collision Statistics Group\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);
#endif
```

632 Appendix C
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;

    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)
intfd;/* file descriptor */
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin's shared library sample code

{ 
    intflags = 0;/* 0 ---> get any available message */

    /*
    * zero first byte of control area so the caller can callcheck_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() */
/**
 * 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 */
            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,
                "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) {
    /*
     */

}
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin's shared library sample code

/* 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 {
/* no 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);
            }
        }
    }
}
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin's shared library sample code

```c
i++; }
return(-1);
}

#define SET_SPEED 994

int dsenet_set_special(fd, ppa, arg) int fd;
   int ppa;
   char *arg;
{
   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);
   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");
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 \n");
   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
"), 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 function that 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 <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];
byte msg_buf_3[MAX_MSG_LENGTH+1];

int termlines; /* holds the screen line numbers of the*/

/* external variable declaration */
extern int errno;

/* NLS defines */
lc *nlcatd 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
#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

c char cmdstr[80];
c char new_phys_addr[80];
c char new_rif[20];
c char new_mtusize[20];
c char new_speed[30];
c char download_file[256];
c char driver_special_arg[256];
c char tempstr[] = "\0\0\0";
c int ppa;
c 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 */
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c
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];
int ppa_count;

/* for third parties */
#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];
shl_tlib;

tslib_start()
{
    char *lang;
    char buf[20];

    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];

/* 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;
opterr = FALSE;
while (((opt = getopt(argc, argv, ":etaBb:M:M:Ss:u:ud:xx:3:")) != -1) |

switch (opt) {
    case 'e' :
        echocmd = TRUE;
    case 't':
        break;
    case 'f':
        break;
    case 'M':
        break;
    case 's':
        break;
    case 'u':
        break;
    case 'd':
        break;
    case 'x':
        break;
    case '3':
        break;
    default:
        break;
}
Shared Library Examples for the driveradmin and lanscan Commands

**The driveradmin sample code**

```c
terse = TRUE;
break;
case 'a':
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, " ");
}
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 '(': 
    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);
        }
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

    /* 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", utils_get_message(M_WAKE_UP),
            VERSION);

    /* get time in seconds, print time */
    utils_d_datime(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 */
            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, "Enter 3rd parties device filename: ");
    lcs_3rd_parties();
    third_p_options = 0;
    break;

} /* end switch */
} /* end while (TRUE) */
} /* end main() */

/********************************************************************************

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[optc-1]);
  }
  else { /* no third party option */
    exit(-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
",
        utils_get_message(L_UNABLE_TO_OPEN_DEVICE_FILE),third_p_dev_file);
      exit(-1);
    }
  }

  /***********************************************************/

#Shared Library Examples for the driveradmin and lanscan Commands
#The driveradmin sample code
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

" /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);
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);
    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;

    /* Call the DLPI to change statistics */
    get_statistics_ack = dl_get_statistics_ack_v2;
    if (get_statistics_ack)
        /* ChangeML */
        getstats_failed = 0;
    else
        /* ChangeML */
        getstats_failed = 1;

    /* Free the DLPI resource */
    dl_get_statistics_req_free_v2(get_statistics_req);
    dl_get_statistics_ack_free_v2(get_statistics_ack);

    return getStats_failed == 0 ? mib_ptr : 0;
} /* get_stats() */

 Appendix C 651
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c
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() */

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;
        return((char *) ctrl_area + phys_addr_ack->dl_addr_offset);
```
*/ end get_phys_addr() */

 /***************************************************************************/

display_phys_addr(p, len)
 u_char *p;
 int len;
 {
 int ix;

 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) { 
 printf(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);
}

/* Change the physical address on LAN Interface device using DLPI *
to send the DL_SET_PHYS_ADDR_REQ and wait for the
DL_OK_ACK */
    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_funcname, "ds%s_reset", ppa_info.dl_name);
    /* Load the shared library */
    if (!shl_load(shlib_filename, BIND_IMMEDIATE, 0)) { 
        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 "))); /* catgets 6006 */
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")), mib_ptr->ifMtu);
    mib_ptr->ifMtu = atoi(new_mtusize);
    printf((catgets(nls_lanadmin,NL_SETN,6008,
"MTU Size = %u\n")), mib_ptr->ifMtu);

    strioctl.ic_cmd = DLPI_MTU_BYTE_SIZE;
    strioctl.ic_timout = 0;
    strioctl.ic_len = sizeof(arg);
    strioctl.ic_dp = &arg;
    arg.reqtype = SET_MTU_SIZE;
    arg.vtype = INTEGER_TYPE;
    arg.value.i = atoi(new_mtusize);
    if (ioctl(fd, I_STR, &strioctl) < 0) {
        fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5008,
"I_STR failed\n"))); /* catgets 5008 */
        exit(-1);
    }

*/
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

"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() */
/* *******************************************************/

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);

    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();
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c

/* change_mtusize() */

mib_ptr->ifMtu);
} /* end change_mtusize() */

/*****************************/

change_speed()
{
    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")));  
        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_timout = 0;
    strioctl.ic_len = sizeof(arg);
    strioctl.ic_dp = &arg;
    arg.reqtype = SET_SPEED;
    arg.vtype = INTEGERTYPE;
    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);
} /* end change_speed() */
```

Appendix C 657
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

/**********************************************************************/
download(loadflag)
int loadflag;
{
   int(*dload)();
   shl_t*libptr;
   int fn_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 */
    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 */
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c
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);

/* 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() */

/***************************************************************************/
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")))
}```
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

/* catgets 4004 */
fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,4001,
   " [-m] [-M mtu_size]\n")));
/* catgets 4001 */
fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,4002,
   " [-R]\n")));
/* catgets 4002 */
fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,4003,
   " [-s] [-S speed]\n")));
/* catgets 4003 */
fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,4025,
   " [-x options] [-X options]\n")));
fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,4005,
   " [-e]\n")));
/* catgets 4005 */
fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,4006,
   " [-t]\n")));
/* catgets 4006 */
fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,4007,
   "Default: %s\n")), cmdstr);
/* catgets 4007 */
fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,4008,
   "Default: %s\n")));
/* catgets 4008 */
fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,6004, "]\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,
   "A Set new station address corresponding to PPA Number.\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 */
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c
    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, "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,4028, "options Specific options for -x/-X. \n")));
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,4019, "PPA     PPA Number of interface. Req'd if any options above are selected. \n")));
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,4020, "e     Echo commands. \n")));
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,4021, "t     Terse prompts. \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;

    /* Additional code... */
```
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c
/* 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\n");
            fprintf(stderr, "Enter 3rd parties' device filename : ");
            continue;
        }

    /* See if we can open this special device name */
    if ((temp_fd = open(third_p_dev_file, O_RDWR)) == -1) {
        fprintf(stderr, "Invalid device file name.\nPlease 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, "\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, "\n%sd\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;
        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();*/
            fprintf(stderr, "This option is not supported by driver at this time \n");
            break;
        } /* end switch */
    } /* end while */
} /* end lcs_3rd_parties() */
The driveradmin sample code

/**************************************************************************

void 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, "Unable to open device file /dev/dlpi\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, "LAN Interface Menu\n",
            utils_get_message(L_LAN_CARD_TEST_MENU), 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, "LAN Interface Menu\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 */
                lcs_get_nmid();
                break;
            case LCS_QUIT_CMD: /* exit the Diagnostic program */
                utils_quit_cmd();
                break;
            case LCS_RESET_CMD: /* reset the LAN Interface */
                break;
        }
    }
}
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) {
        printf(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) {
        printf(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)
        printf(stderr, "%s\n",
            utils_get_message(L_UNABLE_TO_CLEAR_STATISTICS_REGISTERS));
    detach_ppa();
} else
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c
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\%s\n", utils_get_message(L_LAN_CARD_STATUS_DISPLAY));
    /* get and print time */
    utils_d_datime(stdout);

    /* display the Card Instance Number */
    printf("\%s\%d\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;
        }
    }  /* end lcs_display() */
```
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c
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);

/* 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%: ", 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 = atol(ppa_str);
    }
    /* Display the statistics for this interface */
    *(dispstats))(fd, cur_ppa, termlines);

    /* Unload the shared library after we're done with it. */
    shl_unload(lib);
    detach_ppa();
} /* end lcs_display() */
```

Appendix C 667
/* 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;
    intreset_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\n",
            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 device specific reset function */
        if (reset(libptr, flag) == -1) {
            fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5100,
                "reset failed: %s\n")), shlib_filename);
            return;
        }
    }
} /* lcs_reset */
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c
/* Call the corresponding device reset, if any! */
sprintf(shlib_funcname, "ds%s_reset", ppa_info.dl_name);

/* 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
", utils_get_message(L_UNABLE_TO_RESET_LAN_CARD));
detach_ppa();
} else
    fprintf(stderr, "%s %d
", utils_get_message(L_UNABLE_TO_ACCESS_NMID), cur_ppa);
} /* end lcs_reset() */

 adverts-get the next message from a stream
 Adam                  */

int get_msg(fd)
int fd; /* file descriptor */
{
    intflags = 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
    */
```
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c
* 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);
}
    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")), err_ack->dl_errno);
                /* catgets 5109 */
                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: 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)
int fd;
int length;
int pri;
{
    /* 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);
    }
}
The driveradmin sample code

} /* 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
"))); /* catgets 5114 */
        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++) {
      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() */

*************************************************************************
detach from the current PPA
*************************************************************************
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

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 TERMINFO 1 /* returned value when asking for info in*/
/* terminfo*/

/* system functions declaration    */
char *getenv(); /* get then environment variable TERM*/

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;
} /* 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_JUN,
  L_JUL, L_AUG, L_SEP, L_OCT, L_NOV, L_DEC };

/* 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"};

Appendix C 675

Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

/* 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, "%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");
                } else {

                    lines_count = lines_count + COMMAND_DIALOG_LINES-1;
                }
            }
for ( j = 0; j < length; j++ )
cmd[j] = tolower(cmd[j]);

/* parse command*/
command_code = utils_parse_cmd(cmd,menu_type);

/* get some white spaces in msg_buf_1*/
strcpy(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 :

Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c
first_header_index  = MAIN_LAN_CMD;
last_header_index   = MAIN_VERBOUSE_CMD;
first_trailer_index = MAIN_LAN_CMD_DEF;
last_trailer_index  = MAIN_VERBOUSE_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*/
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_DRIVER_MENU_CMD));
    fprintf(stderr, "%s\n", utils_get_message(LCS_DRIVER_MENU_CMD_DEF));
    line_count++;
}
```
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
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 == ' ') ||
    (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;
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

/* 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
",
  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_syer(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 == '\t')));

if ((c != '\n') && (c != EOF) && (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*/
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c
/* 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*/
        first_field = FALSE; /* not first field now*/
    else { /* not first field on input line*/
        fprintf(stderr, /* echo with new line*/
            "%s\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("%s\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*/
    /*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 message 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*/
    command_match = FALSE;

    /* if strict equality between cmd and msg, return index*/
    if (((command_match == TRUE) && (cmd_len == msg_len ))
        return(index);

    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 */
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

/* 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++;

} /* end while (index <= last_header_index) */

if (third_p_display_menu) {
    /* get command in the message catalog */
    pt_msg = utils_get_message(LCS_THIRD_PARTY_MENU_CMD);
    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) ) /* 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;
}
} /* end if third_p_display_menu */

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() */

/*******************************************************/
void
utils_d_datime(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 */
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

clock = time((dword *) 0);

/* get pointer on date and time structure*/
ptdateler = 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,
        /* 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_datime() */

/******************************************************************/
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; /* nls message catalog */
    char *default_message; /* message to use if nls file is not there */
    #define L_DIAG_BASE 7000   /* 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)
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;
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

def get_mac_type():
    """
    else if (get_msg(fd) == -1)
        get_info_failed = TRUE;
        else if (check_ctl(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);
        }
        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_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,5022,
                "Unable to get RIF flag: errno = %d\n")), errno);
            exit(1);
        } else {
            printf(catgets(nls_lanadmin,NL_SETN,6007,"lan%d Source Routing
                = "), cur_ppa);
            if (rif_flag == TURN_ON) /* Source routing is on */
Appendix C 689

Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c
printf (catgets (nls_lanadmin,NL_SETN, 6008, "%s\n"), STR_ON);
else if (rif_flag == TURN_OFF) /* Source routing is off */
    printf (catgets (nls_lanadmin,NL_SETN, 6008, "%s\n"),
    STR_OFF);
else
    printf (catgets (nls_lanadmin,NL_SETN, 6009, "%x\n"), rif_flag);
}

return (rif_flag);
} /* 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';
    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); /* prakashr : INDaa27035 : Not an error condition, return 0
```
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c
/*
 */

/* 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;
    interror;

    /* 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;
}

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.
")));
        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.
")));
        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. */
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:
   *     "/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>_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, TYPEPROCEDURE, 
&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")));
    }

    /* The name of the shared library that contains the function to 
     * do the firmware download associated with this device is:
     *     "/usr/lib/lanadmin/libds<driver_name>.sl"
     * OR "/usr/lib/lanadmin/libds<driver_name>.sl" */

    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)) != -1) {
      /* The name of the shared library function to do the firmware 
       * download associated with this device is:
       *     "ds<driver_name>_get_special" */
      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, TYPEPROCEDURE, 
&driver_special) < 0)
        shl_unload(lib);
      else fn_found = 1;
    }
  }

  if (!fn_found) {
    fprintf(stderr, (catgets(nls_lanadmin,NL_SETN,5116,
"This option is not supported for this hardware.\n")));
  }
} /* end get_driver_special() */
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c
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"))); 
    detach_ppa();
    exit(-1);
}

/* Load/unload firmware for this interface */
error = (*(driver_special))(fd, ppa_info.dl_ppa, driver_special_arg, 
&ret);
/* Print return string on success */
if (!error)
puts(ret);
/* Unload the shared library after we're done with it. */
shl_unload(lib);
if (error)
exit(error);
} /* end get_driver_special() */

/***************************************************************************/

menu_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: 
* 
"/usr/lib/lanadmin/libdl<driver_name>.sl"
* OR "/usr/lib/lanadmin/libds<driver_name>.sl" */
```
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```c
if (attach_ppa(cur_ppa) == -1)
{
    fprintf(stderr, "%s %d\n",
    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 */
    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(-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. */
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 */
#define NULL 0

/* error codes    */
#define NO_SYS_ER 0 /* code returned by system calls when */
#define SYS_ER -1 /* code returned by system calls when */
#define NO_ERROR 0 /* diagnostic function return code if no */
#define FILE_ERROR -2 /* returned by function when an error */
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

/* 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 */

/* 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 L_CLEARING_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

```
#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_DEF_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_MENU_CMD     7042
#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
```
Shared Library Examples for the driveradmin and lanscan Commands

**The driveradmin sample code**

```c
void break_key();
void lcs_get_name();
void lcs_reset();
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

SHARE_LIB the driveradmin and lanscan Commands

Appendix C

The driveradmin sample code

#define MODULEID
#if defined(MODULEID) && !defined(lint)
static char rcsid[]="@(#) lanadmin_mg.c: PHNE_15969 98/09/15";
#endif

char *admin_msgtable[] = {
    "        ", /* catgets 7000*/
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

```
"    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
", /* catgets  7021*/
"Enter command: ", /* catgets  7022*/
"Ambiguous command, try again.", /* catgets  7023*/
"Unrecognized command, try again.", /* catgets  7024*/
"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*/
" = 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 */
" Mar", /* catgets 9009 */
" Apr", /* catgets 9010 */
" May", /* catgets 9011 */
" Jun", /* catgets 9012 */
" Jul", /* catgets 9013 */
" Aug", /* catgets 9014 */
" Sep", /* catgets 9015 */
Shared Library Examples for the driveradmin and lanscan Commands

The driveradmin sample code

\verbatim
" Oct", /* catgets 9016 */
" Nov", /* catgets 9017 */
" Dec" /* catgets 9018 */
};

###########################################################
# MAKEFILE for enetadmin tool and the its shared library.
###########################################################

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)

OBJS = enetadmin.o enetadmin_mg.o
LIBS = /usr/lib/libHcurses.a

default all: enetadmin libdsenet.1
enetadmin: $(OBJS)
$(CC) $(CFLAGS) -o enetadmin $(OBJS) $(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

#include <stdio.h>
#include <ctype.h>
#include <fcntl.h>
#include <signal.h>
#include <netio.h>
#include <nl_types.h>
#include <locale.h>
#include <unistd.h>
#include <sys/stdsysms.h>
#include <sys/stat.h>
#include <sys/libIO.h>
#include <sio/llio.h>
#include <sys/stropts.h>
#include <sys/mib.h>
#include <sys/stream.h>

# 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

/**************************** enetlinkloop.c**************************/
*MODULE:  enetlinkloop
*
*DESCRIPTION:
*This user-space program checks LAN connectivity by
*writing TEST frames to specified nodes.
*DEPENDENCIES:
*This program requires DLPI.
*
******************************************************************/
Shared Library Examples for the driveradmin and lanscan Commands

The enetlinkloop's sample code

```c
#include <sys/dlpi.h>
#include <sys/dlpi_ext.h>

extern char*sys_errlist[];
extern interrno;
extern char*setlocale();
extern char*getenv();
char*catgets();
voidnls_start();
voidtimeout_hdlr();
intmyintr();

#define NL_SETN1
#define NATIVE_COMPUTER "C"
#define ADDRSIZE6
#define FCADDRSIZE3/* Fibre Channel (FC) N_Port Address Size */
#define FC_TYPE4 4 /* Fibre Channel (FC) Linkloop type */
#define BUFSIZE65280
#define MAX_802_31497
#define MAX_802_54135
#if define NOCODE
#define MAX_FDDI4475
#else
#define MAX_FDDI4000
#endif
#define MAX_FC65280/* Max. user specifiable pkt size for FC */
#define SEND_SAP0x80
#define MAX_INT0x7FFFFFFF
#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_U16 /* max user input for rif */
#define MAX_RIF_SIZE18 /* max rif size - user + 2 control bytes */

u_char riftmp[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;

intfd;
inttimeout;/* timeout (in seconds) */
intaddrsize;

#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, 0xc2, 0x2c, 0xac, 0x6c, 0xec,
  0x1c, 0x9c, 0x5c, 0xdc, 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,
  0x16, 0x96, 0x56, 0xd6, 0x36, 0xb6, 0x76, 0xf6,
  0x0e, 0x8e, 0x4e, 0xce, 0x2e, 0xae, 0x6e, 0xe6,
  0x1e, 0x9e, 0x5e, 0xde, 0x3e, 0xe6, 0x7e, 0xfe,
  0x01, 0x81, 0x41, 0xc1, 0x21, 0xa1, 0x61, 0xe1,
  0x11, 0x91, 0x51, 0xd1, 0x31, 0xb1, 0x71, 0xf1,
  0x09, 0x89, 0x49, 0xc9, 0x29, 0xa9, 0x69, 0xe9,
  0x19, 0x99, 0x59, 0xd9, 0x39, 0xb9, 0x79, 0xf9,
  0x05, 0x85, 0x45, 0xc5, 0x25, 0xa5, 0x65, 0xe5,
  0x15, 0x95, 0x55, 0xd5, 0x35, 0xb5, 0x75, 0xf5,
  0x0d, 0x8d, 0x4d, 0xc4, 0x2d, 0xad, 0x6d, 0xed,
  0x1d, 0x9d, 0x5d, 0xd4, 0x3d, 0xbd, 0x7d, 0xfd,
  0x03, 0x83, 0x43, 0xc3, 0x23, 0xa3, 0x63, 0xe3,
  0x13, 0x93, 0x53, 0xd3, 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, 0xd7, 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
************************************************************************/
#define AREA_SIZE 655280 /* 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
* 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[];
The enetlinkloop's sample code

```c
{
    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 addr */
    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; /* char ptr to the parameter of the -r option */
    int temp_ppa;
    u_char dlsap[20];
    int dlsap_len;
    int thirdp_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;
        case 't':
            timeout_str = optarg;
            break;
        case 's':
            size_str = optarg;
            break;
        case 'r':
            rif_str = optarg;
            break;
        } // end of while loop
    } // end of for loop
}
```
Shared Library Examples for the driveradmin and lanscan Commands

The enetlinkloop's sample code

```c
    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  *
    *  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, "%s\n",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\n"));
        exit(1);
      }
    }
```
/* Find the first PPA. */
if (get_ppa_list() == -1)
exit(1);

/* Was an PPA Number specified? */
if (ppa_str) {
/* 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;
}
}
Shared Library Examples for the driveradmin and lanscan Commands

The enetlinkloop's sample code

/* 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 = 3 4472 info bytes */
            /* Reserved 4:3 = 0 reserved */
        }
    } 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"));
        }
    }

    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, o[MAX_RIF_SIZE_U], riflen, tlen, n, error;

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'
"), 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);
    }

    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);
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,"")) {
Shared Library Examples for the driveradmin and lanscan Commands

The enetlinkloop's sample code

```c
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"
   "Continuing processing using the \"C\" locale.\n"
); 

#endif
#endif

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() */
```
Shared Library Examples for the driveradmin and lanscan Commands

The enetlinkloop’s sample code

/**
 * bind to a sap with a specified service mode and max_conind;
 * returns the local DLSAP and its length
 */

void 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 = (unsigned 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)
int fd;/* file descriptor */
{
    int flags = 0;/* 0 --- get any available message */

    /*
Shared Library Examples for the driveradmin and lanscan Commands

The enetlinkloop's sample code

```c
* 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);
} /* end get_msg() */

/****************************************************************************
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 */

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));
    }
    return(result);
} /* end get_msg2() */
```

Appendix C
/* 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 */
                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
")));
                fprintf(stderr, (catgets(nls_linkloop,NL_SETN,78,
                    "   dl_error_primitive = 0x%02x
")), err_ack->dl_error_primitive);
                fprintf(stderr, (catgets(nls_linkloop,NL_SETN,79,
                    "   dl_errno = 0x%02x
")), err_ack->dl_errno);
                fprintf(stderr, (catgets(nls_linkloop,NL_SETN,80,
                    "   dl_unix_errno = %d
")), err_ack->dl_unix_errno);
            }
        }
    } /* end check_ctrl() */
Shared Library Examples for the driveradmin and lanscan Commands

The enetlinkloop's sample code

```c
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 check CTRL() */

/******************************************************************
put a message consisting of only a control part on a stream
*******************************************************************/
int
put_ctrl(fd, length, pri)
int fd;/* file descriptor */
int length;/* length of control message */
int pri;/* 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);
```

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)
intfd; /* file descriptor */
intctrl_length; /* length of control part */
intdata_length; /* length of data part */
intrpri; /* 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() */

/**********************************************************
Get the list of available PPAs.
**********************************************************/
int
get_ppa_list()
{
    dl_hp_ppa_req_t *papa_req = (dl_attach_req_t *)ctrl_area;
    dl_hp_ppa_ack_t *papa_ack = (dl_hp_ppa_ack_t *)ctrl_area;
    dl_hp_ppa_info_t *papa_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 */
    papa_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);
  }
  /* 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++) {
    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) {
        /* */
        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 ]
fprintf(stderr, catgets(nls_linkloop, NL_SETN, 30, " [-s size ] [-r rif ] [-v ] [-3 thirdp_filename] linkaddr ...
exit(1);
} /* 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 == '	'))
        s++;

    /* check for sign bit */
    if (*s == '-') {
        sign = -1;
        s++;
    } else if (*s == '+') {
        sign = 1;
        s++;

    } else if (*s == '0') {
        sign = 0;
        s++;
    } else if (*s == '1') {
        sign = 1;
        s++;

    } else if (*s == '2') {
        sign = 2;
        s++;

    } else if (*s == '3') {
        sign = 3;
        s++;

    } else if (*s == '4') {
        sign = 4;
        s++;

    } else if (*s == '5') {
        sign = 5;
        s++;

    } else if (*s == '6') {
        sign = 6;
        s++;

    } else if (*s == '7') {
        sign = 7;
        s++;

    } else if (*s == '8') {
        sign = 8;
        s++;

    } else if (*s == '9') {
        sign = 9;
        s++;

    } else {
        return(FALSE);
    }

    /* skip while number is valid */
    while (*s >= '0' && *s <= '9')
        s++;

    sum = 0;
    /* consume the number */
    while (*s >= '0' && *s <= '9')
        sum = sum * 10 + (*s - '0') * sign;

    /* check for overflow */
    if (sum > max) {
        return(FALSE);
    } else {
        *i = sum;
        return(TRUE);
    }
}
Shared Library Examples for the driveradmin and lanscan Commands

The enetlinkloop's sample code

```c
} 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)
    u_char*hisaddr;
    u_char*txbuf;
    int size;
{
    dl_test_req_t*test_req = (dl_test_req_t *)ctrl_area;
    intrlen = 0;
    intfclen = 0;
    u_char *sap_p;
    u_char *fctype_p;

    /* If 802.5, factor in RIF len */
    if (ppa_info.dl_mac_type == DL_TPR)
        rlen = riftmp[0];
    else
        if (ppa_info.dl_mac_type == DL_FC)
```
Shared Library Examples for the driveradmin and lanscan Commands

The enetlinkloop's sample code

fclen = 1;

test_req->dl_primitive = DL_TEST_REQ;
test_req->dl_dest_addr_length = addrsize + rlen + fclen + 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)
{
  fctype_p = (char *)ctrl_area+test_req->dl_dest_addr_offset+addrsize;
  *fctype_p = FC_TYPE4;
}

/* Copy in the SAP */
sap_p = (char*)ctrl_area+test_req->
  dl_dest_addr_offset+addrsize+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+addrsize+1,
  riftmp, rlen);

memcpy((char *)data_area, txbuf, size);
put_both(fd, sizeof(dl_test_req_t)+addrsize+rlen+fclen+1, size, 0);
} /* end send_test_req() */

/**************************************************************************
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 */
u_char *myaddr;/* local node addr (used to compare with response) */
u_char *txbuf;/* the transmit data */
{
  int j;/* indexes through count */
  u_char *dest_addr[ADDRSIZE];
  u_char *src_addr[ADDRSIZE];
Shared Library Examples for the driveradmin and lanscan Commands

The enetlinkloop’s sample code

```
u_char hisaddr[ADDRSIZE]; /* the remote node’s link level address */

u_char hexaddr[ADDRSIZE*2+3]; /* contains hex formatted vers of link addr */

intheader_error; /* a receive is repeated if an hdr err occur */

inprinted; /* boolean/if the OK/FAILED msg already prntd */

int inmsg_res;

dl_test_con_t *test_con = (dl_test_con_t *)ctrl_area;

print = FALSE;
print((catgets(nls_linkloop, NL_SETN, 16, "Link connectivity to LAN station: %s\n")), destaddr);

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)) {
        printf((catgets(nls_linkloop, NL_SETN, 17, " -- FAILED\n"));
        printf((catgets(nls_linkloop, NL_SETN, 18, "    Address is not individual\n"));
        fflush(stdout);
        return;
    }
} else {
    if (ppa_info.dl_mac_type != DL_FC) {
        if (!INDIVIDUAL(hisaddr)) {
            printf((catgets(nls_linkloop, NL_SETN, 17, " -- FAILED\n"));
            printf((catgets(nls_linkloop, NL_SETN, 18, "    Address is not individual\n"));
            fflush(stdout);
            return;
        }
    }
}
recv_ok = 0;
recv_bad_data = 0;
recv_bad_len = 0;
recv_timeout = 0;
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;
        /* Yes */
    } continue;

    check_ctrl(DL_TEST_CON);
    if (msg_res == GOT_BOTH) {
        memcpy(dest_addr,
            (char *)ctrl_area+test_con->dl_dest_addr_offset,
            address);
        memcpy(src_addr,
            (char *)ctrl_area+test_con->dl_src_addr_offset,
            address);
        /* If Token Ring, mask out the RIF bit */
        if (ppa_info.dl_mac_type == DL_TPR)
            src_addr[0] &= 0x7f;
        if (!compare(myaddr, dest_addr, addrsize)) {
            if (verbose) {
                if (!printed) {
                    printf(catgets(nls_linkloop, NL_SETN, 17, " --
                        FAILED\n"));
                
}}
Shared Library Examples for the driveradmin and lanscan Commands

The enetlinkloop's sample code

```c
    printf((catgets(nls_linkloop, NL_SETN, 20, " %d: bad destination address %s\n")),
           j, net_ntoa(hexaddr, dest_addr, addrsize));
    recv_bad_header++;
    header_error = TRUE;

    if (!compare(hisaddr, src_addr, addrsize)) {
        if (verbose) {
            if (!printed) {
                printf(catgets(nls_linkloop, NL_SETN, 17, " -- FAILED\n"));
                printed = TRUE;
            }
            printf((catgets(nls_linkloop, NL_SETN, 21, " %d: bad source address %s\n")), j,
                   net_ntoa(hexaddr, src_addr, addrsize));
            recv_bad_header++;
            header_error = TRUE;
        }
        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++;
        }
        recv_bad_len++;
    } else if (!compare(txbuf, data_area, size)) {
        recv_bad_data++;
    } else {
        recv_ok++;
    }
    fprintf(stderr, catgets(nls_linkloop, NL_SETN, 38, "error - did not receive data part of message\n"));
    exit(1);
```
} 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();
}
} /* end loopback() */

#########################################################
# MAKEFILE for enetlinkloop
LIB =/usr/lib
CP = /usr/bin/cp
CC =/usr/bin/cc
SRCLOC=/opt/enetlinkloop
LAN_CFLAGS=-Wl,-E -l:libdld.sl -DCONVERGED_IO
INCLUDES=-I /usr/include
CFLAGS= $(LAN_CFLAGS) $(INCLUDES) -Wp,-H300000 \ 
-D_PTHREADS_DRAFT4 -L $(LIB)
LDFLAGS= +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.

Adapter Card Physical hardware, under software control, which is typically attached either directly to an I/O bus or to an auxiliary bus (e.g. SCSI) attached to a directly connected adapter. A device typically combines a hardware controller with the mechanism (e.g. disk controller with disk).

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 (\texttt{<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.

Auto load A capability made possible via the DLKM feature. It occurs when the kernel detects a particular loadable module is required to accomplish some task, but the module is not currently loaded. The kernel automatically loads the module. During an auto load, the kernel also loads any modules that the module being loaded depends upon, just as it does during a demand load.

BAR Base Address Register. On a PCI card, one of the registers in PCI configuration space that contains the size and alignment requirements needed to map the card's registers. Each 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.

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 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.

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.
Central Bus CDIO (CB-CDIO) The BN-CDIO which is responsible for discovering and initializing CEC components.

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.

Continuous DMA A type of DMA that makes a host memory buffer continuously available to an I/O device. This type of DMA is mainly used for control structures and circular queues that are shared between the device driver and the hardware device.

Core Electronics Complex (CEC) The chip set which interfaces directly to the processor in the processor-memory interconnect. In simple systems, this usually includes memory controllers and I/O Adapters. On more complex systems, it might include high-speed interconnects and coherency controllers.

D-E

DLKM Dynamically Loadable Kernel Module. A feature available in HP-UX 11.0 that supports dynamic loading and unloading of kernel modules, to avoid wasting kernel memory by keeping modules in core when they are not in use.

DLPI Data Link Provider Interface

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.

Device Driver Environment (DDE) A defined set of services and entry points which allow a driver to function.

F-I

GIO General I/O System.

High Availability (HA) Used to describe a computer system that has been designed to allow users to continue with specific applications even though there has been a hardware or software failure.

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.

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

Interrupt Service Routine A function that handles interrupts that are received for a specific device driver. A pointer to this
routine is linked to a system vector table. When an interrupt occurs, it is routed to the ISR that is placed in the section of the Interrupt Vector Table that corresponds to the received interrupt.

**Instance** A number assigned to an I/O tree node. The number is unique within a driver class.

**I/O Adapter** Hardware to provide IOVA translation between an I/O bus and the processor/memory interconnect. Devices on the I/O bus issue bus transactions to IOVA memory targets and the I/O adapter translates IOVA memory targets to physical addresses. The I/O adapter also participates in the coherency protocol of the processor caches for platforms that are coherent or semicoherent.

**I/O Bus** Interconnect bus for I/O cards and devices. PCI is an example of an I/O bus.

**I/O Node** An element of an I/O tree which includes all relevant information needed for configuring a single hardware module.

**I/O PDIR** I/O Page Directory. Address translation table associated with an I/O adapter. The I/O PDIR is analogous to the PDIR used by 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**

**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.

**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.

**LVM** The Logical Volume Manager is a disk management subsystem that offers access to filesystems as well as features such as disk mirroring, disk spanning, and dynamic partitioning.

**MAC** Medium Access Control.
Glossary

Map PCI Device/Function

Map PCI Device/Function
The act of mapping a PCI device or function involves determining the size and alignment requirements for each memory or I/O range described by an implemented configuration-space base register. Using these requirements, PCI Services finds a suitable hole in the memory or I/O address space and updates the corresponding base register to point to this range. This is taken care of by the system (firmware and/or the kernel) at the time of the card’s initialization.

Map PCI to Port Handle
Mapping a PCI I/O space address to a port handle is the act which allows a driver to access the I/O space using pci_read_port_uintNN_isc() and pci_write_port_uintNN_isc(), passing in the port handle as a argument. The mapping is done through a call to pci_get_port_handle_isc().

Map PCI to Virtual Address
Mapping a PCI memory space address to a virtual address is the act that allows a driver to access PCI space using READ_REG_UINTNN_ISC() or WRITE_REG_UINTNN_ISC() with that virtual address. The mapping is done through a call to map_mem_to_host().

Memory Mapped I/O (MMIO)
I/O that occurs by mapping the devices’s I/O to system memory.

MP Multi-Processor

MP Safe
Describes a module which is protected in an MP environment through the use of various spinlocks and semaphores. Note that MP-safeness does not imply any performance considerations due to the granularity of the semaphores (e.g., use of a single I/O Empire semaphore or separate semaphores for each instance all imply MP-safeness).

MP Scalable
Describes an MP module which may add components without causing more drain on other MP modules. An MP-scalable driver will provide a separate spinlock for each instance of the driver. Non MP-scalable drivers may still be MP-safe but perhaps only provide a single semaphore and spinlock for all instances of the driver. Adding more instances of a non MP-scalable driver will therefore cause additional taxing of those resources for each instance added to the system.

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.

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.
On-Line Addition and Replacement (OLA/R) The ability to insert adapter cards and replace such cards while a system is being used (Hot Plug).

PA Precision Architecture. When referring to buses, these are buses which conform to the Precision I/O Architecture.

Packet DMA A type of DMA that maps a host memory buffer temporarily. This is used when pre-existing memory objects must be mapped for DMA, or when a mapping only needs to be temporary.

PCI Peripheral Component Interconnect. An industry standard bus used mainly by current generations of HP platforms as a means of providing expansion I/O.

PCI Address An address in the PCI memory or I/O space. This is the type of address found in a PCI memory or I/O base address register. It is NOT a virtual address or an I/O port handle, which a driver could use to access a card.

PCI Card A PCI bus can have up to 32 devices; each device can have up to eight functions. A PCI card can have single or multiple devices; each device can have single or multiple functions. For example, a four-port LAN card is a multi-device PCI card, but none of these devices is multi-functional. On the other hand, a dual-port SCSI card is a single device, but it has two functions.

PCI Configuration Space This always-accessible space allows a driver to configure and obtain status from PCI devices or functions.

PCI I/O Space The ‘space’ that is addressed by an I/O cycle on the PCI bus. This is a less often used way to access card registers on cards who choose to respond to PCI I/O accesses. Most cards have registers that are in PCI memory space instead of I/O space (i.e., they respond to PCI memory cycles, not PCI I/O cycles). PCI memory space The space that is addressed by a memory cycle on the PCI bus. It is called memory space to indicate that it is memory-mapped input/output, as opposed to a special I/O style of input/output. Typical cards map their registers into PCI memory space, meaning they can only be accessed by PCI memory cycles.

PCI Memory Space The “space” that is addressed by a memory cycle on the PCI bus. It is called memory space to indicate that it is memory-mapped input/output, as opposed to a special “I/O” style of input/output. The current PA Workstation I/O architecture allows the PA processor to directly access PCI memory space (i.e., a single instruction). Typical cards map their registers into PCI memory space, meaning they can only be accessed by PCI memory cycles.

Physical Address Real address by which host memory or an I/O device register is accessed.

Port Handle The kernel resource associated with a mapped range of PCI I/O space. This handle is used to access the I/O space addresses by calling pci_read_port_uintNN_isc() and pci_write_port_uintNN_isc().

Port I/O (PIO) Communication with an I/O device using the device’s ports.

PPA Physical Point of Attachment
Glossary

SAP

**SAP** Service Attach Point

SCSI Small Computer System Interface. An industry standard external I/O bus available on all HP9000 systems.

**SCSI** 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.

**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.

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.
Symbols
/stand/build, 196
/usr/contrib/bin/intctl command, 551
__LP64__, 541

Numerics
64-bit data module, 536
64-bit safety, 533

A
Aborting an I/O Request, SCSI, 375
adding a Series 700 driver to configuration files, 190
ANSI function prototypes, 544
arguments, driveradmin, ntwrk, 316
ARP configuration, ntwrk, 240
arpcom structure
  packet types, 579
arpcom structure, ntwrk, 216
audience, 18

B
Beta Semaphores, 72
  overview, 74
binding and unbinding, protocol, ntwrk, 234
Block Device Features, 37
block devices, 36
block I/O, 39
bound SAP data fields, ntwrk, 262
buf structure, 47
buf Structure, Device Queue Management, 162
buffering data using uiomove(), 136
build environment, 196
Bus Device Reset, SCSI, 374
Bus Structure, SCSI, 349

C
C_ALLCLOSES Flag, 131
Cache Coherence, 60
calling
  uiomove(), 136
Calling driver_attach(), SCSI, 360
Calling driver_install(), SCSI, 357
calling physio(), 136
cdevsw, 39
CDIO
  basic components, 28
Character Device Driver Features, 37
character I/O, 39
Coherency, Driver Requirements for, 61
Coherent I/O Systems, 65
command word
  ioctl, 139
command, /usr/contrib/bin/intctl, 551
commands and processing, driver control, ntwrk, 263
compatibility disclaimers, 19
compiling a driver, 187
Concepts DLKM, 504
conf.h header file, 545
configuration
  ARP, ntwrk, 240
  IP, ntwrk, 240
  lanscan output, ntwrk, 240
Configuration Routines, Writing, 102
Configuration Space Accesses, 58
Configuration Space Services, 167
Configured, 520
Context Dependent I/O Modules, 28
Control Block, SCSI, 330
control functions, network driver, 250
current driver settings, SAM, ntwrk, 313
Current Negotiated Information, SCSI, 365

D
data
  fields, bound SAP, ntwrk, 262
  fields, enet_dlip_data_t, ntwrk, 247
  fields, enet_ift, ntwrk, 259
  fields, enlan_ift, ntwrk, 261
  link interface layer, ntwrk, 209
Data Caching, 37
Data protection
  SCSI, 435
Data Structure Diagram, SCSI, 351
data structures
  kernel, 47
Data Structures, Interface Card Management, 162
Data Structures, SCSI, 330
dd_close() Routine, SCSI, 413
dd_ioctl() Routine, SCSI, 415
dd_ioctl_okay() Routine, SCSI, 418
dd_open() Routine, SCSI, 420
dd_pass_thru_done() Routine, SCSI, 423
dd_pass_thru_okay() Routine, SCSI, 423
dd_read() Routine, SCSI, 424
dd_start() Routine, SCSI, 426
dd_strategy() Routine, SCSI, 429
dd_write() Routine, SCSI, 432
Index

Deadlocks
  overview, 75
defines required, SAM, ntwrk, 313
Defining Installation Structures, 85
Description DLKM, 504
Description Service, 171
device
  /DLPI drivers synchronization, ntwrk, 244
drivers, STREAMS, ntwrk, 215
  file, ntwrk, 209, 239
Device Driver Interface Management, 162
Device Driver Interrupt Driven Routines, 161
Device Driver, Sample Header, 91
device drivers, 36
  and major numbers, 39
  kernel association, 39
Device Queue Management, 161
Device Special File, create, 202
device special files
  major and minor numbers, 39
Device switch table
  SCSI, 403
Device Switch Tables, 41
devices
  block, 36
  character, 36
  raw, 36
disclaimers
  support/compatibility, 19
DLKM
  Advantages, 509
  Concepts, 504
  Description, 504
  Driver Loading Concepts, 510
  Driver Unloading Concepts, 510
  DriverConfiguration Concepts, 512
  Kernel Config. Tool Set Commands and Options, 517
  kernel configuration tool use, 515
  Kernel Configuration Tools Description, 516
  master File Definition, 512
  Miscellaneous Modules, 515
  Modstub.o File Definition, 514
  Module Packaging, 508
  Module Types, 509
  Procedures for Dynamically Configured Loadable Modules, 520
  Procedures for Statically Configured Loadable Modules, 530
  space.h File Definition, 514
  STREAMS Drivers, 514
  system File Definition, 513
  Terms and Concepts, 505
  Tools, 515
  Tools for Building Static or Dynamic Kernels, 516
  Tools that interface to DLKM, 517
DLKM Overview, 503
DLPI
  primitives and IOCTLs, ntwrk, 256
  protocol, primitives, ntwrk, 246
  protocol, sequence, ntwrk, 239
DLPI protocol
  kinds, 580
  primitives, 573
DMA Mapping, 63
DMA Services, 169
dma.h header file, 546
dma_sync_IO, rules, 62
driver
  Configuration File, SAM, ntwrk, 306
  control commands and processing, ntwrk, 263
  INIT Script, SAM, ntwrk, 307
Driver Configuration Overview, 102
driver environment, 25
Driver Event Handling Services, 173
Driver Event Timeout Values, OLAR, 487
driver fragment skeleton, 54
Driver Handler, Registering, 176
Driver install routine
  SCSI, 402
Driver Installation and Initialization, SCSI, 357
Driver Loading Concepts DLKM, 510
Driver Master File, Creating, 190
Driver Master File, editing, 191
Driver Master File, example, 192
Driver Modifications for OLAR Support, 484
Driver Modifications Needed, OLAR, 480
Driver Name, Choosing, 82
Driver Probe Routines, Writing, 112
Driver Registering its Event Handler, OLAR, 484
Driver Registering its Event Mask, OLAR, 485
Driver set up structures
  SCSI, 399
Driver Unloading Concepts DLKM, 510
driver_attach()
  ntwrk, 228
Index

driver_attach() routine, 108
driver_attach(), diagnostics, 108
driver_close()
declaration, 129
driver_close() routine, 129
driver_close(), Exclusive Open Device, 130
driver_close(), Multiple Open Device, 130
driver_close(), Shared Open Device, 130
Driver_dev_init() routine
   SCSI, 403
driver_dev_init() routine, 110
driver_if_init() routine, 110
driver_install()
   ntwrk, 226
driver_install() routine, 104
driver_install(), Device and WSIO Pseudo
   Drivers, 105
driver_install(), Interface Drivers, 106
driver_install(), Monolithic Drivers, 107
driver_install(), non-WSIO Pseudo Drivers, 106
driver_ioctl() routine, 137
driver_isr() routine, 157
driver_minor_build() routine, 124
driver_minphys() routine, 146
driver_open() routine, 125
driver_psize() routine, 160
driver_read() routine, 132
driver_select() routine, 147
driver_strategy(), 153, 154
driver_strategy() routine, 151
driver_write() routine, 132
driveradmin
   arguments, ntwrk, 316
   command, ntwrk, 314
   shared library, ntwrk, 316
driveradmin shared library example, 623
DriverConfiguration Concepts DLKM, 512
driverlinkloop command, ntwrk, 320
Drivers
   pre-11.0, 533
drivers
   adding Series 700 driver to configuration
      files, 190
device, 36
interface, 36
interrupt context, 42
lower half, 42
pseudo, 36
two levels, 42
upper half, 42
drivers, LBI, 552
drivers, TBI, 552
drv_info_t Structure, 88
drv_ops_t Structure, 85
dynamically assigned major numbers, 39
E
   encapsulation method, 572
   encapsulation methods support, ntwrk, 320
   Endian Services, 168
   enet_802_2_ics(), ntwrk, 269
   enet_dlpi_data_t
      data fields, ntwrk, 247
      structure, ntwrk, 246
   enet_dlpi_unitdata_out(), ntwrk, 267
   enet_dlpi_wakeup(), ntwrk, 244
   enet_ethernet_lcs(), 270
   enet_hw_req(), ntwrk, 267
   enet_if_data_fields, ntwrk, 259
   enet_if_t structure, ntwrk, 258
   enet_isr()
      Interrupt Service Routine, 271
   enet_process_looper(), ntwrk, 269
   enet_process_packet(), ntwrk, 269
   enet_receive_pkts(), ntwrk, 269
   enet_slow_hw_req(), ntwrk, 267
   enet_transmit_complt(), ntwrk, 267
   ENET_TRANSMIT_FRAME, ntwrk, 267
   enet_transmit_pended_frames(), ntwrk, 268
   enetlinkloop, 703
   enlan_if()
      data fields, ntwrk, 261
      structure, ntwrk, 261
   Entry point routines
      SCSI, 408
   Entry Point Routines, Writing, 125
   environment layer, STREAMS, ntwrk, 210
   EPIC bridge, ntwrk, 273
   Error handling
      SCSI, 411
   Ethernet protocol, TYPE values, 582
   event flags, LBI, 553
   event flags, TBI, 557
   Event Handling Function, OLAR, 488
   Event Timeout Values, Setting, 179
   example
      I/O read request, 45
   Examples
      Shared Library for driveradmin, 623
      Shared Library for lanscan, 623
      exclusive open, 126
   External Interfaces to an Interface Driver,
      SCSI, 325
Index

extracting the system file, 196

F
fast path data transmission, ntwrk, 255
figures
  Driver Configurations, 30
  WSIO CDIO as a buffer zone, 30
File System Support, 37
Files, Device Special, 40
filter_packet(), 598
flow, interrupt migration, 563
format, packet protocol, ntwrk, 235
format_link_nice(), 598
format_link_raw(), 601
formatting messages, ntwrk, 299
functions, message service, ntwrk, 251

G
General I/O system (GIO), 27
Generic Function Switch, 48
get/set request flow, SAM driver, ntwrk, 309

H
handling interrupts, 52
Handling Timeouts, SCSI, 376
Hardware Considerations, 64-bit safety, 542
header file
  conf.h, 545
dma.h, 546
ui o.h, 546
header files
  device classes, 397
general, 397
SCSI, 397
Header Files, System Defined, 83
HP-UX Logging and Tracing, ntwrk, 284
hw_dlpi structure, ntwrk, 216, 221
hw_ifl structure
  encapsulation method, 572
  MAC type, 572
  ntwrk, 216
  protection, ntwrk, 222
  protocol type, 572

I
I/O
  flow of a request, 43
  kernel data structures, 47
  system calls, 42
I/O Adapters, 64
I/O Path, SCSI, 353
I/O Port Access Services, 58
I/O Port Space Services, 167
I/O subsystem structure, 27
I/O Switch Tables, 48
I/O Synchronization Service, 172
I/O within SCSI Device Driver, 354
I/O Within SCSI Interface Layer, 356
I/O Within SCSI Service Layer, 355
ID number, subsystem, N, ntwrk, 285
Identifying Routines for Driver, 97
IEEE 802.2 LLC use, SAP values, 581
IHV Network Drivers, ntwrk, 304
ILP32 data models, 536
inbound path, ntwrk, 268
int32_t, 540
int64_t, 541
intended audience, 18
interface
  layer, data link, ntwrk, 209
  name, ntwrk, 239
  statistics, displaying, ntwrk, 318
  statistics, request, ntwrk, 317
Interface Driver Development, 357
Interface Driver Development, SCSI, 357
Interface Driver Service Entry Points, SCSI, 325
Interface Driver Switch, SCSI, 340
Interface Driver, Sample Header, 93
interface drivers, 36
Interface Function Switch, 48
interface select code table, 47
Interrupt Control Stack (ICS), 71
interrupt line sharing, 552
Interrupt Migration, 552
interrupt migration flow, 563
Interrupt Processing, SCSI, 374
Interrupt Service Routine
  enet_isr(), 271
interrupt service routine, 157
Interrupt Service Routine (ISR), 73
Interrupt Services, 170
interrupts
  handling, 52
  interrupts, transaction based, 552
  interrupts, line based, 552
intptr_t, 540
ioctl command word, 139
IOCTL processing, ntwrk, 253
iovec structure, 49
IP configuration, ntwrk, 240
ISC, SCSI, 336
isc_table_type Structure, 163
K

kernel and drivers association, 39
Kernel Config. Tool Set Commands and Options DLKM, 517
kernel configuration tool use DLKM, 515
Kernel Configuration Tools Description DLKM, 516
Kernel Containing Driver, build, 197
Kernel data structures, 47
kget_log_instance(), ntwrk, 295
KLOG_CK(), ntwrk, 294
klogg_write(), ntwrk, 295
ktrc_write(), ntwrk, 291

L

lan

description, ntwrk, 318
encapsulation methods, displaying, ntwrk, 318
output configurations, ntwrk, 240
lan scan shared library example, 623
LBI drivers, 552
LBI event flags, 553
line based interrupts, 552
line sharing, interrupt, 552
Lock acquisition rules, 75
Lock orders, 75
SCSI, 435
logged_info, ntwrk, 262
logged_link, ntwrk, 262
logging, network, 280
logging/tracing
assigning subsystem ID, ntwrk, 285
classifying log data, ntwrk, 287
classifying trace data, ntwrk, 286
configuring developed subsystems, 616
formatting trace data, ntwrk, 287
HP-UX, ntwrk, 284
internationalization, 615
makefiles, ntwrk, 290
message catalog, 615
ntwrk, 280
source files, ntwrk, 290
subsystem installation testing, 621
what and when, ntwrk, 288
Logical Unit Structure, SCSI, 346
LP64 considerations, 140, 537
LP64 data models, 536

M

MAC types, 572
major and minor numbers
in device special files, 39
major numbers
and device drivers, 39
defined, 39
dynamic assignment and ranges, 39
Mapping I/O Space, 56
Mass Storage Stack, 323
master
agent, ntwrk, 277
master File Definition DLKM, 512
mechanism
software trigger, 53
Mechanisms synchronization, 72
Memory Allocation, 56
Memory Allocation Services, 56, 174
Memory Mapped Services Registers, 57
message
block reception, ntwrk, 255
block transmission, ntwrk, 254
catalog, shared library, ntwrk, 317, 319
service functions, ntwrk, 251
MIB II
extended structure, 570
standard structure, 569
MIB, event message, 574
Migration, Interrupt, 552
minor numbers
defined, 40
Miscellaneous Modules DLKM, 515
Modified.o File Definition DLKM, 514
Module Packaging DLKM, 508
Module Types DLKM, 509
MTU values, 575
multiple open, 126
Multiprocessing, 69
Idle Time, 72
Timing hazards, 72
Multiprocessing to Uniprocessing Comparison, 71
Multiprocessor implications
SCSI, 407

N

N, subsystem ID number, ntwrk, 285
Native Language Support (NLS), 615
netfmt(1M), ntwrk, 284, 299
nettl(1M), ntwrk, 283
nettlconf(1M), ntwrk, 283
Index

nettlgen.conf(4), ntwrk, 283
Network
  Alternative Means of Development
    Debugging, 302
  Assign Subsystem ID, 285
  Calling driver_attach(), 228
  Calling driver_install(), 226
  Classify Log Data, 287
  Classify Trace Data, 286
  Configuring a Networking Driver Through SAM, 303
  Control Functions, 250, 263
  Data Link Interface Layer, 209
  Data Structures, 258
  Data Structures and Interfaces, 216
  Defines Required at the Driver, 313
  Device Driver/DLPI Driver Synchronization, 244
  Device File, Interface Name, and PPA Number, 239
  Distinguishing Packet Protocol Format, 235
  DLPI Interface to Upper Layers, 239
  Driver Configuration File, 306
  Driver Development Steps, 213
  Driver INIT Script, 307
  Driver Structure Protection, 222
  Driver Support for DLPI, 246
  Driving the NIC, 257
  Entering STREAMS from ICS, 246
  Format Trace Data, 287
  Formatting Networking Trace/Log Messages, 299
  Generic Network Configuration Supported by SAM, 306
  HP-UX Networking Interface Architecture, 207
  hw_dlpi Structure Description and Initialization, 221
  hw_ift Structure Protection, 222
  hw_ift_t Structure Description and Initialization, 216
  Inbound Path, 268
  Initializing Networking Device Drivers, 224
  IOCTL Processing, 253
  IP and ARP Configuration, 240
  Logging Code Sample, 296
  Major Data Structures, 246
  Management Support, 276
Message Block and Queue Functions and Macros, 238
Network Monitoring Commands, 314
Network Protocol Layer, 209
Opening and Closing a Driver, 249
Outbound Path, 266
Overview of Networking Driver Structure, 211
Passing Data to HP-UX Tracing/Logging, 290
Platform Specifics, 273
Protection and Synchronization for Driver and Transport, 222
Protocol Binding and Demultiplexing, 233
Protocol Interface Layer, Device File, and STREAMS Head, 209
Reception of Message Blocks, 255
SAM Interface to IHV Network Drivers, 304
SAM/Driver get/set Request Flow, 309
STREAMS DLPI and Network Driver, 242
STREAMS Environment, 210
STREAMS Model Device Drivers, 215
STREAMS Synchronization, 245
Summary of DLPI Primitives and IOCTSs, 256
Tracing and Logging, 280
Transmission of Message Blocks, 254
V Class, 273
network debugging, alternative means, ntwrk, 302
network driver
closing, 249
closure functions, 250
driver_attach(), 228
driver_install(), 226
initializing, 244
opening, 249
sequence, STREAMS DLPI, 243
network interface, architecture, 207
NIC, ntwrk, 257
Noncoherent I/O Systems, 65
ntwrk, 249
O
OLAR
  Driver Event Timeout Values, 487
  Driver Modifications for OLAR Support, 484
  Driver Modifications Needed, 480
Driver Registering its Event Handler, 484
Driver Registering its Event Mask, 485
Event Handling Function, 488
Handling, 490
Introduction, 479
Miscellaneous Changes Required at the Driver, 499
On Line Addition Process, 490
On Line Replacement Process, 492
Overview, 478
Performing OLAR of PCI I/O Cards, 500
Resume, 497
Supported Hardware and Software, 479
Suspend, 494
WSIO Event Handling Structures, 481
WSIO Interfaces, 481
wsio_drv_info_t, 481
On Line Addition Process, OLAR, 490
On Line Replacement Process, OLAR, 492
Ordered Interrupts Service, 172
outbound path, ntwrk, 266
output messages, localizing, ntwrk, 317
Overview DLKM, 503

P
packet
headers, 576
protocol, format, ntwrk, 235
type, network, 233
types, 579
PCI
Automatic IRQ determination, 453
Blocking transactions, 445
Bus-dependent implementation, 462
Bus-independent functionality, 462
byte swapping, 449
Configuration space restrictions, 455
Configuration Spaces, 443
Constants and data structures, 466
Device operation, 457
Device setup, 451
DMA, 458
driver entry point routines, 475
Endian issues, 449
I/O base registers, 453
I/O space, 444
Leveraging existing drivers, 461
Mapping base address registers into I/O space, 451
Mapping the memory base register, 454
Mastered reads and writes to host memory, 446
Masters and Coherence, 460
Memory base registers, 452
Multiprocessor safety, 465
Nonblocking transactions, 445
PCI_LITTLE_ENDIAN_ONLY flag, 457
pre-swapping, 449
Processor and card mastered reads and writes to host memory, 447
Processor mastered reads and writes, 445
Register Spaces, 443
sample driver_attach(), 472
sample driver_install(), 471
Transaction ordering, 445
Unsupported WSIO functions, 464
Using base address registers, 451
Write side-effects, 446
PCI OLAR Support, 176
PPA number, ntwrk, 239
primitives and IOCTLs, DLPI, ntwrk, 256
Procedures for Dynamically Configured Loadable Modules DLKM, 520
Procedures for Statically Configured Loadable Modules DLKM, 530
processing
packets, ntwrk, 234
Processing Completed I/O Requests, SCSI, 371
Promiscuous, Inbound and Outbound, 236
Protection and Synchronization, SCSI, 368
protocol
binding and unbinding, ntwrk, 234
demultiplexing, ntwrk, 233, 234
interface layer, ntwrk, 209
kind, network, 233
kinds, 580
layer, ntwrk, 209
types, 572
value, ntwrk, 233
Pseudo Driver, Sample Header, 94
pseudo drivers, 36
ptr32_t, 541

Q
qlisp driver, SCSI, 385
qlisp_build_cmd 64, SCSI, 388
Index

qlisp_build_cmd, SCSI, 388
qlisp_call_cbfns, SCSI, 391
qlisp_fast_complete, SCSI, 390
qlisp_get_req_iocb, SCSI, 388
qlisp_if_abort, SCSI, 393
qlisp_if_bdr, SCSI, 393
qlisp_if_lun_open, SCSI, 392
qlisp_if_reset_bus, SCSI, 393
qlisp_if_start, SCSI, 389
qlisp_init interrupts, SCSI, 387
qlisp_init_alloc_queues, SCSI, 386
qlisp_init_attributes, SCSI, 386
qlisp_init_common, SCSI, 387
qlisp_init_firmware, SCSI, 387
qlisp_init_ifsw, SCSI, 387
qlisp_init_master, SCSI, 387
qlisp_init_pci_cfg, SCSI, 386
qlisp_init_reset_card, SCSI, 386
qlisp_init_slave, SCSI, 388
qlisp_init_start queues, SCSI, 387
qlisp_iocb_cnt, SCSI, 388
qlisp_isr, SCSI, 392
qlisp_mailbox_cmd, SCSI, 388
qlisp_multi driver, SCSI, 380
qlisp_multi_dev_probe, SCSI, 382
qlisp_pci_attach, SCSI, 388
qlisp_post_reset_delay, SCSI, 391
qlisp_process_rsp, SCSI, 391
qlisp_reset_bus, SCSI, 393
qlisp_send_marker, SCSI, 394

writing a driver_isr(), 157
writing a driver_minor_build(), 124
writing a driver_minphys(), 146
writing a driver_open(), 125
writing a driver_psize(), 160
writing a driver_read() routine, 132
writing a driver_select(), 147
writing a driver_strategy(), 151
writing a driver_write() routine, 132

Routines and services
SCSI, 412
Rules for lock acquisition, 75

S

SAM
Driver Configuration File, ntwrk, 306
Driver get/set request flow, ntwrk, 309
Driver INIT Script, ntwrk, 307
SAM, System Administration Manager, ntwrk, 303
Sample Driver, SCSI, 378
SAP values, 581
SCSI, 357

Aborting an I/O Request, 375
Bus Device Reset, 374
Bus Structure, 349
Calling driver_attach(), 360
Calling driver_install(), 357
Current Negotiated Information, 365
Data protection, 435
Data Structure Diagram, 351
Data Structures, 330
dd_close() Routine, 413
dd_ioctl() Routine, 415
dd_ioctl_okay() Routine, 418
dd_open() Routine, 420
dd_pass_thru_done() Routine, 423
dd_pass_thru_okay() Routine, 423
dd_read() Routine, 424
dd_start() Routine, 426
dd_strategy() Routine, 429
dd_write() Routine, 432
Device switch table, 403
Driver install routine, 402
Driver Installation and Initialization, 357
Driver_dev_init() routine, 403
Entry point routines, 408
Error point routines, 411

R

raw devices, 36
Raw I/O, Buffered or Unbuffered, 38
references to other material, 22
Register Mapping and DMA, SCSI, 366
Register Services, 166
regular data path transmission, ntwrk, 254
Releasing Pending Timeouts, 271
Reset Bus, SCSI, 375
Resume, OLAR, 497
routines
calling physio(), 136
driver_strategy() read, 154
driver_strategy() write, 153
software trigger, 54
writing a driver_attach(), 108
writing a driver_close(), 129
writing a driver_dev_init(), 110
writing a driver_if_init(), 110
writing a driver_install(), 104
writing a driver_ioctl(), 137
External Interfaces to an Interface Driver, 325
Handling Timeouts, 376
I/O Path, 353
I/O within SCSI Device Driver, 354
I/O Within SCSI Interface Layer, 356
I/O Within SCSI Service Layer, 355
Include header files, 397
Interface Driver Service Entry Points, 325
Interruptive Driver Switch, 340
Interrupt Processing, 374
ISC, 336
Lock and semaphore orders, 435
Logical Unit Structure, 346
Multiprocessor implications, 407
Processing Completed I/O Requests, 371
Protection and Synchronization, 368
qlisp driver, 385
qlisp_build_cmd, 388
qlisp_build_cmd 64, 388
qlisp_call_cbfns, 391
qlisp_fast_complete, 390
qlisp_get_req_ioeob, 388
qlisp_if_abort, 393
qlisp_if_bdr, 393
qlisp_if_lun_open, 392
qlisp_if_reset_bus, 393
qlisp_if_start, 389
qlisp_init_interruptions, 387
qlisp_init_alloc_queues, 386
qlisp_init_attributes, 386
qlisp_init_common, 387
qlisp_init_firmware, 387
qlisp_init_ifsw, 387
qlisp_init_master, 387
qlisp_init_pci_cfg, 386
qlisp_init_reset_card, 386
qlisp_init_slave, 388
qlisp_init_start_queues, 387
qlisp_iocb_cnt, 388
qlisp_isr, 392
qlisp_mailbox_cmd, 388
qlisp_multi driver, 380
qlisp_multi_dev_probe, 382
qlisp_pci_attach, 388
qlisp_post_reset_delay, 391
qlisp_process_rsp, 391
qlisp_reset_bus, 393
qlisp_send_marker, 394
Register Mapping and DMA, 366
Reset Bus, 375
Routines and services, 412
Sample Driver, 378
SDTR/WDTR Negotiation, 365
Select_q, 350
Sending SCSI I/O Requests to the Target, 370
Service Interface Driver Service Calls, 328
Services, 437
Sleep and Wakeup Mechanism, 377
Special Routines, 374
struct buf, 343
Target Structure, 347
Underlying routines, 412
What to Negotiate, 366
When to Negotiate, 365
SCSI Control Block, 330
SCSI lock
Bus, 435
LUN, 435
Subsystem, 435
Target, 435
SCSI Parameters, Setting, 363
SCSI Services Interface, 325
SCSI Services Interface Functions, 325
SCSI Specific Part, 358
SCSI Subsystem Overview, 323
SCSI_GET_INITIATOR_PARAMS, 364
scsi_probe(), 121
SDTR/WDTR Negotiation, SCSI, 365
select code
interface, 47
Select_q, SCSI, 350
Semaphore orders
SCSI, 435
Sending SCSI I/O Requests to the Target, 370
Service Interface Driver Service Calls, SCSI, 328
Services
SCSI, 437
Services Interface Functions, SCSI, 325
Services Interface, SCSI, 325
set_up_8022(), 596
set_up_ether(), 598
set_up_ip(), 597
set_up_link(), 596
Setting SCSI Parameters, 363
shared library
driveradmin, ntwrk, 316
message catalog, ntwrk, 317, 319
Index

to display statistics, ntwrk, 315
Shared Library Examples, 623
shared open, 126
skeletons
driver fragment, 54
Sleep and Wakeup Mechanism, SCSI, 377
sleep(), 76
SNAP, protocol TYPE values, 582
Software Considerations, 64-bit safety, 543
software trigger mechanism, 53
software trigger routine, 54
space.h File Definition DLKM, 514
Special Routines, SCSI, 374
Specific Part, SCSI, 358
Spinlock routines
overview, 73
Spinlocks, 72
overview, 73
spinlocks
deadlock avoidance, ntwrk, 223
protection model, ntwrk, 222
SSCSI
Driver set up structures, 399
statistic definitions, ntwrk, 317
statistics display via shared library, ntwrk, 315
STREAMS
device driver, ntwrk, 215
DLPI Network Driver Sequence, 243
environment layer, ntwrk, 210
head, ntwrk, 209
synchronization, ntwrk, 246
STREAMS Drivers DLKM, 514
struct buf, SCSI, 343
structure
uio, 49
structures
buf, 47
enet_dlpi_data_t, ntwrk, 246
enet_ift_t, ntwrk, 258
enlan_ift, ntwrk, 261
hw_dlpi, ntwrk, 216, 221
hw_ift, ntwrk, 216
iovect, 49
Subagent, ntwrk, 277
subformat
calls, 604
functions, 588
option, 614
subsys_N_format(), 592
subsys_N_get_options(), 592
subsystem
ID number, N, ntwrk, 285
support disclaimers, 19
Supported Hardware and Software, OLAR, 479
Suspend, OLAR, 494
Synchronization
sleep(), 76
wakeup(), 76
synchronization
device/DLPI drivers, ntwrk, 244
STREAMS, ntwrk, 246
Synchronization Mechanisms, 72
system
Administration Manager, SAM, ntwrk, 303
System Attribute Services, 173
system calls, 42
system File Definition DLKM, 513
system file, extracting the, 196
System Header Files, Choosing, 83
T
Target Structure, SCSI, 347
TBI drivers, 552
TBI event flags, 557
Terms and Concepts DLKM, 505
timeout mechanisms, 50
timeout() routine, 51
Timeouts, Explanation, 50
tlBanner_char(), 613
tl_check_cat_version(), 613
tl_format_fprintf(), 609
tl_format_write(), 607
tl_get_line(), 614
tl_get_parms(), 611
tl_header_format1(), 605
tl_log_class(), 614
tl_raw_format(), 610
tl_trace_kind(), 613
Tools DLKM, 515
Tools for Building Static or Dynamic Kernels
DLKM, 516
Tools that interface to DLKM, 517
tracing, network, 280
tracing/logging
assigning subsystem ID, ntwrk, 285
classifying log data, ntwrk, 287
classifying trace data, ntwrk, 286
configuring developed subsystems, 616
formatting trace data, ntwrk, 287
internationalization, 615
Index

makefiles, ntwrk, 290
message catalog, 615
ntwrk, 280
source files, ntwrk, 290
subformatter code, 585
subsystem installation testing, 621
transaction based interrupts, 552
transmission
  fast data path, ntwrk, 255
  message block, ntwrk, 254
  regular data path, ntwrk, 254
troubleshooting, HP approach, ntwrk, 280
TYPE values
  Ethernet, 582
  SNAP, 582

U
uio structure, 49
uio.h header file, 546
uiomove(), 136
Underlying routines
  SCSI, 412
uint32_t, 540
uint64_t, 541
uintptr_t, 540
Uniprocessing to Multiprocessing
  comparison, 71
untimeout() routine, 51
using physio(), 135

V
V class PCI I/O architecture, ntwrk, 273

W
wakeup(), 76
Writing a Driver, Overview, 81
WSIO Event Handling Structures, OLAR, 481
WSIO Interfaces, OLAR, 481
WSIO Mapping Services, 65
WSIO Services, new for 11.11, 164
wsio_completion_cb(), 561
wsio_drv_data_t Structure, 90
wsio_drv_info_t Structure, 91
wsio_drv_info_t, OLAR, 481
WSIO_EVENT_LBI_INTR_MIGR, 553
WSIO_EVENT_OFFLINE_CPU, 557
WSIO_EVENT_ONLINE_CPU, 562
wsio_generic_event_t, 554
wsio_intr_activate(), 565
wsio_intr_deactivate(), 561, 565
wsio_intr_enable(), 561
wsio_intr_free(), 565
wsio_intr_get_assigned_cpu(), 561, 565
wsio_intr_get_irq_line(), 565
wsio_intr_get_txn_info(), 561, 565
wsio_intr_migr_info_t, 554
wsio_intr_set_cpu_spec(), 561, 565
wsio_reg_drv_capability_mask(), 553
wsio_register_addr_probe(), 114
wsio_register_dev_probe(), 112