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

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

The first section contains an overview of the structure of networking drivers. You may use the steps outlined in this section as a general guide to HP-UX driver design. The second section introduces the HP-UX networking interface architecture for the PCI platform. Be sure to review this section before beginning development of your networking driver. The remaining sections of this chapter contain network device driver topics and sample code for each part. Refer to this information, as well as the enet sample driver provided in your driver development kit, to create your PCI networking device driver.

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

Basic STREAMS module/driver development and general networking concepts and RFCs are not included in this documentation.
Overview of Networking Driver Structure

The flowchart in Figure 7-1, “Steps to Develop a Networking Driver,” shows a sequence of high level steps that you can use as a development guide when writing networking drivers on HP-UX systems. The sequence of information in the flowchart also closely maps to the sequence in which the information is presented in this chapter. Refer to the specific module descriptions below for pointers to the subject areas shown in the flowchart.

Prior to beginning the development of your driver, HP recommends that you carefully study the material on HP-UX Networking Interface Architecture in the following section.

STREAMS model device drivers

IHVs and ISVs are expected to write their own DLPI layer along with the network interface device driver in HP-UX 11.0. The DLPI layer is implemented as a STREAMS driver and the PCI network interface driver as a WSIO driver. The interface between the DLPI layer and the network interface device driver is not defined by HP. Third parties are free to define it according to their design needs. This document provides a framework which includes a STREAMS DLPI driver, `dlpi_enet`, a PCI network interface driver, `enet`, and the interface between the two, as part of the development kit.

NOTE

The names DLPI STREAMS driver, native DLPI, DLPI driver and DLPI layer are used interchangeably. Also, network interface driver, WSIO network driver, and PCI network interface driver mean the same.

Development Steps

a. HP-UX Network Interface Architecture  Overview of the STREAMS environment in HP-UX.

b. Data Structures  Describes the data structures in the networking interface layer: `hw_ift_t`, `hw_dlpi_t` and device driver data structure framework. Refer to “Data Structures
Creating Networking Device Drivers

Overview of Networking Driver Structure

and Interfaces” on page 184 for detailed information about these data structures.

c. Protection and Synchronization Describes the OSF/Encore spinlock protection model. Refer to “Protection and Synchronization for Driver and Transport” on page 189 for more detailed information about supporting the spinlock scheme in HP-UX.
Creating Networking Device Drivers

Overview of Networking Driver Structure

Figure 7-1  Steps to Develop a Networking Driver

Figure 7-1: Steps to Develop a Networking Driver

1. **Begin Driver**
   - HP-UX

2. **Data**
   - Describes the install and initialization routines for WSIO networking drivers and STREAMS drivers.

3. **Protection and Synchronization**

4. **Driver Initialization**
   - WSIO Driver
   - STREAMS Driver

5. **Protocol configuration, binding and Demultiplexing**
   - Network Management
   - Logging & Tracing
   - Driver Initialization
   - WSIO Driver

6. **mblk and queue macros**

7. **DLPI Sequence**

8. **STREAMS Driver**

9. **WSIO Driver**

10. **DLKM?**
     - Yes: DLKM Wrappers
     - No: Network Management

11. **Auxiliary**
     - Yes: SAM Support
     - No: SAM

12. **Network Management**

13. **Logging & Tracing**

14. **Driver Initialization**

15. **Driver**

**d. Driver Initialization** Describes the install and initialization routines for WSIO networking drivers and STREAMS drivers.
Creating Networking Device Drivers

Overview of Networking Driver Structure

The attach routine is discussed for the WSIO driver. Refer to “Initializing Networking Device Drivers” on page 191 for detailed information about these routines.

e. Protocol Configuration, Binding, and Demultiplexing  Describes configuration of the INET stack for the STREAMS model drivers. Also, the routines for the driver to bind and demultiplex upper-layer protocols to a device are explained. Refer to “Protocol Binding and Demultiplexing” on page 201 for detailed information about these routines.

f. mblk and queue macros  Lists macros commonly used by networking and STREAMS drivers. Refer to “mblk and queue macros” for detailed information about these routines.

g. DLPI sequence  Describes the general operation of DLPI STREAMS and network device drivers.

h. STREAMS Driver  Provides the overview of STREAMS DLPI driver and describes the major functions in the driver.

i. WSIO networking driver  Provides the overview of a networking device driver and describes the major functions in the driver.

j. DLKM Support?  If not, go to l.

k. DLKM Wrappers  Describes how to convert the STREAMS and network driver to DLKM modules.

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

m. Network Management Support?  If not, go to o.

n. Network Management Code  Describes the routines to support network management requests. Refer to “Network Management Support” on page 240 for more detailed information.

o. Logging and Tracing Support?  If not, go to q.

p. Logging and Tracing Code  Describes the routines to support Logging and Tracing. Refer to “Formatting Networking Trace/Log Messages” on page 271 for more detailed information.
Creating Networking Device Drivers

Overview of Networking Driver Structure

q. Auxiliary Files? HP customers expect to have automated
configuration through SAM, and to be able to display
link and encapsulation statistics and tracing and
logging messages. If not, go to Driver Complete.

r. SAM Support? If not, go to s.

s. SAM Files Provides configuration files to support the HP
menu-driven utility to configure a networking driver.
Refer to “Configuring a Networking Driver Through
SAM” on page 245 for more detailed information.

t. driveradmin Support? If not, go to u.

u. driveradmin Files Describes the shared library for the driveradmin
command. This is used to display link statistics and
perform various administrative tasks. Refer to
“Network Monitoring Commands: driveradmin,
driverlinkloop and lanscan” on page 246 for more
detailed information.

v. lanscan Support? If not, go to w.

w. lanscan Files Describes the shared library for the lanscan command.
This is used to display link encapsulation information.
Refer to “Network Monitoring Commands:
driveradmin, driverlinkloop and lanscan” on page 246
for more detailed information.

x. Logging and Tracing? If not, the driver is completed.

y. Logging and Tracing Files Describes the files needed to support
logging and tracing requests. Refer to Appendix C,
“How to Design a Networking Trace/Log Subformatter
and a Sample Subformatter,” for more detailed
information.

HP-UX Networking Interface Architecture

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

The HP-UX networking subsystem comprises three logical layers, as
shown in Figure 7-2, “Three-layered HP-UX Interface to the PCI Bus”,
and is briefly described in the following three subsections:
Creating Networking Device Drivers

Overview of Networking Driver Structure

“Data Link Interface Layer” on page 182
“Network Protocol Layer” on page 182
“Protocol Interface Layer, Device File, and STREAMS Head” on page 182
Figure 7-2  Three-layered HP-UX Interface to the PCI Bus

Application Layer

TCP/IP Networking Commands & Applications

Driver Networking Commands & Utilities

Protocol Interface Layer

Streams Head & Device File

Network Protocol Layer

TCP

UDP

STREAMS-based Protocol

Data Link Provider

Network Device Drivers

PCI bus

Media

Ethernet Card

Token Ring Card

X.25 LAPB

FDDI Card
Creating Networking Device Drivers

Overview of Networking Driver Structure

Data Link Interface Layer

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

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

Network Protocol Layer

The network protocol layer, above the data-link interface layer, encompasses four protocol families:

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

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

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

Protocol Interface Layer, Device File, and STREAMS Head

This interface layer directly supports applications. The main functions of this layer are to:

- Identify different applications on the same host (for example, a socket interface or a device file interface).
• Provide services from transport layer protocols and above to applications.

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

• Sockets
• Streams
• Device files

Streams Environment Layer

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

Driver writers should refer to the following documents for information concerning STREAMS modules and device drivers. Notable attention should be paid to the DLPI references. This document will only briefly discuss the STREAMS mechanisms and will concentrate on specific HP variants.

The following documents are recommended sources:

Hewlett-Packard Company:

• HP 9000 Networking DLPI Programmer’s Guide, HP Part No. 98194-90059

Other References:

• Data Link Provider Interface Specifications, Unix International

Streams Model Network Driver

As mentioned earlier, from HP-UX 11.0 onwards, third party network drivers are expected to have their own DLPI layer. DLPI STREAMS driver interface with STREAMS modules/drivers (IP and ARP modules) and other transport stacks. The interface between a native (third party) DLPI STREAMS driver and the PCI network interface driver is not controlled by HP. Third parties are free to design this interface.
Creating Networking Device Drivers

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, “Networking Driver Control Block and Structures.”). The driver control block provides information used in driving and controlling the interface hardware.

**hw_ift_t Structure Description and Initialization**

The `hw_ift_t` structure provides a consistent method for the network system utilities, `lanscan` (see `lanscan(1M)`), `driveradmin` and `driverlinkloop`, to display detailed information about all network devices. The `hw_ift_t` structure is described below.

```c
typedef struct hw_ift {
    hw_dlpi_t hp_dlpi;
    uint32_t mac_type;
    uint32_t llc_flags;
    uint32_t maj_num;
    uint32_t nm_id;
    uint32_t instance_num;
    uint32_t instance_num;
    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**: It must be initialized to all zeros.
- **mac_type**: Device type. See “MAC Types and Protocol Types” on page 502, Appendix D, for network media types.
- **llc_flags**: Link Level Control (LLC) encapsulation method.
## Data Structures and Interfaces

Defined flag values and the encapsulation method are listed in “MAC Types and Protocol Types” on page 502, Appendix D.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mjr_num</td>
<td>Major number of the device file associated with the device. The major number could be statically assigned or, if it is set to -1, the major number is allocated dynamically during driver initialization.</td>
</tr>
<tr>
<td>nm_id</td>
<td>Network management ID. nm_id should be initialized via a call to the <code>get_nmid()</code> routine.</td>
</tr>
<tr>
<td>instance_num</td>
<td>Device instance number. This is the value returned by calling the <code>wsio_isc_to_instance()</code> routine.</td>
</tr>
<tr>
<td>mtu</td>
<td>Maximum transmission unit (number of bytes) for a particular type of link or encapsulation. See “MTU Values” on page 505, Appendix D, for a list of predefined MTU values.</td>
</tr>
<tr>
<td>name</td>
<td>Driver device name used for naming shared libraries for <code>lanscan</code> and <code>driveradmin</code>.</td>
</tr>
</tbody>
</table>

### NOTE

The driver names `lan` and `fddi` are reserved for HP devices.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hdw_path</td>
<td>Hardware path obtained by calling <code>io_node_to_hw_path()</code> followed by <code>io_hw_path_to_str()</code>.</td>
</tr>
<tr>
<td>hdw_state</td>
<td>Hardware state of the device. 0, if the device is OK. If the device is not available, hdw_state must be set to LAN_DEAD.</td>
</tr>
<tr>
<td>mac_addr_len</td>
<td>Number of bytes of mac_addr[] for MAC address.</td>
</tr>
<tr>
<td>mac_addr</td>
<td>MAC address of the device. For Ethernet/IEEE 802.3 and FDDI, the address is in canonical form. For IEEE 802.5, the address is in wire form.</td>
</tr>
<tr>
<td>features</td>
<td>Features supported by device. Two flags are supported: DRV_MP Set this flag and make sure the device driver is MP-scalable or MP-safe, that is, uses spinlock()/spinunlock() to avoid...</td>
</tr>
</tbody>
</table>
race conditions. See “Protection and Synchronization for Driver and Transport” on page 189 in this chapter for more information. When this flag is set, the driver cannot use any spl* calls.

DRV_MBLK This flag must be set since the third party network driver is purely based on STREAMS model.

DRV_IP_MULTICAST This flag must be set if a driver supports the IP multicast feature.

DRV_LANC_PROMISC_SUPPORT This flag must be set if a driver supports promiscuous listening.

DRV_NO_FAST_PATH This flag must be set if a driver does not support fast path as described in “Transmission of Message Blocks”.

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

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

ppa PPA number for the interface. The driver should initialize this field with hw_ift->instance_num.

watch_timer For Hewlett-Packard internal use only. This field must be set to 0 for non-Hewlett-Packard devices.

reserved2 For Hewlett-Packard internal use only. This field must be set to 0 for non-Hewlett-Packard devices.

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

next Pointer to next hw_ift structure in list. This field is set by calling the hw_ift_attach() routine during device driver initialization. See “Initializing Networking
Device Drivers on page 191 in this chapter for detailed information.

The following example shows the initialization of the `hw_ift` structure. Initialization is generally done in the driver init routine:

```c
struct enet_ift_t *enetift_ptr;
hw_ift_t *hw_ift_ptr;
char mac_addr[6];
struct isc_table_type *isc_ptr; /* pointer to an isc_table structure */

hw_path_t hw_path;

hw_ift_ptr = &(enetift_ptr->hwift);

hw_ift_ptr->mac_type = DEV_ETHER;
hw_ift_ptr->llc_flags = IEEE | SNAP;
hw_ift_ptr->mjr_num = enet_drv_info.drv_info->c_major;
hw_ift_ptr->nm_id = get_nmid();
hw_ift_ptr->instance_num = wsio_iscto_instance(isc_ptr, NULL);
hw_ift_ptr->mtu = ETHER_MTU;
hw_ift_ptr->name = "enet";

io_node_to_path(isc_ptr->card_node,NULL,&hw_path);
io_hw_path_to_str(hw_ift_ptr->hdw_path,NULL,&hw_path);
hw_ift_ptr->hw_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. This structure should be initialized to zero. Further discussion of structure fields is not provided.
Protection and Synchronization for Driver and Transport

The major synchronization issue with networking device drivers is to avoid data corruption and race conditions when shared structures are accessed by multiple threads in MP systems. Driver data structures also need protection against interrupts. HP-UX transport networking adopted the OSF/Encore spinlock protection model in order to gain parallelism and provide scalable network performance. The spinlock scheme provides finer granularity locks, protecting data structures at finer levels, as opposed to grabbing a global network lock. More information on spinlocks is available in “Multiprocessing”, Chapter 4 of this manual, and spinlock(KER2) in the HP-UX Device Driver Reference manual.

NOTE

Each spinlock causes a busy-wait. Device-driver developers should be aware of the impact on system performance caused by the frequency of acquiring a spinlock and the duration of holding a spinlock.

As discussed in previous sections, the data structure in the network interface layer that link device drivers to the protocol layer is the hw_ift. The drivers have their own data structures—driver control block, send and receive management structures. These data structures are protected by using spinlocks.

hw_ift Structure Protection

One spinlock, the hwift_lock field, in the hw_ift structure is defined to protect the access to the structure fields.

The macros to acquire or release the hwift_lock spinlock to protect hw_ift structure fields are defined below.

HW_IFT_LOCK (hw_ift_ptr) Acquire a spinlock on hwift_lock.

    hw_ift_ptr: Pointer to an hw_ift structure.

HW_IFT_UNLOCK (hw_ift_ptr) Release previously acquired hwift_lock spinlock.
Creating Networking Device Drivers
Protection and Synchronization for Driver and Transport

hw_ift_ptr: Pointer to a hw_ift structure.

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

Driver Structure Protection

DLPI and network drivers use spinlocks to protect their internal data structures. HP-UX pre-defines the order (major order) for spinlocks for LAN and STREAMS drivers to avoid deadlock conditions when non-direct code paths are executed due to interruptions, such as faults, traps and interrupts.

Drivers can increase concurrency by choosing to have finer granularity locks. Since the major lock order is pre-defined by HP-UX, drivers could use different minor order spinlocks to protect access to data structures. For example, a WSIO driver could use one lock for transmit path and one for receive path data structures. This would allow the driver to receive and transmit concurrently.

A list of the relative pre-defined lock orders for spinlocks used by HP-UX LAN products is shown below.

LAN_LANX_LOCK_ORDER Lock order for a spinlock used by HP-UX LAN device drivers, such as btlan3 and lan2, to protect local data structures. This lock order should be used by all third-party networking device drivers during initialization of a spinlock used to protect device-driver structures.

LAN_HWIFT_LOCK_ORDER Lock order for spinlock hwift_lock, defined in sio/lan_dlpikrn.h, and the lock order protecting the embedded MIB structure.

STREAMS_USR1_LOCK_ORDER Lock order for spinlock used by STREAMS drivers to protect their data structures.
Initializing Networking Device Drivers

In HP-UX 11.0, developing a network interface driver involves developing a STREAMS DLPI driver and a WSIO network interface driver. A DLPI driver is part of STREAMS/UX and is used by the file system for device open and close. For this reason the DLPI driver is both a STREAMS and CDIO driver. The network interface driver is a WSIO driver. The initialization of one driver does not depend on the other. Initialization for each driver is described below.

STREAMS DLPI Driver Initialization

For a detailed description of generic STREAMS driver development, refer to the STREAMS/UX for HP900 Reference Manual. This section briefly explains the initialization process with the help of excerpts from a sample driver `dlpi_enet`.

The install routine of a STREAMS DLPI driver, `driver_install()`, should call CDIO install `install_driver()`(CDIO3) and STREAMS/UX install `str_install()` functions.

The CDIO system requires the following data structures to be defined and initialized before calling `install_driver()` in `driver_install()`.

```c
drv_ops_t dlpi_enet_drv_ops;
drv_info_t dlpi_enet_drv_info = {
    "dlpi_enet",       /* driver name */
    "pseudo",         /* driver class */
    DRV_CHAR | DRV_PSEUDO | DRV_MP_SAFE, /* type */
    -1,                /* block major number */
    -1,                /* character major number */
    NULL, NULL, NULL          /* always NULL */
};
```

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

```c
static struct module_info enet_rminfo = {
    5050, "enetlan", 0, 65536, 65536, 1
};
static struct module_info enet_wminfo = {
    5050, "enetlan", 0, 65536, 1, 1
};
```
Creating Networking Device Drivers
Initializing Networking Device Drivers

static struct qinit enet_rinit = {
    0, enet_rsrv, enet_open, enet_close, 0, &enet_rminfo
};
static struct qinit enet_winit = {
    enet_wput, enet_wsrv, 0, 0, 0, &enet_wminfo
};
struct streamtab enet_info = { &enet_rinit, &enet_winit }
streams_info_t dlpi_enet_str_info = {
    "dlpi_enet",               /* name */
    -1,                       /* dynamic mj # */
    { &enet_rinit, &enet_winit, NULL, NULL },
    /* streamtab */
    STR_IS_DEVICE|MGR_IS_MP|STR_SYSV4_OPEN,
    /* stream flags */
    SQLVL_QUEUE,             /* sync level */
    ""
};
The install routine for the sample driver is shown below. Please note that the install routine should be prefixed with the driver name.

int
dlpi_enet_install(void)
{
    int rv;
bzero((caddr_t)&dlpi_enet_drv_ops, sizeof(drv_ops_t));
    msg_printf("dlpi_enet: install\n");
    if (!(rv = install_driver(&dlpi_enet_drv_info,
        &dlpi_enet_drv_ops))) {
        if(rv = str_install(&dlpi_enet_str_info)) {
            uninstall_driver(&dlpi_enet_drv_info);
            msg_printf("dlpi_enet: install failed\n");
        }
    }
    return rv;
}

WSIO Network Driver Initialization

Each HP-UX PCI networking device driver must have a driver_install (WSIO_DRV) routine and a driver_attach (WSIO_DRV) routine.

If the networking device driver controls and interacts with a hardware device, the driver is also required to have an interrupt service routine to handle the device interrupts.

The following brief descriptions of the required install, attach, and
initialization routines are provided as an introduction to networking device driver initialization.

int

driver_install()  An entry of \texttt{driver_install()} is called during the I/O system configuration process. When the \texttt{driver_install()} routine is called, it hooks the \texttt{driver_attach()} entry on the top of a linked list of attach routines for all of the interface drivers in the system.
#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)
    isc_ptr Pointer to isc_table_type structure.

int driver_isr(struct isc_table_type *isc_ptr, caddr_t cb_ptr)
    isc_ptr Pointer to isc_table_type structure.
    cb_ptr Pointer to driver control block. This cb_ptr is driver developer defined and passed as a pointer through the isrlink() routine during the driver_attach() or driver_init() routines.

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

Networking-specific routines are discussed below.

**Calling driver_install()**

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

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

The following example shows a call to driver_install().

```c
static drv_ops_t enet_drv_ops = {
    NULL,       /* open */
    NULL,       /* close */
}```
Creating Networking Device Drivers

Chapter 7

195

Initializing Networking Device Drivers

NULL, /* strategy */
NULL, /* dump */
NULL, /* psize */
NULL, /* reserved */
NULL, /* read */
NULL, /* write */
NULL, /* ioctl */
NULL, /* select */
NULL, /* option1 */
NULL, /* reserved1 */
NULL, /* reserved2 */
NULL, /* reserved3 */
NULL, /* link */
0,    /* device flags */

};
static drv_info_t enet_drv_info = {
    "enet", /* driver name */
    "lan", /* class name */
    DRV_CHAR | DRV_SAVE_CONF | DRV_MP_SAFE,
    /* driver flags */
    -1, /* block major number */
    -1, /* character major number */
    NULL, NULL, NULL, /* structures always set to NULL */
};

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

static wsio_drv_info_t enet_wsio_info = {
    &enet_drv_info, /* driver info */
    &enet_drv_ops, /* driver ops */
    &enet_data,    /* driver data */
};
extern int (*pci_attach)();
/* to attach PCI driver to system */
int (*enet_saved_attach)();
Creating Networking Device Drivers

Initializing Networking Device Drivers

```c
int enet_install()
{
    enet_saved_attach = pci_attach;
    /* save the current top entry */
    pci_attach = enet_attach;
    /* link attach entry to list */

    return(wsio_install_driver(&enet_wsio_info));

    /*
     * enet_wsio_info is the driver header information that contains
     * the information normally found in /usr/conf/master.d
     */
}
```

**Calling driver_attach()**

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

---

**NOTE**

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

---

When the `driver_attach()` routine recognizes the device ID, it takes actions such as allocating and initializing its driver control blocks and PCI I/O registers. The routine also calls the `pci_read_cfg_uint32_isc()` routine to get the system PCI configuration of the device (if needed). `driver_attach()` also sets up driver initialization call and finally calls the `isc_claim()` to claim the device. A sample `driver_attach()` routine is shown below:

```c
struct gfsw enet_gfsw;
...
...
```
int
#ifdef __LP64__
enet_attach( uint32_t id, struct isc_table_type *isc)
#else
enet_attach( PCI_ID id, struct isc_table_type *isc)
#endif
{
    msg_printf("enet attach id = \%x\n",id);
#endif
    /* Support for PCI only */
#if nidvendor_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 = (diag (*) ())NULL;
#else
    isc->if_id = (int)(id & 0x0000ffffU);
#else
    isc->if_id = (int)id.device_id;
#endif
    isc_claim(isc, &enet_wsio_drv_info);
Creating Networking Device Drivers
Initializing Networking Device Drivers

io_inform("PCI 100 Base TX Controller found", isc, 5);
return enet_saved_pci_attach(id, isc);
}

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

int enet_init(struct isc_table_type *isc)
{
    enet_ift_t *enet_iftp;
...
    ubit32 base_addrp, id, revid, latency_timer, int_reg;
    ubit32 sub_id, ssid, cfda, csr6;
    BUS_TRANS_DESC desc;
    ubit32 error;
...
    /*
     * Allocate driver control block - enet_iftp
     */
    ...
    /*
     * Obtain memory for Transmit and Receive Descriptor Rings and any additional driver data structures*/
    ...
    /*
     * Get/Set PCI configuration
     */
    pci_read_cfg_uint32_isc(isc, SSID, &ssid);
    enet_iftp->sub_id = (ubit16)(ssid >> 16);  
    enet_iftp->sub_vendor_id = (ubit16)(ssid & 0x0000ffff); 

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

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

    /* Read the Configuration Interrupt information */
    pci_read_cfg_uint32_isc(isc, CFIT, &int_reg);
/* Read the Configuration Driver Area information */
pci_read_cfg_uint32_is(isc, CFDA, &cfda);
  cfda = 0;
pci_write_cfg_uint32_is(isc, CFDA, cfda);
  ...

/* Turn on PCI memory access and bus master capability on host */
pci_write_cfg_uint8_is(isc, CFCS, CFCS_MEMORY_SPACE_ACCESS |
CFCS_MASTER_OPERATION |
CFCS_PARITY_ERROR_RESPONSE |
CFCS_SYSTEM_ERROR_ENABLE |
CFCS_I_O_SPACE_ACCESS);
  ...

/* Init and reset the controller */

/* Perform general enet_ift initialization */
  ...

/* Setup hwift structure */
  ...

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

/* Setup interrupt handler. The -1 option will cause
* isrlink to read the configuration interrupt
* register to determine the irq for the PCI device */
isrlink(isc, enet_isr, -1, (long)isc, (long)enet_iftp);
  ...
  ...

Networking Driver-Specific Initialization

If the initialization is successful, the driver_init() routine proceeds with the following steps:
Creating Networking Device Drivers

Initializing Networking Device Drivers

- Initializes the MIB structure and the hw_dlpI and hw_ifT structures (see the preceding sections, “hw_ifT_t Structure Description and Initialization” and “hw_dlpI Structure Description and Initialization”, for details).

- Calls the hw_ifT_attach() routine to link the hw_ifT structure to a global list of hw_ifT structures of active interfaces. The hw_ifT_attach() routine is defined as:

```c
hw_ifT_attach(hw_ifT_t *hw_ifT_ptr)

hw_ifT_ptr Pointer to the associated hw_ifT_t structure
```
Protocol Binding and Demultiplexing

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

To correctly demultiplex inbound packets, a networking driver must:

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

Table 7-1, “Packet Type, Protocol Kind, and Protocol Value,” summarizes the information a networking driver requires to demultiplex inbound packets for corresponding upper-layer protocols. More detailed information is provided in the section “DLPI Sequence” on page 205.

### Table 7-1 Packet Type, Protocol Kind, and Protocol Value

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

Protocol kind is the type of protocol to bind. Interpretation of the protocol value field depends on the protocol kind value. “Protocol Kinds and Values” on page 510, Appendix D, lists the names of the available kinds of protocols to bind and some of the available protocol values.

When the networking driver binds a protocol with protocol kind and type values, the driver knows what kind of packets to handle for that bind. The networking driver processes inbound packets on the interrupt control stack (ICS) for all the protocol binds by calling an associated STREAMS queue. To do this, it calls `putnext()` (see the STREAMS/UX for the HP 9000 Reference Manual) in the device driver’s interrupt service routine. The driver must use the protocol ID that was carried in the dl_proto_info field of the DL_HP_BIND to pass the packet to the
Protocol Binding and Demultiplexing

Protocol Binding and Unbinding

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

Protocol Demultiplexing

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

1. Distinguish packet protocol format and type:
   - Ethernet
   - IEEE 802.2 Link Level Control (LLC) (non-SNAP)
   - SNAP (IEEE 802.2 LLC extended)

2. Locate the proper inbound packet service routine or queue for each valid incoming packet.

Distinguishing Packet Protocol Format

The following information can be used to determine the protocol format and type.

To determine whether the packet is an Ethernet-type packet:

- If the value of the TYPE field of an inbound packet is equal to or greater than 0x600, the packet is an Ethernet-type packet. The protocol kind of the packet is LAN_TYPE, and the protocol value is the TYPE field specified in the packet (see Appendix D, "Protocol Kinds and Values").

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

To determine whether the packet is a SNAP-type IEEE 802.2 LLC packet:
• The packet is considered to be a SNAP packet (defined in IEEE 802.1a) if both the DSAP and the SSAP values are 0xAA. The protocol kind of the packet is LAN_SNAP, the protocol value is 0xAA, and the protocol value extended is the 5-byte SNAP protocol data specified in the SNAP header (see “Protocol Kinds and Values” on page 510, Appendix D).

• Otherwise, it is an IEEE 802.2 LLC non SNAP-type packet. The protocol kind is LAN_SAP and the protocol value is the DSAP field that is specified in the packet (see “Protocol Kinds and Values” on page 510, Appendix D).

The relationships of protocol kind, protocol value, and protocol processing for different types of packets are shown in Table 7-1, on page 201.

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

If the upper layer requires header stripping, the device driver strips off the link-level control (LLC) header before passing the inbound packet to the upstream queue.
Message block and queue_t functions and macros

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

The list of commonly used message block functions.

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

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

- putq() queue message to be processed by queue service procedure
- putnext() call queue's put procedure
- canput() test whether queue can receive message
- qreply() send the message back upstream
- OTHERQ() other queue in the queue pair
- streams_put(), streams_put_release() allow non-STREAMS/UX (e.g. driver ICS) to ‘put’ in a queue
DLPI Sequence

The Data Link Provider Interface (DLPI) specifies a STREAMS-based kernel implementation of the ISO Data Link Service Definition (ISO 8886) and Logical Link Control (ISO 8802/2 LLC). DLPI allows a data link service user to access and use a variety of conforming data link services without special knowledge of the provider’s protocol. The interface specifies access to data link service providers and does not define a specific protocol implementation. HPUX transports (e.g. TCP/IP, UDP, OSI) are now STREAMS modules. Third parties are expected to develop STREAMS DLPI driver conforming to DLPI version 2.0 to support their network interface drivers and can not depend on the HP DLPI implementation. This section provides information about how third party drivers can integrate into a STREAMS/UX framework in HP-UX.

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

Device file, Interface name and PPA number

DLPI users can access DLPI providers through generic DLPI device files (i.e a device file corresponding to a DLPI STREAMS driver). A DLPI device file can be created by mknod(2) or insf(1M) by using device driver information from lsdev(1M). The following example shows the devices enet (sample WSIO PCI driver) and dlpi_enet (sample DLPI STREAMS driver). The device files created for the STREAMS DLPI driver are also shown.

```
# lsdev

......................

| 239 | -1 | enet | lan |
| 240 | -1 | dlpi_enet | pseudo |
```

```
# ls /dev/enet*

lrwxrwxrwx 1 rootsys 11 Apr 12 18:47 /dev/enet ->
```
Creating Networking Device Drivers

DLPI Sequence

/dev/dlpi_enet

# ll /dev/dlpi_enet*
crw-rw-rw- l rootsys 72 0x0000f0 Apr 12 18:46 /dev/dlpi_enet
crw-rw-rw- l rootsys 240 0x000000 Apr 12 18:46 /dev/dlpi_enet0

lanscan(1M) lists all the LAN interfaces in the system from the list of hw_ift_ts (every network interface driver should do hw_iftAttach() during initialization). From this list we can identify the interface name and PPA numbers. Please refer to section, “Initializing Networking Device Drivers” on page 191, for details about hw_iftAttach().

The following output from lanscan illustrates the interface name and PPA numbers for the sample WSIO network driver. As can be seen the sample driver has ‘attached’ to LAN interfaces at hardware paths 8/0/1/0 and 8/0/2/0.

Table 7-2  lanscan Output

<table>
<thead>
<tr>
<th>Hdw Path</th>
<th>Station Address</th>
<th>Card Im#</th>
<th>Hdw State</th>
<th>Net-Interface Name PPA</th>
<th>NM ID</th>
<th>MAC Type</th>
<th>HP-DLPT Support</th>
<th>DLPI Mjr#</th>
</tr>
</thead>
<tbody>
<tr>
<td>8/16/6</td>
<td>0x0060 B07ED BF0</td>
<td>0</td>
<td>UP</td>
<td>lan0 snap0</td>
<td>1</td>
<td>ETHER</td>
<td>Yes</td>
<td>119</td>
</tr>
<tr>
<td>8/0/1/0</td>
<td>0x0060 B07A2 21E</td>
<td>1</td>
<td>UP</td>
<td>enet1</td>
<td>2</td>
<td>ETHER</td>
<td>No</td>
<td>240</td>
</tr>
<tr>
<td>8/0/2/0</td>
<td>0x0060 B0B2D 850</td>
<td>2</td>
<td>UP</td>
<td>enet2</td>
<td>3</td>
<td>ETHER</td>
<td>No</td>
<td>240</td>
</tr>
</tbody>
</table>

IP and ARP Configuration

Once the interface name and the PPA number are known, ifconfig(1M) is used to configure IP and ARP. When ifconfig is done for enet1 listed by
lanscan above, the IP and ARP streams are setup as listed in the steps below.

1. `ifconfig` opens device file `/dev/enet` and understands that PPA configured is 1
2. `ifconfig` issues an ioctl to push IP module on top of `dlpi_enet` driver
3. `ifconfig` issues another ioctl to issue attach and bind request for PPA 1
4. `ifconfig` opens device file `/dev/enet` and issues ioctl to push ARP on top of `dlpi_enet` driver
5. `ifconfig` again performs step 3 for ARP/dlpi_enet stream
6. `ifconfig` opens `/dev/ip` and uses it as dummy multiplexer and IP/dlpi_enet and ARP/dlpi_enet streams are linked under dummy multiplexer.

**STEAMS DLPI and network driver overview**

Figure 7-4, "DLPI Sequence," shows the basic structure of STEAMS DLPI driver implementation in HP-UX. Two main data structures, `enet_hw_ift_t` and `enet_hw_dlpi_t` establish the linkage between the STEAMS driver and the network device driver; it is not an exported interface by HP-UX. This is an example implementation. Third party developers may define their own interface to address their design needs. Please refer to “Initializing Networking Device Drivers” on page 191 for initialization of `enet_hw_ift_t` and `enet_hw_dlpi_t`. 
Figure 7-4  DLPI Sequence

Application Layer
putmsg()
DLS user

getmsg()

User Space

Streams Head
/dev/dlpi and /dev/dlpiX

Network Protocol Layer

Data Link Layer

STREAMS DLPI driver

driver_wput()
driver_wsrv()

 fast_in()

_unitdata_out()

 driver_isr()

(*dlpi_ioctl())
(_control())
INFO, ATTACH,
BIND, PPA_REQ,
_proc_ioctl()

(driver_install())
(driver_attach())
driver_link() driver_init()

WSIO network driver

PCI bus

Kernel Space
The general STREAMS/DLPI buffer/message processing is done in the STREAMS DLPI driver. The network driver implements the device initialization, input, output, and control functions. Modularizing functionality this way makes it possible to support different types of LAN interfaces with one STREAMS DLPI driver. The high level functions in the drivers and their interaction is shown in Figure 7-4 on page 208. The next two sections briefly describe the STREAMS DLPI and network interface drivers.

Synchronization Between Device Driver and DLPI driver

For a non-STREAMS character I/O mechanism, synchronization between device driver and device can be accomplished by having the device driver sleep, via the `sleep()` kernel call, on a unique number, typically, an object address, while waiting for the request to complete. Upon receiving the request completion information from the device, the device driver then resumes the process via the `wakeup()` kernel call. For STREAMS, however, this kind of sleep-wakeup synchronization mechanism is not permitted because STREAMS may run on either the ICS or the STREAMS scheduler context stack. Synchronization between DLPI and the device driver is not defined in the DLPI 2.0 documentation.

The sample DLPI driver has an `enet_dlpi_wakeup()` routine to support the necessary synchronization between DLPI and networking device drivers.

This `enet_dlpi_wakeup()` routine, which simulates the STREAMS environment `wakeup()` kernel call, is defined in `sio/libdlpi.a` as:

```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 `(*dlpi_ioctl)()` routine.

Certain actions are required of the network device driver when device control requests passed through the `(*dlpi_ioctl)()` routine have a negative returned value.

The following rules summarize actions each networking device driver must take in dealing with such `dlpi_ioctl` requests:

1. The control request does one of the following:
Creating Networking Device Drivers

DLPI Sequence

1. If the control request completes immediately with no error, the (*dlpi_ioctl()) routine should immediately return 0 to DLPI.

2. If the control request completes immediately with an error, the error should be returned as a positive value (from errno.h).

3. If the control request cannot complete immediately (that is, the driver must make a request to the hardware), the device driver must hold the enet_hw_ift_lock and return a globally unique negative value to DLPI.

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

3. The device driver completes the previous (*dlpi_ioctl()) by putting the results in the appropriate place for that ioctl.

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

STREAMS synchronization

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

Entering STREAMS from ICS

In order to enter the STREAMS framework correctly from non-STREAMS/UX code, the STREAMS/UX provides streams_put utilities. The driver ICS function can call streams_put(), passing it a function and a queue. STREAMS/UX runs the function as if it were the queue's put routine. The function passed in the call can safety manipulate the queue and access the same data structures as the queue's put routine. streams_put_release() executes the streams_put functionality on a specified processor. Refer to STREAMS/UX for HP 9000 Reference Manual for further information.
STREAMS DLPI Driver

This section discusses the STREAMS DLPI driver briefly. The objective here is to present the code flow of DLPI sample driver, `dlpi_enet`, as background to the sample driver code. Refer to the sample driver code for details. This section contains the following topics:

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

Major Data Structures

**NOTE** These data structures are part of Sample driver. They do not constitute any interface defined by HP-UX.

```c
typedef struct enet_hw_ift {
    hw_dlpi_t    hp_dlpi;
    uint32_t     mac_type;
    uint32_t     llc_flags;
    uint32_t     mjr_num;
    uint32_t     nm_id;
} enet_hw_ift_t;
```

The information in this data structure relates to the network device driver that associates itself with STREAMS DLPI, `dlpi_enet`. There is one such data structure for each network device under `dlpi_enet`. The structure itself is shown below.
Creating Networking Device Drivers

STREAMS DLPI Driver

```c
uint32_t instance_num;
uint32_t mtu;
char *name;
uint8_t hdw_path[MAX_HDW_PATH_LEN];
uint32_t hdw_state;
uint32_t mac_addr_len;
uint8_t mac_addr[MAX_MAC_ADDR_LEN];
uint32_t features;
uint32_t ppa;
uint32_t watch_timer;
lock_t *hwift_lock;
struct enet_ift *next;
)enet_hw_ift_t;
```

The table below lists all the fields along with their purpose.
Table 7-3  enet_hw_ift_t Data Fields

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>hp_dlpi</td>
<td>structure with driver ‘entry’ points (see below)</td>
</tr>
<tr>
<td>mac_type</td>
<td>MAC type - Ethernet, FDDI, ATM see lan_dlpikern.h</td>
</tr>
<tr>
<td>llc_flags</td>
<td>supported LLC encapsulation methods</td>
</tr>
<tr>
<td>mjr_num</td>
<td>major number of the device</td>
</tr>
<tr>
<td>nm_id</td>
<td>get network management ID</td>
</tr>
<tr>
<td>instance_num</td>
<td>return from wsio_isc_to_instance() - WSIO instance number</td>
</tr>
<tr>
<td>mtu</td>
<td>MTU of the link</td>
</tr>
<tr>
<td>name</td>
<td>name of the network driver - sample driver is enet</td>
</tr>
<tr>
<td>hdw_path</td>
<td>hardware path of the network device</td>
</tr>
<tr>
<td>hdw_state</td>
<td>state of the hardware</td>
</tr>
<tr>
<td>mac_addr_len</td>
<td>length of MAC address in octets</td>
</tr>
<tr>
<td>mac_addr</td>
<td>MAC address</td>
</tr>
<tr>
<td>features</td>
<td>features of driver - DRV_* in lan_dlpikrn.h</td>
</tr>
<tr>
<td>ppa</td>
<td>PPA number of the interface -&gt; set to instance number</td>
</tr>
<tr>
<td>watch_timer</td>
<td>watch dog timer support</td>
</tr>
<tr>
<td>hw_ift_lock</td>
<td>MP protection</td>
</tr>
<tr>
<td>next</td>
<td>link to next enet_hw_ift_t</td>
</tr>
</tbody>
</table>

enate_hw_dlpi_t

This structure is part of enet_hw_ift_t and contains driver service functions exported to DLPI driver. It also contains MIB for the network interface.

enate_hw_dlpi_t

{
Creating Networking Device Drivers

STREAMS DLPI Driver

caddr_t drv_data;
intptr_t(*dlpi_output)();
int(*dlpi_build_hdr)();
intptr_t(*dlpi_ioctl)();
caddr_tmib_ptr;

int mib_len;
}

Table 7-4 enet_hw_dlpi_t Data Fields

<table>
<thead>
<tr>
<th>field name</th>
<th>purpose of the field</th>
</tr>
</thead>
<tbody>
<tr>
<td>drv_data</td>
<td>Driver dependent data structure to be passed while calling driver 'entry' points</td>
</tr>
<tr>
<td>dlpi_output</td>
<td>Output routine for packet</td>
</tr>
<tr>
<td>dlpi_build_hdr</td>
<td>Routine provided for building header</td>
</tr>
<tr>
<td>dlpi_ioctl</td>
<td>Routine provided by driver to process device dependent part of DLPI primitives</td>
</tr>
<tr>
<td>mib_ptr</td>
<td>Pointer to mib stats</td>
</tr>
<tr>
<td>mib_len</td>
<td>Byte count of mib data pointed by mib_ptr</td>
</tr>
</tbody>
</table>

enet_dlpi_data_t

This data structure contains STREAMS DLPI driver information for a Stream that is open currently with the driver.

typedef struct _enet_dlpi {
    enet_hw_ift_t*hwiftp;
    cred_t*cred;
    queue_t*queue_ptr;
    dev_tenet_dev;
    uint32_t dlsap_addr_length;
    uint8_t dlsap_addr[MAX_DLSAP_LEN];
    uint16_t service_mode;
};
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 lists the fields along with explanation.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>hwiftp</td>
<td>The interface that associated with this open stream</td>
</tr>
<tr>
<td>cred</td>
<td>credential structure of the user who opened this stream</td>
</tr>
<tr>
<td>queue_ptr</td>
<td>queue pointer to the read queue of the stream</td>
</tr>
<tr>
<td>enet_dev</td>
<td>dlpi_enet device number</td>
</tr>
</tbody>
</table>
### Table 7-5

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>dlsap_addr_length</td>
<td>length of DLSAP address</td>
</tr>
<tr>
<td>dlsap_addr[]</td>
<td>MAC addr + SAP</td>
</tr>
<tr>
<td>service_mode</td>
<td>Only DL_CLDLS supported in the sample driver</td>
</tr>
<tr>
<td>curr_state</td>
<td>DLPI state</td>
</tr>
<tr>
<td>xidtest_flag</td>
<td>dl_xidtest_flg from DL_BIND_REQ; indicates to the driver that XID and/or TEST responses for this stream are to be generated by DLPI driver</td>
</tr>
<tr>
<td>mac_type</td>
<td>interface MAC type</td>
</tr>
<tr>
<td>mac_mtu</td>
<td>interface MTU</td>
</tr>
<tr>
<td>dlsap_ptr</td>
<td>dlsap_t structure list of logged SAPs</td>
</tr>
<tr>
<td>ssap</td>
<td>first SAP logged on stream</td>
</tr>
<tr>
<td>sxsap</td>
<td>first extended SAP logged on stream</td>
</tr>
<tr>
<td>enet_mcast_list</td>
<td>list of multicast addresses on this stream</td>
</tr>
<tr>
<td>promiscuous_flag</td>
<td>this flag is set to the promiscuous level specified in the DL_PROMISCON_REQ primitive</td>
</tr>
<tr>
<td>promisc_filter</td>
<td>this field is set to 1 if the stream has been bound with any SAP</td>
</tr>
<tr>
<td>noloopback_flag</td>
<td>This is set when application wants to handle loopback. This flag is set when DLPI_SET_NOLOOPBACK ioctl is issued. DLPI turns on the MSGNOLOOP flag in mblk message on every outbound message so that driver won't loop back the packet</td>
</tr>
<tr>
<td>no_src_routing</td>
<td>Set when DLPI_NO_SRC_ROUTING is issued</td>
</tr>
<tr>
<td>arp_stream</td>
<td>set if this is ARP stream</td>
</tr>
<tr>
<td>ip_stream</td>
<td>set if this is IP stream</td>
</tr>
<tr>
<td>fast_path</td>
<td>set if application requests to set up fast path</td>
</tr>
</tbody>
</table>
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 mode details of clone driver). The minor number is set to the real major number of the device. The clone open is useful because the application does not need to keep track of which minor number is available and does not need to deal with multiple device files.

As can be seen from the following example, `/dev/dlpi_enet` is a clone device of `dlpi_enet` driver. While `/dev/dlpi_enet0` is a regular device file with a device major number, 240, and minor number, 0.

```
# lsdev
............................................
............................................
239          -1 enet  lan
240          -1 dlpi_enet pseudo
# ll /dev/enet*
lrwxrwxrwx   1 root       sys 11 Apr 12 18:47 /dev/enet -> /dev/dlpi_enet
# ll /dev/dlpi_enet*
```

### Table 7-5 enet_dlpi_data_t Data Fields

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>fast_path_pkt_type</td>
<td>is the fast path packet type</td>
</tr>
<tr>
<td>fast_path_llc_length</td>
<td>is the LLC header length used in the fast path</td>
</tr>
<tr>
<td>pre_state</td>
<td>keeps the state before a pending ioctl or control request with the driver, so that when the request is complete the streams can be set to correct state</td>
</tr>
</tbody>
</table>
When a clone device is opened, the clone driver invokes the DLPI driver’s open routine with the CLONEOPEN flag set. The open function `dlpi_enet_open()` allocates the `enet_dlpi_data_t` for the stream being opened and initializes it. The minor number of a normal device file open is used as the index into `enet_dlpi_data_ptr_arr[]` to store and access `enet_dlpi_data_t` for the stream. The indexes 1 to 99 are reserved for regular open in sample driver. For clone opens, an unused minor number starting from 100 is allocated. The `enet_dlpi_data_t` for the stream is stored in the `enet_dlpi_data_ptr_arr` indexed by new minor number.

**Control Functions**

The function `dlpi_enet_wput()`, the streams put procedure, calls various control functions to service DLPI M_PROTO and M_PCPROTO messages with local management primitives (information reporting, attach, bind, and others like multicast, promiscuous). This function consists of a switch table that calls the service function based on message `dl_primitive`. The following is a list of service functions and their function:
### Table 7-6  Message Service Functions

<table>
<thead>
<tr>
<th>Function Name [prefixed by enet_dlpi]</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>_info()</td>
<td>Service function for DL_INFO_REQ. The information is returned upstream in structure dl_info_ack_t. If the PPA is not attached yet, then mac type and mtu is set to DL_CSMACD and IEEE8023_MTU.</td>
</tr>
<tr>
<td>_multicast_list()</td>
<td>This function is called to service the DL_HP_MULTICAST_LIST_REQ primitive. In turn this function calls driver dlpi_ioctl() to get the list by passing the command DL_HP_GET_MIB_STATS.</td>
</tr>
<tr>
<td>_ppa_req()</td>
<td>Receipt of DL_HP_PPA_REQ results in calling this function. hw_if_t_list is searched for this PPA and the information from hw_if_t is returned.</td>
</tr>
<tr>
<td>_attach()</td>
<td>The information about PPA to be attached is found from hw_if_t_list; dlpi_ioctl() is issued to the driver with primitive DL_HP_HDW_INIT. The enet_dlpi_data_t for this stream is updated with network interface information and the stream DLPI state.</td>
</tr>
<tr>
<td>_detach()</td>
<td>Disable all multicasts that were enabled through this stream by issuing dlpi_ioctl()s to the network driver. If promiscuous mode was enabled by this stream then disable it. clean_str_spu_sw_q() is called to clean up any requests in the STREAMS/UX. Finally update the state in enet_dlpi_data to DL_UNATTACHED.</td>
</tr>
<tr>
<td>_bind()</td>
<td>DL_BIND_REQ primitive request indicates to bind a DLSAP to the stream. Protocol kind (LAN_TYPE, LAN_SNAP or LAN_SAP) is determined by SAP value in the request. Driver ioctl request is setup and DL_HP_BIND command is issued to the driver. Once driver bind is successful, dlsap_t is allocated and initialized with protocol type and value of SAP. enet_dlpi_data_t for this stream is updated with this bind details.</td>
</tr>
</tbody>
</table>
Creating Networking Device Drivers
STREAMS DLPI Driver

Table 7-6 Message Service Functions

<table>
<thead>
<tr>
<th>Function Name [prefixed by enet_dlpi]</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>_unbind()</td>
<td>Driver ioctl request is setup for unbind and DL_HP_UNBIND is issued. dlsap_t is deallocated and the information in enet_dlpi_data_t about the bind is set to default value.</td>
</tr>
<tr>
<td>_subs_bind()</td>
<td>Upon receipt of DL_SUBS_BIND_REQ, this function is called. Driver ioctl request is setup and DL_HP_BIND command is issued to the driver. If the dl_subs_bind_class is DL_PEER_BIND, then a new dlsap_t is allocated and initialized with protocol type and value of SAP. With DL_HEIRARCHICAL_BIND the dlsap_addr information in enet_dlpi_data_t is updated with bind details.</td>
</tr>
<tr>
<td>_subs_unbind()</td>
<td>For each dlsap_t bound, compare the unbind request SAP, and if there is, match the issue driver DL_HP_UNBIND.</td>
</tr>
<tr>
<td>_xidtest_out()</td>
<td>This function services DL_TEST_REQ, DL_TEST_RES, DL_XID_REQ and DL_XID_RES. It builds the test/xid packet and sends it to the driver using dlpi_output().</td>
</tr>
<tr>
<td>_control()</td>
<td>The primitives serviced by this function are - DL_ENABMULTI_REQ, DL_DISABMULTI_REQ, DL_SET_PHYS_ADDR_REQ, DL_PROMISCON_REQ, DL_PROMISCOFF_REQ and DL_HP_HW_RESET_REQ. The respective ioctl commands are issued to driver via dlpi_ioctl(). If the request didn't complete immediately then this routine sleeps on address of the sleep object of the dlpi_ioctl().</td>
</tr>
<tr>
<td>_status()</td>
<td>Send hw_if-&gt;hdw_state upstream in response to DL_HP_HW_STATUS_REQ request.</td>
</tr>
</tbody>
</table>
Creating Networking Device Drivers
STREAMS DLPI Driver

Chapter 7

Table 7-6 Message Service Functions

<table>
<thead>
<tr>
<th>Function Name [prefixed by enet_dlpi]</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>_getphyaddr()</td>
<td>Requests the network driver ioctl command DL_HP_GET_PHYS_ADDR, which gets the permanent ROM physical address of the network interface, to service DL_PHYS_ADDR_REQ.</td>
</tr>
<tr>
<td>_get_mibstats()</td>
<td>Request the network driver, through the driver ioctl command, to get standard MIBs; this is to service the DLPI primitive DL_GET_STATISTICS_REQ.</td>
</tr>
<tr>
<td>_get_mib_req()</td>
<td>Services MC_GET_MIB_REQ (sys/mci.h). The driver ioctl DL_HP_GET_STATISTICS is issued to get current MIB statistics.</td>
</tr>
<tr>
<td>_set_mib_req()</td>
<td>Services MC_SET_MIB_REQ. The driver ioctl DL_HP_RESET_STATS is issued to reset the MIB statistics.</td>
</tr>
</tbody>
</table>

IOCTL Processing

STREAMS/UX provides the capability for user processes to perform control functions by using ioctl calls on device drivers in a stream. These commands cause the stream head to create an M_IOCTL message that includes the ioctl arguments and to send the message downstream to be received and processed by a device driver. The streams put function 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 DL_UNITDATA_REQ primitive. The LLC header format is built for the specific interface in a new M_DATA message block and is linked to M_PROTO, and the whole complex message is sent back to the application.
Creating Networking Device Drivers
STREAMS DLPI Driver

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

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

Transmission of Message blocks

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

Regular Data Path

The regular data path message transmission works as follows. The
streams put function dlpi_enet_wput() receives the
DL_UNITDATA_REQ primitive request from the application to send a
message to a destination specified in the unitdata message.
dlpi_enet_wput() calls the enet_dlpi_unitdata_out() function to
service the request. enet_dlpi_unitdata_out() applies sanity checks
for the stream's DLPI state and the request parameters and builds the
LLC header. The LLC header message block is linked with the first
M_DATA (with DL_UNITDATA_REQ) and proceeds with calling the
driver's output routine dlpi_output() in enet_hw_dlpi structure.

Fast Path

For better performance, fast path is used to transmit and receive data.
The basic idea of fast path is that the DLPI user sends DLPI ioctl
DLPI_IOC_HDR_INFO to set up the fast path on the stream. The DLPI
builds an LLC header template and sends it back to the user. Then, for
an outbound packet, the user prepends the link header to the data, based
on the template of the link header, and sends M_DATA messages to
DLPI. DLPI passes this packet to the network driver without going
through the process of building the link header. Similarly, for an inbound
packet on the fastpath stream, DLPI strips off the LLC header and
passes it to the user without building and prepending the
DL_UNITDATA_IND primitive to the data.
Reception of Message blocks

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

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

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

`enet_dlpi_unitdata_in()` 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 the “Fast Path” section above.

Summary of DLPI Primitives and IOCTLs

The following table summarizes the DLPI primitives and IOCTLs that have been dealt with in the sample drivers, along with appropriate comments. Note that the processing of most DLPI primitives and IOCTLs involves driver interaction. This is discussed in the next section.

<table>
<thead>
<tr>
<th>Table 7-7</th>
<th>DLPI Primitives and IOCTLs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DLPI Primitive or IOCTL</strong></td>
<td><strong>Comments</strong></td>
</tr>
<tr>
<td>DL_INFO_REQ</td>
<td>Information reporting</td>
</tr>
<tr>
<td>DL_INFO_ACK</td>
<td></td>
</tr>
<tr>
<td>DL_ERROR_ACK</td>
<td></td>
</tr>
<tr>
<td>DLPI Primitive or IOCTL</td>
<td>Comments</td>
</tr>
<tr>
<td>--------------------------</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_OK_ACK</td>
<td></td>
</tr>
<tr>
<td>DL_ERROR_ACK</td>
<td></td>
</tr>
<tr>
<td>DL_BIND_REQ</td>
<td>Bind</td>
</tr>
<tr>
<td>DL_BIND_ACK</td>
<td></td>
</tr>
<tr>
<td>DL_SUBS_BIND_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_SUBS_BIND_ACK</td>
<td></td>
</tr>
<tr>
<td>DL_UNBIND_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_SUBS_UNBIND_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_OK_ACK</td>
<td></td>
</tr>
<tr>
<td>DL_ERROR_ACK</td>
<td></td>
</tr>
<tr>
<td>DL_ENABMULTI_REQ</td>
<td>Other</td>
</tr>
<tr>
<td>DL_DISABMULTI_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_PROMISCON_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_PROMISCOFF_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_PHYS_ADDR_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_GET_STATISTICS_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_OK_ACK</td>
<td></td>
</tr>
<tr>
<td>DL_ERROR_ACK</td>
<td></td>
</tr>
<tr>
<td>DL_UNITDATA_REQ</td>
<td>DLPI Ver 2.0 Connection less Data transfer</td>
</tr>
<tr>
<td>DL_UNITDATA_IND</td>
<td></td>
</tr>
<tr>
<td>HP EXTENDED DLPI PRIMITVES</td>
<td></td>
</tr>
</tbody>
</table>
Table 7-7 DLPI Primitives and IOCTLs

<table>
<thead>
<tr>
<th>DLPI Primitive or IOCTL</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL_HP_MULTICAST_LIST_REQ</td>
<td>These are HP extensions to DLPI 2.0 and may change. They are defined in sys/dlpi_ext.h</td>
</tr>
<tr>
<td>DL_HP_PPA_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_HP_OK_TO_LOG_SNAP</td>
<td></td>
</tr>
<tr>
<td>DL_HP_HW_RESET_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_HP_HW_STATUS_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_HP_RESET_STATS_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_HP_RESET_STATS_REQ</td>
<td></td>
</tr>
<tr>
<td>MC_GET_MIB_REQ</td>
<td>These HP specific IOCTLs and may change. They are defined in sys/mci.h</td>
</tr>
<tr>
<td>MC_SET_MIB_REQ</td>
<td></td>
</tr>
<tr>
<td>HP IOCTLs</td>
<td></td>
</tr>
<tr>
<td>DLPI_SET_NOLOOPBACK</td>
<td>These HP specific IOCTLs and may change. They are defined in sys/dlpi_ext.h</td>
</tr>
<tr>
<td>DLPI_IOC_HDR_INFO</td>
<td></td>
</tr>
<tr>
<td>DLPI_IOC_DRIVER_OPTIONS</td>
<td></td>
</tr>
</tbody>
</table>
Creating Networking Device Drivers
WSIO network driver

WSIO network driver

This section briefly explains the code flow of a network driver. Please refer to Figure 7-4 on page 208 for network driver major functions and their interaction with the DLPI driver. Refer to the sample code for the details.

Data Structures

enet_ift

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

```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 contrl. 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
******************************/
etnet_tb_t*tbr;/* Transmit buffer Ring */
etnet_td_t *tdr;/* Transmit Descriptor Ring */
...
...
/******************************
Receive Section - RX SECT
******************************/
etnet_rd_t*rdr;/* Receive Descriptor Ring */
etnet_rb_t*rbr;/* Receive buffer Ring */
...
...
/******************************
Full Duplex, speed and Transmit Threshold setting - SETTINGS
******************************/
...
...
/******************************
Local Driver Receive Stats - STATS
******************************/
rcv_stats_t rstats;/* Receive Statistics*/
******************************
Local Driver Transmit Stats - STATS
Creating Networking Device Drivers

WSIO network driver

***************************************************************/
trx_stats_ttstats;/* Transmit Statistics*/
***************************************************************/

Mib Specific Section
*******************************************************************************/
mib_xEntrymib_xstats;
mib_Dot3StatsEntrydot3_ext_stats;
mib_Dot3CollEntrydot3_ext_coll;
*******************************************************************************/
* Misc
*******************************************************************************/
...
...
*******************************************************************************/
lock_t*enet_r_lock;
*******************************************************************************/
et_h fw_if t enhwift;
*******************************************************************************/
} enet_if t_t;

Table 7-8  enet_if t Data Fields

<table>
<thead>
<tr>
<th>Field Name/Generic Description</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>lancift</td>
<td>Contains generic LAN information</td>
</tr>
<tr>
<td>PCI INFO</td>
<td>Has PCI configuration information</td>
</tr>
<tr>
<td>DEV REG</td>
<td>Fields have Control and Status Register addresses</td>
</tr>
<tr>
<td>DEV SPEC</td>
<td>Device specific information, such as ISC structure, Serial ROM data, driver states, and cable state</td>
</tr>
</tbody>
</table>
Table 7-8  enet_ift Data Fields

<table>
<thead>
<tr>
<th>Field Name/Generic Description</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>TX SECT : tbr, tdr..</td>
<td>This set of fields contain transmit buffers, transmit descriptors, and counters</td>
</tr>
<tr>
<td>RX SECT : rbr,rdr..</td>
<td>This set of fields contain receive buffers, receive descriptors, and counters</td>
</tr>
<tr>
<td>SETTINGS</td>
<td>Full duplex, link speed, selected connection type, and transmit threshold settings</td>
</tr>
<tr>
<td>STATS</td>
<td>Driver local Receiver and Transmitter statistics</td>
</tr>
<tr>
<td>mib_xstats</td>
<td>MIB objects (RFC 1066/1156) and additional counters</td>
</tr>
<tr>
<td>dot3_ext_stats</td>
<td>Extended MIB statistics</td>
</tr>
<tr>
<td>dot3_ext_coll</td>
<td>Extended MIB collisions</td>
</tr>
<tr>
<td>enet_r_lock</td>
<td>Lock for accessing enet_ift</td>
</tr>
</tbody>
</table>

**enet_ift**

enet_ift holds generic LAN information for the network interface. The structure is shown below and table explains about fields.

typedef struct{
  enet_hw_ift_t hwift;
  lan_timerlantimer;
  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;
} enlan_ift;
Creating Networking Device Drivers
WSIO network driver

```c
uint32_t RXD_SPECIAL_DROPPED;
short int is_scaninterval;
/* Configuration info */
int num_multicast_addr;
int broadcast_filter;
int multicast_filter;
enlanc_promisc_type_t promiscuous_filter;
int hdw_initialized;
uint8_t mcast[96];
uint32_t mcast_ref_cnt[16];
mib_xEntry *mib_xstats_ptr;
lock_t* enlanc_lock;
} enlan_ift;
```

Table 7-9  enlan_ift Data Fields

<table>
<thead>
<tr>
<th>Field Name/Generic Description</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>hwift</td>
<td>Generic Hardware information</td>
</tr>
<tr>
<td>lantimer</td>
<td>DMA/Control timer to track if a DMA or control operation is taking too long</td>
</tr>
<tr>
<td>hw_req()</td>
<td>h/w interface request function pointer</td>
</tr>
<tr>
<td>dma_time()</td>
<td>DMA timeout error handling</td>
</tr>
<tr>
<td>STATUS &amp; STAT</td>
<td>More statistics</td>
</tr>
<tr>
<td>num_multicast_addr</td>
<td>Number of multicast addresses active</td>
</tr>
<tr>
<td>broadcast_filter</td>
<td>Read packet filters</td>
</tr>
<tr>
<td>multicast_filter</td>
<td></td>
</tr>
<tr>
<td>promiscous_filter</td>
<td></td>
</tr>
<tr>
<td>mcast, mcast_ref_cnt</td>
<td>Multicast addresses and their reference count</td>
</tr>
</tbody>
</table>
For each bind, the network driver keeps track of the bound SAPs and relevant information about the bind. The following structures are used to maintain this information.

```c
struct logged_info{
    int protocol_val[5];
    void (*rint)();
    caddr_t data_ptr;
    uintptr_t lu_protocol_info;
    int flags;
}
```

Table 7-9  
**enlan_ift Data Fields**

<table>
<thead>
<tr>
<th>Field Name/Generic Description</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>mib_xstats_ptr</td>
<td>MIB object</td>
</tr>
<tr>
<td>enlac_lock</td>
<td>Lock to access enlac_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.

```c
struct logged_info{
    int protocol_val[5];
    void (*rint)();
    caddr_t data_ptr;
    uintptr_t lu_protocol_info;
    int flags;
};
```

Table 7-10  
**Bound SAP Data Fields**

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>protocol_val</td>
<td>SAP, Type, or Canonical value</td>
</tr>
<tr>
<td>rint()</td>
<td>pointer to DLPI input routine</td>
</tr>
<tr>
<td>data_ptr</td>
<td>Driver control block (enet_if)</td>
</tr>
<tr>
<td>lu_protocol_info</td>
<td>queue pointer of the stream which did the bind</td>
</tr>
<tr>
<td>flags</td>
<td>LANC_ON_ICS and LANC_STRIP_HEADER bits</td>
</tr>
</tbody>
</table>

The following structure is used to link the logged_infos.

```c
struct logged_link{
```
Creating Networking Device Drivers

WSIO network driver

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

**IOCTLS**

The DLPI layer uses the driver ioctl function available in `enet_hw_dlpi` to communicate the device-dependent DLPI primitives to the device driver for processing. Refer to "Writing a driver_ioctl() Routine" on page 131, Chapter 5, for details about the `dlpi_ioctl()` function. Essentially the DLPI driver calls this function with a driver control block `enet_if', the ioctl command, and the message block with request data. The function pointer `dlpi_ioctl` is initialized to `enet_ioctl()` during initialization by the network driver.

The table below summarizes the driver IOCTL commands and the processing by the network driver.

**Table 7-11**  
**Driver IOCTL Commands and Processing**

<table>
<thead>
<tr>
<th>IOCTL</th>
<th>Processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL_HP_BIND</td>
<td>This command binds a DLSAP to the stream. Driver logs the bind specifics in <code>logged_info</code> structure and calls <code>enlanc_log_protocol()</code> to add to the list of bound SAPs.</td>
</tr>
<tr>
<td>DL_HP_UNBIND</td>
<td>Driver removes the previously bound DLSAP from the list of <code>logged_info</code> structures using <code>enlanc_remove_protocol()</code> function.</td>
</tr>
<tr>
<td>DL_HP_LOOKUP_PROTO</td>
<td>Driver checks that a DLSAP is bound to the interface. This is done by using function <code>enlanc_lookup()</code>.</td>
</tr>
</tbody>
</table>
### Table 7-11  
**Driver IOCTL Commands and Processing**

<table>
<thead>
<tr>
<th>IOCTL</th>
<th>Processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL_HP_ENABLMULTI</td>
<td>Call enlan_media_control() process this request; this function checks for validity of multicast address. Otherwise ADD_MCAST is called.</td>
</tr>
<tr>
<td>DL_HP_DISABMULTI</td>
<td>Same processing as above, except that the multicast address is removed.</td>
</tr>
<tr>
<td>DL_HP_PROMISON</td>
<td>Driver calls enlan_media_control() to enet_hw_req() to enet_ctl_req() to enable promiscuous to 'update' the current promiscuous filter.</td>
</tr>
<tr>
<td>DL_HP_PROMISCOFF</td>
<td>Same as above, except we are updating/removing the promiscuous filter.</td>
</tr>
<tr>
<td>DL_HP_GET_PHYS_ADDR</td>
<td>The functions are called in the following order - enet_hw_req(), enet_ctl_req(), enet_get_stats(). Physical address is returned.</td>
</tr>
<tr>
<td>DL_HP_SET_PHYS_ADDR</td>
<td>Same as above. Local address is copied with new address.</td>
</tr>
<tr>
<td>DL_HP_RESET_STATS</td>
<td>The function called are - enlan_media_control(), enet_hw_req(), enet_ctl_req(), and enet_ext_clearmib() to clear MIB.</td>
</tr>
<tr>
<td>DL_HP_GET_STATISTICS</td>
<td>The functions called are - enet_hw_req(), enet_ctl_req(), and enet_ext_mibstats() to get MIB statistics.</td>
</tr>
</tbody>
</table>
Creating Networking Device Drivers

WSIO network driver

Table 7-11

Driver IOCTL Commands and Processing

<table>
<thead>
<tr>
<th>IOCTL</th>
<th>Processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL_HP_HW_RESET</td>
<td>The following functions are called in order: enlanc_media_control(), enet_hw_req(), enet_ctl_req(), and enet_reset() to perform hardware reset.</td>
</tr>
</tbody>
</table>

Outbound Path

The enet driver write path starts with enet_output() called from the DLPI layer through the enet_hw_dlpi->dlpi_output function pointer.

enet_output()

This function calls enet_hw_req() to handle the write request.

enet_hw_req()

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

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

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

enet_slow_hw_req()

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

ENET_TRANSMIT_FRAME

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

enet_transmit_complete()

enet_process_complete() processes transmit complete interrupts. Call enet_transmit_slow_complete() to process non-unicast frames or setup frames. Transmit error handling is done by calling enet_transmit_error(). If there are frames queued for transmission, call enet_transmit_pending_frames() to restart transmission.

enet_transmit_pending_frames()

While there are frames pending transmission, map the frames, setup the transmit descriptors, and issue transmit poll to the device to transmit the frames.

Inbound Path

The enet read path is on the ICS. enet_isr() is called when the network interface's PCI interrupt is received and enet_receive_frame() is invoked to process received frames.

enet_receive_pkts()

This function is called from the receive interrupt handler. Some sanity checking is done on the received frames to determine if they are good. The message block chain is constructed from receive descriptor. If the driver state is ENET_ONLINE, call enet_process_packet() to process the frame. Otherwise, call enet_process_looper() to process the frame. Replenishing the receive descriptor ring with buffers is done while doing frame receive processing.

enet_process_packet()

This function determines that the frame header is Ethernet or IEEE 802.2 and enlanc_ether_ics() or enlanc_802_2_ics() is called.
Creating Networking Device Drivers

WSIO network driver

accordingly.

**enet_process_looper()**

This function processes the loopback packet. The current driver sub state determines the action taken. The packet buffer is validated but not used and discarded.

**enlanc_802_2_ics()**

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

**enlanc_ether_ics()**

Protocol kind (LAN_TYPE, LAN_SNAP or LAN_SAP) and protocol value are extracted from the received packet. If the interface supports promiscuous mode and if it is set, route the packet to all streams qualified for the set promiscuous level, using `enlanc_route_promisc()`. The lookup for logged DLSAPs is `enlanc_lookup()`, and if there is a match, this routine sends the packet to the logged stream by calling the function registered during the bind.
DLKM drivers

DLKM driver support may be provided in future.
Platform Specifics

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

V Class

The following brief overview of the V class PCI I/O architecture provides a good background for driver writers porting a driver to V Class Systems.

EPIC is the bridge between the PCI bus and Processors, Memory, and interconnections. Two types of host memory are accessible by an I/O card DMA transaction—non-coherent shared memory on the EPIC bridge and channel-based access to coherent system memory. Multiple channels are available to PCI slots or card functions. Driver instances related to different slots will not share a DMA channel or steal resources from each other. The I/O card cannot access any non-coherent address space beyond EPIC.

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

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

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

• Allocate transmit and receive descriptor memory in shared memory wsio_allocate_shared_memory() so they can be accessed faster.
• Shared memory does not need wsio_map() since it is already both virtually and physically contiguous.
The code examples below illustrate use of the function
\texttt{wsio\_allocate\_shared\_memory()}.
\texttt{V} class specific enhancements are
with \texttt{#ifdef V\_CLASS} macro and \texttt{if\ (is\_SPP)} \texttt{\{} code blocks.

\texttt{\/* 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 \texttt{driver enet\_c} for more details. Look for \texttt{#ifdef V\_CLASS}
or \texttt{if(is\_SPP())} statements.
*/
enet\_init (struct isc\_table\_type *isc) {
  enet\_ift\_t *enet\_iftp;
  size\_t size;
  u\_long phys\_base;
  ...
  ...
  \texttt{\/* size: initialized to the size of enet\_iftp->tdr}
  (transmitter descriptor ring)
  */
  ...
  err = wsio\_allocate\_shared\_memory(isc, size,
    \texttt{(caddr\_t \*)&phys\_base},
    \texttt{(caddr\_t \*)&enet\_iftp->tdr},
    0);
  VASSERT(err == SHMEM\_OK);
  ...
  ...
}
Network Management Support

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

Hewlett-Packard moved from a single monolithic agent to a variable number of agents, called subagents. Whenever a new driver is added to a system, a user-space subagent specific to this driver is also supplied. This subagent provides the MIB instrumentation needed to access the MIB objects associated with the driver. Figure 7-5 shows the master agent/subagents relationship and partitioning of the subagents. The assumption now is that whoever supplies the new driver will also supply the subagent for that driver.

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

The new Network Management interface will be user-based, contained completely within a user-space library (libnm.a), and in general will have a one-to-one mapping to the calls provided by the old Network Management interface.
In replacing the /dev/netman, the following ioctls will not be available.

- NMIOGET
- NMIOSET
- NMIODEL
- NMIOCRE
- NMPEEK
- NMPOKE
Creating Networking Device Drivers

Network Management Support

Network Management services are to be used by STREAMS-based networking interfaces that provide an ifEntry in the MIB-II ifTable object (see below for ifEntry struct in sys/mib.h.) In the 4.3 Berkeley based networking stack, the ifTable was directly tied to the global ifnet structure list. When a networking interface registered an ifnet structure via if_attach, an ifIndex value was returned which was to be used in the MIB-II ifEntry object managed by that networking interface. The ifTable was known by the TCP/IP protocol stack and all interfaces to retrieve the ifTable and a specific ifEntry was through the TCP/IP protocol stack. With the movement to a STREAMS-based TCP/IP protocol stack, the global ifnet structure list no longer exists and, therefore, the global ifTable management no longer exists.

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

u_int32 get_nmid

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

u_int32 get_nmid() > 0 indicates that the call succeeded and the value returned is the ifIndex value.

<=0 indicates that the request failed to allocate an ifIndex value.

Example code in enet driver:

enet_iftp->lancift.hwift.nm_id = get_nmid();

u_int32 return_nmid

Return a previously assigned ifIndex to the pool of available ifIndex values. This network management service should be called by all kernel entities that own an ifIndex value before it is unloaded from the system.

u_int32 return_nmid (u_int32 ifIndex)

ifIndex: ifIndex value to be returned to the pool of available ifIndex values.
<0 indicates that the ifIndex value being returned was not the previously assigned ifIndex value.

>=0 Indicates the ifIndex was successfully returned to the pool.

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

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 struct mib_ifEntry above in the appropriate place so any application can retrieve information for
Creating Networking Device Drivers

Network Management Support

using the Network Management Support services interface.
Configuring a Networking Driver Through SAM

NOTE

SAM does not support native DLPI at this point. This option will be considered in the future.
Creating Networking Device Drivers

Network Monitoring Commands: driveradmin, driverlinkloop and lanscan

The lanscan command that comes bundled with HP-UX works with third-party device drivers. However, the default lanadmin and linkloop commands are not supported on third-party drivers. It is the responsibility of the driver developer to provide these utilities. The following sections document sample utilities, enetlinkloop and enetadmin. A framework for these utilities is presented in Appendix F for the benefit of driver developers.

It is recommended that the utilities be named driveradmin and driverlinkloop to avoid conflict with the default HP-supplied commands. Also it is recommended that these third party utilities be located in the /opt directory on the target machine.

The driveradmin and lanscan commands use shared libraries to display certain interface-specific network data. Developers are responsible for writing these shared libraries if they want driveradmin and lanscan to work with their new device drivers.

Examples of shared libraries that allow driveradmin and lanscan to work with the enet driver and skeleton code for driveradmin and driverlinkloop are available in Appendix D, “Shared Library Examples for the driveradmin and lanscan Commands.”

**driveradmin command**

The driveradmin command allows the user to perform various administrative tasks on a specified LAN interface. To perform most network administrative tasks, the driveradmin command executes the same basic program regardless of the interface. The task of displaying the interface statistics, which can vary from interface to interface, has been put into shared libraries. One shared library is available for each networking driver to work with driveradmin.

**Invoking a Shared Library to Display Statistics**

The shared library is invoked by driveradmin when the user selects the 3rd party option to input a third party’s special device filename. Then the user selects the Display command in the LAN Interface Test Mode menu. The shared library invoked by the Display command is determined by
the Physical Point of Attachment (PPA) that has been selected either implicitly by default or explicitly by the user.

The `driveradmin` routine then completes the steps below:

1. `driveradmin` implicitly selects a PPA by getting the first element of the PPA list. The first element becomes the default PPA.

2. After getting the PPA, `driveradmin` attaches to it. By attaching to the PPA, `driveradmin` can get the driver name that was returned in the attach routine.

   The name of the driver returned in the attach originates from the string that is stored by the driver at initialization time in the name element of the `hw_ift` structure. See “`hw_ift_t` Structure Description and Initialization” on page 184 for a description of this structure.

3. `driveradmin` uses this driver name to determine the name of the shared library file to access by doing a `shl_load()` of a file that has been named, in the following form:

   `/usr/lib/lanadmin/libdsdriver_name.sl`

   where `driver_name` is the string stored in the name element of the `hw_ift` structure. Every other part of the full path name above is hardcoded in `driveradmin`.

4. After loading the shared library, `driveradmin` again uses the driver name to determine the name of the shared library function to use, by doing a `shl_findsym()` of a function name with the following form:

   `dsdriver_name`

   where `driver_name` is as described above. The `ds` stands for “display statistics.”

5. `driveradmin` then uses the handle returned by `shl_findsym()` to invoke the shared library.

Right after the user selects the Display command of the LAN Interface Test Mode menu and just before invoking the shared library function, `driveradmin` displays the status display title, the date and time, and the first line of the statistics, which is always the PPA. This output resembles the following:

```
LAN INTERFACE STATUS DISPLAY
Tue, Jun 1, 1999 10:47:37
PPA Number = 1
```
Creating Networking Device Drivers

Network Monitoring Commands: driveradmin, driverlinkloop and lanscan

Arguments Passed to Shared Library Functions

driveradmin passes the following arguments into the shared library:

- **int fd**  
  File descriptor of third party’s special device filename used by the shared library function to get the statistics.

- **int cur_ppa**  
  PPA of interface whose statistics are to be displayed.

- **int termlines**  
  Number of terminal lines in the current screen/window; typically used by a shared library function to determine whether the number of statistics being displayed is greater than the screen length.

Writing a driveradmin Shared Library

Two requirements must be met for any existing or new shared library function written specifically to display the interface statistics:

- The shared library function must be 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 Fibre Channel. In addition, most Hewlett-Packard shared libraries and most Hewlett-Packard LAN drivers maintain the link-specific MIB statistics. For example, Hewlett-Packard Ethernet/802.3 drivers maintain the Ethernet-like MIB statistics defined in RFC 1398.

Localizing Output Messages

All outputs from shared libraries should be localized. The shared libraries should use the Hewlett-Packard Native Language Support...

Example for Writing a Shared Library

The dsenet.c file provides an example of how to write a shared library. The actual getting and displaying of interface statistics is described in the sample shared library, dsenet(), which is the actual source code for libdsenet.sl, shown in Appendix D, “Shared Library Examples for the driveradmin and lanscan Commands.”

Shared Library Message Catalog

The shared library function, dsenet(), first opens its message catalog. This catalog file is accessed only by its shared library and could be named anything and put any place. To avoid confusion, however, you should conform to Hewlett-Packard conventions for the naming and placement of message catalog files. Each Hewlett-Packard shared library has its own message catalog file in:

/usr/lib/nls/C/dsdriver_name.cat

Getting the Interface Statistics

The dsenet() function uses the DL_GET_STATISTICS_REQ primitive to request interface statistics from the enet driver. The function expects to receive a DL_GET_STATISTICS_ACK primitive that contains the requested statistics. You can, alternatively, use the source of the dsenet() function as an example.

Displaying the Interface Statistics

If the driver maintains RFC 1213 MIB II statistics, the shared library can use the code in the dsenet() function that displays these statistics. If the driver also maintains interface specific statistics, the shared library should display a “Continue” message after displaying the RFC 1213 statistics and wait for a key to be pressed before displaying them. dsenet() functions in this manner.

lanscan Command

The lanscan command displays information about each of the LAN links on the system. lanscan can get access programmatically to all information to be displayed except the encapsulation method. To
determine the encapsulation method, lanscan must make a request to
the shared library. There is one shared library for each networking
driver that is to work with lanscan.

Displaying Encapsulation Methods

A lanscan shared library can display the encapsulation methods
supported by an interface. The shared libraries are invoked by lanscan
when the user selects the “-v” (verbose) option on the command line.
Since lanscan displays information about all LAN interfaces on the
system, a different shared library is invoked for each interface. lanscan
traverses the hw_ift linked list to find out what LAN interfaces are
configured and what information is to be displayed. See “hw_ift_t
Structure Description and Initialization” on page 184 for more
information on the hw_ift structure.

When “-v” is selected:

1. lanscan gets the driver name out of the name element of the hw_ift
   structure to find out the name of the shared library file to access.

2. lanscan does a shl_load() of the file with the following form:

   /usr/lib/lanscan/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 above is
   hardcoded in lanscan.

3. After loading in the shared library, lanscan again uses the driver
   name to find the name of the shared library function to use by
   executing shl_findsym() of a function name with the following form:

   pedriver_name

   where driver_name is as just described with pe standing for “print
   encapsulation.”

4. lanscan then uses the handle returned by shl_findsym() to invoke
   the shared library.

Argument Passed to the Shared Library

lanscan passes the following argument into the shared library:

hw_ift_t *hwift;   Pointer to the hw_ift structure for the interface whose
information is being displayed by lanscan.
Recommendations for the lanscan

Shared Library Function

The shared library function should start displaying the encapsulation methods at the point where the cursor currently is located. It should not output any spaces, tabs, or line feeds. The shared library function has columns 43 (column count starting from 0) through 80 with which to display all the supported encapsulation methods. The shared library function should not output any spaces, tabs, or line feeds after displaying the encapsulation methods.

Shared library outputs should always be localized. That is, the Native Language Support (NLS) message catalogs should be used. Refer to the HP Native Language Support: User’s Guide for further information.

Shared Library Message Catalog

The `peenet()` shared library function first opens its message catalog. Each shared library has its own message catalog file in:

```
/usr/lib/nls/C/pe<driver_name>.cat
```

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:

```c
/* LLC Encapsulation Types */
#define IEEE0x01/* IEEE 8022*/
#define HP_EXT_IEEE0x02/* HP Extended IEEE 8022*/
#define SNAP0x04/* IEEE SNAP*/
#define ETHERTYPE0x08/* Ethernet*/
#define NOVELL0x10/* Ethernet */
```
Creating Networking Device Drivers

Network Monitoring Commands: driveradmin, driverlinkloop and lanscan

**driverlinkloop Command**

The `driverlinkloop` command uses IEEE 802.2 link-level test frames (TEST path) to check connectivity within a local area network. `driverlinkloop` explicitly gets a PPA from the input command line or implicitly selects a PPA by getting the first element of the PPA list and attaches to this PPA via HP-DLPI as default. Unless users specify the third party's option in the command line to input a third party DLPI stream driver's filename, `driverlinkloop` will attach to the picked PPA via this third party's DLPI stream driver. After attachment, the `driverlinkloop` routine will use DL_TEST_REQ primitive to “ping” peer data link providers to test the data-transfer path.

Example:
```
enetlinkloop -l 1 -3 /dev/dlpi_enet 0x0060B07EAAFD
```
Network Tracing and Logging Support for Troubleshooting

This section describes and explains the use of HP-UX network tracing and logging facilities.

To aid in troubleshooting network problems, support for network troubleshooting must address several trends:

• The complexity of network systems is increasing.
• The number of protocols and standards is large and continues to grow.
• The number of possible combinations of services and applications created and used on a network is increasing.
• The troubleshooter is usually far removed from those who understand the network, products, and systems best.

Troubleshooters need knowledgeable support tools to address this complexity and difficulty. Support tools must provide as much information as possible about when and where problems occur. The network code must provide the troubleshooter with failure occurrence, cause, and suggested repair information.

HP-UX network tracing and logging facilities are tools for capturing network events and packets in a log for analysis to support troubleshooting. (Sometimes special diagnostic and test tools must also be used; for example, network traffic analyzers, interpretability tests, and other such aids.)

HP-UX network tracing and logging facilities permit subsystems to record events in a central location for subsequent processing. That information can then be provided to customers and support personnel to audit network activity and troubleshoot network problems.

Introductory Overview of HP-UX Tracing and Logging

HP-UX network tracing and logging facilities provide the following general features:

• A mechanism for recording log events and trace data
Creating Networking Device Drivers

Network Tracing and Logging Support for Troubleshooting

- A facility for determining what information to capture
- A mechanism for selecting and formatting the recorded information
- A set of user-interface commands that:
  - Configure, start, and stop the trace and log services.
  - Format captured messages.

These commands and the other HP-UX network tracing and logging facilities (files, subroutines, etc.) discussed in the following sections, provide a programmatic interface that allows user routines and kernel routines to access the services.

Figure 7-6, “Network Tracing and Logging Elements and Data Flow I,” shows the data flow among the following elements of the HP-UX trace and log system:

- nettlconf command
  - nettlgen.conf subsystem configuration database (for the following three commands)
  - nettl command
  - netfmt command

- Storage buffer in shared memory
- Subsystem Management Table in shared memory
- Storage buffer in kernel
- Subsystem Management Table in kernel
- ntl_reader daemon
- nktl_daemon daemon
These elements are explained in the following sections of this chapter.
Creating Networking Device Drivers

Network Tracing and Logging Support for Troubleshooting

**nettlgen.conf(4)**

The `nettlgen.conf` file stores subsystem records, particularly the unique subsystem ID. This subsystem information is used by the `nettl` and `netfmt` commands to identify and control subsystem tracing and logging behavior. Each subsystem must have a unique subsystem ID. The ID is used as identification for all interactions with the tracing and logging facility.

---

**NOTE**

You must obtain this subsystem ID from Hewlett-Packard (see “Assign Subsystem ID” on page 258.

---

**nettlconf(1M)**

The `nettlconf` command creates and updates the database file `/etc/nettlgen.conf`, the file used to configure each subsystem. This database file controls the behavior of the `nettl` and `netfmt` commands for tracing, logging, and formatting (trace/log) messages. See `nettlconf(1M)`, `nettl(1M)`, and `netfmt(1M)`.

Information such as the subsystem name, library name, and subformatter function are given to the `nettlconf` command, which stores them in the `/etc/nettlgen.conf` configuration file. This command is used in the configure script of the subsystem module during a system install/update time to integrate the subsystem into the trace and log tool. Subsystems use the `nettlconf` command to store a description of themselves in the `nettlgen.conf` database file—typically performed only once, at product installation time.

**nettl(1M)**

This command uses the subsystem information to create subsystem management tables in shared memory and in the kernel, and it starts, stops, and sets the capture criteria for tracing and logging. Specifically, `nettl` creates a port where messages can be stored while being written to the output file. `nettl` initializes the ktl driver, also called `netdiag1`, and `nettl` starts up the `nktl_daemon` and `ntl_reader` daemons. See the manpage for more detailed information.

**netfmt(1M)**

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

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

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

3. `netfmt` calls the functions of subsystems for which it has data.

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

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

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

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

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

The following guidelines may help developers remain customer-oriented in designing tracing and logging facilities for solving the problems of their troubleshooter clients:

- Log only what is needed to solve problems.
- Record all information to diagnose the problem in the log.
- Provide a hex dump to the troubleshooter only as the last resort.
- Make each product do as much self-diagnosis and repair as possible, and do it quietly. Notify the end user only when intervention is required or requested.
- Give the customer what is needed to solve the customer's problems,
not the developer’s problems.

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

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

**Assign Subsystem ID**

Each networking product requires its own unique subsystem ID number, which must be assigned by the Hewlett-Packard OpenConnect Team.

To do so, you must email a request for a unique subsystem ID for your product to Hewlett-Packard at nettl_support@india.hp.com. In the message, you should identify a suggested interface subsystem name for your product. You will be assigned this name if it is not already planned for or in use by Hewlett-Packard products. Check /usr/include/sys/subsys_id.h in your system prior to selecting a name. Do not request names such as lan, lo, ni, X25_, and others that are already assigned.

Your response from HP will include a unique subsystem ID number and a subsystem name in an up-to-date file of unique subsystem ID numbers and associated subsystem names.

This subsystem ID number is represented as the variable N in the rest of this chapter.

---

**NOTE**

You must use the file you receive from Hewlett-Packard as /usr/include/sys/subsys_id.h in your HP-UX device driver development system when you compile your networking device driver.

---

**Classify Trace Data**

Tracing can capture or make snapshots of loopback or header information, as well as inbound and outbound packets going through the network. The main purpose of tracing is to analyze networking problems...
discovered in either a log error message or in the failure of a networking
operation to complete successfully. Tracing follows or records normal
events and abnormal events alike and is typically used on events that
occur frequently, such as connections opening and closing, or
retransmitted data.

Trace kinds are defined as:

- **PDU**: Inbound and outbound Protocol Data Units (including
  header and data).
- **Header**: Inbound and outbound protocol headers.
- **Loopback**: Trace of packets emanating and returning to the same
  system.
- **Procedure**: Trace of entry and exit from all procedures.
- **Error**: Invalid state transitions, invalid protocol data units,
  bad headers, resource errors, system call errors, and
  protocol violations. Distinguishing when to use an error
  trace or an error log can be difficult. In some cases, you
  may want both. The tracing and logging utility goes to
different files, and locating and synchronizing the
entries between the two files may be too difficult.
Having both an error log and error trace helps to
synchronize the two files. Sometimes other log
messages are also recorded in the trace file when
tracing is enabled.
- **State**: Protocol states or connection states, not limited to
  entry and exit from a layer or procedure. Use this trace
  kind when recording information about normal state
  transitions.
- **Connection**: Information about connections as they are made and
  destroyed.
- **Logging Trace**: Special kind of trace that contains a log message. This
  trace kind will help the troubleshooter locate and
  synchronize logging and tracing output.

**Format Trace Data**

Troubleshooters should trace both incoming and outgoing data through
the stack. The trace records from different processes should be threaded
together to form a complete record of the path the PDU takes going from
the user application out the wire, and vice versa.

Refer to the following guidelines when implementing your tracing routines.

- Each process should trace incoming and outgoing data from both top and bottom. Alternatively, each protocol could trace only its incoming and outgoing headers.

- A subformatter for a process's trace information must be provided by the implementer of the process.

- The subformatter formats only the data for which that process is responsible. For example, if the X.25 driver sends a trace record, it decodes only the X.25 portion of the PDU, leaving the rest for the process above it to decode. Likewise, OTS decodes only the Network, Transport, and Session layer portions, leaving the upper layers to the application processes.

**Classify Log Data**

Logging is a way of capturing and recording specific network activities and infrequent significant network events, such as state changes, errors, and connection establishment. The main purpose of logging is to inform the system operator about these significant events and to make a permanent record for later interrogation. Typical log messages are about errors (catastrophic, recoverable and non-recoverable), warnings (major and minor), or system-wide information (such as changes to configuration or operation).

Logged events are considered in the following classes:

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disaster</td>
<td>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.</td>
</tr>
<tr>
<td>Error</td>
<td>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</td>
</tr>
</tbody>
</table>
was able to recover. For example, an error class condition occurs when a process must abort its operation or take extra steps to recover a certain state.

**Warning**
Indicates an abnormal event, but not necessarily a networking problem; event possibly caused by a subsystem problem. Examples include possible pointer alignment problems or data being accessed that has not been initialized.

**Informative**
Describes infrequent operations and current system activities, such as protocol module initiation and termination sufficiently important to post.

### What and When to Log

The most important part of logged messages is the ASCII string describing the event, which is the first item a system operator might see on the system console following an event in the network operation.

Deciding what to log and when it should be logged often involves trade-offs in terms of usability, performance, schedule constraints, and management and peer pressure. Besides the items outlined in the preceding tracing or logging sections, some general guidelines include:

- If an event results or causes the product or system to be unusable by all users, it should be logged as a Disaster class log message.
- If an event affects a single application, it should be logged as an Error class log message.
- If an event may cause an error or disaster in the future, or cause performance degradations, it should be logged as a Warning class log message.
- If an event occurs infrequently, and is something the user may want to know about but will not cause future problems, it should be logged as an Informative class log message.
- If an event occurs frequently, or with regularity, it is probably not appropriate to log it, but to trace it instead. Don't use Informative log messages in place of tracing.
- Don't log “Me Too!” messages in Error or Disaster class. These are events which occur in response to an error or disaster event in another place, but aren't themselves a disaster or error. “Me Too!” messages are characterized as providing no additional information to
solve the problem at hand.

• Do not acquire a new log instance if one is already available for the particular event thread you are on. (A log instance is a unique static number used to identify the thread of events attending an interface.)

• Include as much information as possible in log messages. The troubleshooter should be able to know what happened, what caused it, and how to proceed to fix the problem on the basis of your log message alone.

• State the exact commands to use to perform the recommended actions.

• If the explanation is too long to include in the log message, refer troubleshooters to the appropriate manual to take further steps or gain more knowledge about the problem.

• Encapsulate logging calls in functions or macros.

• Adhere to the logging error classes (Disaster, Error, Warning, and Informative) to promote uniformity in the troubleshooting process you recommend and to facilitate communication with HP support groups.

• Restrict logged information to only a few, well-defined types; for example, event number, a bounded array, or a string.

• Identify error recovery procedures for Disaster- and Error-class events.

• Devote most of your effort to understanding and documenting the procedures listed above. Only after completing error recovery procedures for these events should you focus on Informative and Warning class events, and then only if they would actually be useful.

**Passing Data to HP-UX Tracing/Logging**

Kernel subsystems that use the trace and log services must include the following in their source files and makefiles.

```c
#include <net_diag.h>   Contains macro calls to check that tracing and logging is enabled for the subsystem.
#include <subsys_id.h>   Contains subsystem information and definitions for log classes and trace kinds.
```

The function calls for kernel subsystems are provided for capturing trace
Creating Networking Device Drivers

Network Tracing and Logging Support for Troubleshooting

and log data.

KTRC_CK()

This routine is used to trace on an all interface-device basis.

This macro allows the calling process to verify if tracing is enabled for the current subsystem. The returned value is 1 (one) if tracing is enabled. It is defined as:

KTRC_CK(subsys_id, trace_kind)

subsys_id Unique subsystem ID of the calling subsystem. The number is assigned by Hewlett-Packard; see “Assign Subsystem ID” on page 258.

trace_kind Defines the kind of trace; available kinds are defined in the subsys_id.h header file and are described in more detail in “Classify Trace Data” on page 258, as follows:

- 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 above in the net_diag.h header file:

# 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 0 and is defined as:

```c
ktrc_write ( int subsys_id,
            int trace_kind,
            int path_id,
            int device_id,
            caddr_ttl_packet,
            int ttl_packet_cnt)
```

- **subsys_id**  
  Unique subsystem ID of the calling subsystem (number assigned by Hewlett-Packard; see the “Assign Subsystem ID” on page 258 section).

- **trace_kind**  
  Defines the kind of trace. All kinds are defined in the header file `subsys_id.h`. The following is the defined trace kind values (see “Classify Trace Data” on page 258). 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
Creating Networking Device Drivers

Network Tracing and Logging Support for Troubleshooting

LOGGING_TRACE_BIT
Log call tracing mask

LOOP_BACK_BIT
For loopback

PTOP_BIT
For point to point

path_id
Connection path on the host. If this is a nonapplicable parameter, pass in –1.

device_id
Device ID number (for example, if_unit) of the calling subsystem message. If this is a nonapplicable parameter, pass in –1.

tl_packet
Either a pointer to an mbuf chain or a pointer to a set of iovec structures as determined by tl_packet_cnt. The calling routine will pass a pointer (cast to caddr_t) to an mbuf chain or an iovec structure. This structure is immediately copied into an mbuf chain owned by tracing and logging facilities. Therefore, it is not necessary for the calling routine to copy the data and then pass a pointer to it.

tl_packet_cnt
If –1, then tl_packet points to an mbuf chain. If greater than 0, this is the number of the iovec structure to which tl_packet points.

As with logging, developers should encapsulate tracing calls in functions or macros. The code scenario in the following section shows a typical use of tracing calls.

**Tracing Code Sample**

The following example shows a trace of an outbound packet whose various parts are located in distinct memory locations. The trace uses the vectored data capability of the ktrc_write() call. The same could be accomplished using an mbuf chain as well.

```
1 #include "../h/netdiag1.h"
2 #include "../h/net_diag.h"
3 #include "../h/subsys_id.h"
4 #include "../h/uio.h"
5 ...
6
7 #define MAX_BUF 3 /* any number of vectors are allowed */
8 #define TRACE 0
```
Creating Networking Device Drivers

Network Tracing and Logging Support for Troubleshooting

```c
#define FALSE 0
...
int trace_pdu_out(pdu_hdr, pdu_hdr_len, pdu_data, pdu_data_len)
char *pdu_hdr;
int pdu_hdr_len;
char *pdu_data;
int pdu_data_len;
{
    int kind;
    int device_id;
    int path_id;
    short subsys_id;
    struct iovec tl_buf[MAX_BUF];
    int tl_buf_cnt;

    /* Set up variables for KTRC_CHECK() */
    subsys_id = MY_SUBSYS_ID_NUMBER; /* assigned by HP */
    kind = PDU_OUT_BIT;
    device_id = -1; /* -1 means not applicable */
    path_id = -1; /* -1 means not applicable */

    if (KTRC_CHECK(subsys_id, kind, device_id))
    {
        /*
         * Tracing is enabled for this subsystem
         * and kind combination.
         *
        */
        tl_buf[0].bufptr = pdu_hdr;
        tl_buf[1].bufptr = pdu_data;
        tl_buf[1].buflen = pdu_data_len;
        tl_buf[2].bufptr = NULL;
        tl_buf[2].buflen = 0;
        tl_buf_cnt = 2;
    }
    ktrc_write(subsys_id,
```
Creating Networking Device Drivers
Network Tracing and Logging Support for Troubleshooting

51     kind,
52     path_id,
53     device_id,
54     &tl_buf,
55     tl_buf_cnt);
56   }
57
58   return(0);
59 }

**KLOG_CK()**

This macro allows the calling process to find out whether logging is enabled for the current subsystem.

The returned value is 1 (one) if logging is enabled. It is defined as:

KLOG_CK(subsys_id, log_class)

subsys_id   Unique ID number (assigned by Hewlett-Packard) of the calling subsystem.

log_class   Defines the classification of event. All classes are defined in the header file subsys_id.h (see also “Classify Trace Data” on page 258). Four classes are defined for logging messages:

- Informative   Normal messages only
- Warning       Warning messages
- Error         Error condition messages
- Disaster      Critical error messages

**kget_log_instance()**

This call accepts no parameters but returns a unique log instance value. The log instance helps thread log messages together so that the user can easily identify the messages that result from the same event. A change in the log instance means that a new event is being logged.

The log instance value should be passed between subsystems through their interface parameter list so each module can have access to it. If a module encounters a unique event, it will obtain a log instance value. Otherwise, the module should use the current log instance value it was passed without calling kget_log_instance(). (See also klogg_write() for information on log instance values.)
Creating Networking Device Drivers

Network Tracing and Logging Support for Troubleshooting

**klogg_write()**

This routine is used to send log messages to the kernel trace and log facility.

Prefiltering is done at the time of the log call, and unwanted messages are dropped. This routine always returns a success of 0 and is defined as:

```c
klogg_write ( int subsys_id,
              int class,
              int device_id,
              int log_instance,
              caddr_t tl_packet,
              int tl_packet_cnt)
```

**subsys_id**  
Unique ID (number assigned by Hewlett-Packard) of the calling subsystem.

**class**  
Defines the classification of event. All classes are defined in the header file `subsys_id.h` (see also “Classify Trace Data” on page 258). Four classes are defined for logging messages:

- Informative: Normal messages only
- Warning: Warning messages
- Error: Error condition messages
- Disaster: Critical error messages

**device_id**  
Device ID number (for example, if_unit) of the calling subsystem message. If this is a nonapplicable parameter, pass in -1.

**log_instance**  
Unique static number used to identify the thread of events attending an interface. If this is a nonapplicable parameter, pass in -1.

**tl_packet**  
Either a pointer to an mbuf chain or a pointer to a set of `iovec` structures as determined by `tl_packet_cnt`. The calling routine passes a pointer (cast to `caddr_t`) to an mbuf chain or an `iovec` structure. This structure is immediately copied into an mbuf chain owned by tracing and logging facilities. So the calling routine need not copy the data and then pass a pointer to the data.

**tl_packet_cnt**  
If -1, then `tl_packet` points to an mbuf chain. If the
value is greater than 0, it is the number of `iovec`
structure (as defined in `uioc.h`) that `tl_packet` points
to.

Logging Code Sample

The following scenarios may help you to better understand how to use
the intrinsic calls of HP-UX logging facilities. These are typical
fragments of code that a subsystem might include to perform logging
calls.

```c
#include "../h/netdiag1.h"
#include "../h/net_diag.h"
#include "../h/subsys_id.h"

... 

#define MAX_BUF 1 /* any number of vectors are allowed */
#define LOG 1
#define FALSE 0
...

extern int log_instance;
extern unsigned short kget_log_instance;
...

int log_disaster()
{
    int class;
    int device_id;
    event_data_type event_data;
    short subsys_id;
    struct iovec tl_buf[MAX_BUF+1];
    int tl_buf_cnt;

    /*
     * Set up variables for call to KLOG_CK()
     */
```
subsys_id = MY_SUBSYS_ID_NUMBER; /* assigned by HP */
class = DISASTER;
device_id = -1; /* -1 means not applicable */

if (KLOG_CK(subsys_id, class)
{
	/*
	* Logging is enabled for this subsystem
	* and class combination.
	*/

	if (log_instance == 0)
{
		/*
	* There was no previous log instance
	* associated with this event. This is
	* the first module to encounter the
	* problem, so it gets the log instance.
	* Log instance should be available to
	* all modules in the subsystem and to
	* other subsystems.
	*/
		log_instance = kget_log_instance();
}

event_data.event_number = THIS_EVENT_NUMBER;
event_data.event_type = THIS_EVENT_TYPE;
/*
* Additional data about the event can be
* placed in the data structure. This
* data structure is entirely up to the
* local developer to design. The
* subformatter for this subsystem must
* be able to decode the data structure,
* but other than that there are no
* restrictions on what gets passed. The
* local developer may choose to use a
* single mbuf chain to hold all the
* event information, or pass a vectored
* buffer to the klogg_write() call to
* 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);
}

The following sections detail facilities and network device driver
developer responsibilities for formatting trace/log data output. Some
sections provide code and output examples. Appendix C, "How to Design
a Networking Trace/Log Subformatter and a Sample Subformatter,"
shows a generic style subformatter that can handle the preceding logging
and tracing examples using the basic calls of netfmt.

Formatting Networking Trace/Log Messages

The netfmt formatter is the facility that presents trace and log
information in human-readable form. It comprises two distinct pieces:

- Subformatter: a function provided by the subsystem to interpret the
  messages and produce human-readable form output.
- Formatter core: responsible for file handling, global filtering, and
dispatching messages to the appropriate subsystem subformatter.
Creating Networking Device Drivers

Network Tracing and Logging Support for Troubleshooting

This `netfmt` formatter is a filter that transforms the binary trace or log output data file into human-readable form. The party that puts the trace and log calls into the code must also provide a means of formatting/interpreting the data passed in those calls. Similarly essential is ensuring the loading of all potentially useful subformatter libraries.

`netfmt` uses the subsystems configuration information to identify the shared libraries (provided by the subsystems) which contain functions to parse subsystem filters and format subsystem data. `netfmt` dynamically loads all shared libraries and finds the functions each time it is executed. `netfmt` calls the functions of a subsystem only when it has data belonging to the subsystem.

The formatter filter handles or discards records based on data in each message header; such filtering is described in the netfmt(1M) manpage. (See also “netfmt(1M)” in “Introductory Overview of HP-UX Tracing and Logging” on page 253.)

The formatter and subformatters determine filtering and formatting options by processing an auxiliary file referred to as the options file. This options file filtering feature is available to any subsystem.

During filtering, the formatter checks the message to make sure it contains good information. If the formatter finds a corrupted message header, an unknown subsystem, a message that is too long to handle, and so forth, it prints an informative message, formats the message header, and discards the remainder of the data. It then continues with the rest of the file.

The formatter provides utility functions that subformatters can call to perform common tasks, such as formatting the message header in a standard fashion, dumping raw data, and outputting the formatted data. These functions are discussed in following sections, with recommendations for usage. Figure 7-6, “Network Tracing and Logging Elements and Data Flow I,” identifies important tracing/logging subroutines and shows their relationship to tracing/logging facilities shown in Figure 7-5, “Master Agent/Subagents Relationship.” You can add additional functions, if necessary.

Appendix E in this guide addresses what the formatted data could look like, and gives further information on this subject. To recap, the formatter provides the following features:

- Loads subformatter shared libraries
Creating Networking Device Drivers
Network Tracing and Logging Support for Troubleshooting

- Processes filtering and formatting options
- Handles binary input
- Handles global filtering
- Processes unknown or bogus data
- Dispatches data to correct subformatter
- Handles common subformatter tasks

How to design a Subformatter and related issues are included in Appendix E.
Alternative Means of Development Debugging

Besides HP-UX network tracing and logging, several alternative methods and facilities for troubleshooting support are available to a developer. These alternatives are mentioned briefly in this section.
The simplest troubleshooting tool is to open up a file and perform debugging writes into it. However, using this scheme does not allow troubleshooters to control the information recorded, and the file often remains unknown and unnoticed. Further, this scheme is difficult to implement for kernel drivers.

A more sophisticated scheme is to use the HP-UX logging facility, syslog (see syslog(3)), which can capture information from various processes. syslog is similar to the logging facility discussed in this section, but it has fewer features. The single routine call, syslog, creates and sends a log message to the syslog daemon, syslogd. A configuration file, /etc/syslog.conf, determines where the message is dispatched: to a file, to another node, or to a user's session. For subsystems with a light amount of logging (that is, using simple printf()s) and no tracing, syslog might be an adequate facility; see the manpage for more detail.

Similarly, STREAMS modules and drivers might use the strlog (see strlog(7)) interface provided by the STREAMS facility to capture information from multiple processes. This interface is similar to syslog, except that additional control over what is captured is provided. The strlog call creates and sends log messages that can be collected with either of the strace or strerr daemons (see strace(1M) and strerr(1M)). As with syslog, this interface does not allow localization of the logged information. See the manpages for more detail.
Creating Networking Device Drivers

Network Tracing and Logging Support for Troubleshooting