Chapter 7  Creating Networking Device Drivers
This chapter provides information for designing and writing PCI networking device drivers.

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

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

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

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

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

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

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

Application Layer
- TCP/IP Networking Commands and Applications
- Driver Networking Commands and Utilities

User Space  Kernel Space

Protocol Interface Layer
- STREAMS Head and Device Files

Network Protocol Layer
- TCP
- UDP
- ARP
- IP
- X25
- OS

Data Link Layer
- Data Link Provider Interface
- Network Device Drivers Interface

PCI Bus
- Ethernet Card
- Token Ring Card
- X.25 Card
- LAPB
- FDDI Card

Creating Networking Device Drivers
HP-UX Networking Interface Architecture
Data Link Interface Layer

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

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

Network Protocol Layer

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

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

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

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

Protocol Interface Layer, Device File, and STREAMS Head

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

Provide services from transport layer protocols and to applications.

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

- Sockets
- Streams
- Device files

**STREAMS Environment**

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

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

The following documents are recommended sources:

**Hewlett-Packard Company:**


**Other References:**

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

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

Figure 7-2   Steps to Develop a Networking Driver

STANDARD KNOWLEDGE BASE
1.
HP-UX Network Interface Architecture
Data Structures
Protection & Synchronization
Network Driver Installation
Protocol Configuration, Binding, and Demultiplexing
mblk and queue macros
DLPI Interface
STREAMS DLPI Network Driver

OPTIONS
2. Aux. Code?
Y
- Auxiliary Code
3. N/W Mgmt Spprt?
Y
- Network Mgt. Support
4. Log & Trace Spprt?
Y
- Logging & Tracing Support
5. Aux. Files?
Y
- Auxiliary Files

Driveradmin Support
Lanscan Support
SAM Support
Auxiliary Files

Driveradmin Support
Lanscan Support
SAM Support
Auxiliary Files

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

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

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

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

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

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

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

Overview of Networking Driver Structure

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

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

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

2. Auxiliary Code

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

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

3. Network Management Support

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

Select or go to the next option.

4. Logging and Tracing Support

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

Select or go to the next option.

5. Auxiliary Files

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

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

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

Select or go to the next option

7. driveradmin files

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

Select or go to the next option.

8. lanscan support

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

The driver is now complete.

**STREAMS Model Device Drivers**

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

---

**NOTE**

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

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

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

Each device driver may maintain its `hw_ift_t` and `hw_dlpi_t` structure as part of a larger structure, the driver control block `enet_if_t` (shown in Figure 7-3 on page 217). The driver control block provides information used in driving and controlling the interface hardware.

**hw_ift_t Structure Description and Initialization**

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

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

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

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

DRV_NO_FAST_PATH
This flag must be set if a driver does not support fast path as described in “Transmission of Message Blocks” on page 254.

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

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

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

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

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

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

next
Pointer to next hw_ift structure in list. This field is set by calling the hw_ift_attach() routine during device driver initialization. See “Initializing Networking Device Drivers” on page 224 in this chapter for detailed information.
The following example shows the initialization of the `hw_ift` structure. Initialization is generally done in the driver `init` routine:

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

hw_path_t hw_path;

hw_ift_ptr = &(enetift_ptr->hwift);

hw_ift_ptr->mac_type = DEV_ETHER;
hw_ift_ptr->llc_flags = IEEE | SNAP;
hw_ift_ptr->mjr_num = enet_drv_info.drv_info->c_major;
hw_ift_ptr->nm_id = get_nmid();
hw_ift_ptr->instance_num = wsio_isc_to_instance(isc_ptr, NULL);

hw_ift_ptr->mtu = ETHER_MTU;
hw_ift_ptr->name = "enet";
io_node_to_path(isc_ptr->card_node, NULL, &hw_path);
io_hw_path_to_str(hw_ift_ptr->hdw_path, NULL, &hw_path);

hw_ift_ptr->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; it should be initialized to zero. Further discussion of structure fields is not provided.
Protection and Synchronization for Driver and Transport

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

NOTE

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

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

hw_ift Structure Protection

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

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

```
HW_IFT_LOCK(hw_ift_ptr)
Acquire a spinlock on hwift_lock.

hw_ift_ptr: pointer to an hw_ift structure.
```
Creating Networking Device Drivers

Protection and Synchronization for Driver and Transport

**HW_IFT_UNLOCK(hw_ift_ptr)**

Release previously acquired hwift_lock spinlock.

*hw_ift_ptr*: pointer to an 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**

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

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

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

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

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

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

```c
drv_ops_t enet_drv_ops;
drv_info_t enet_drv_info= {
    "enet",         /* driver name */
    "pseudo"        /* driver 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, "enet", 0, 65536, 65536, 1
};
static struct module_info enet_wminfo= {
    5050, "enet", 0, 65536, 1, 1
};
static struct qinit enet_rinit= {
```
In addition to a `driver_install` (WSIO_DRV) routine, each HP-UX PCI networking device driver must have a `driver_attach` (WSIO_DRV) routine.

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

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

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

```c
int driver_install()
```

An entry of `driver_install()` is called during the I/O system configuration process. When the `driver_install()` routine is called, it hooks the `driver_attach()` entry to the top of a linked list of attach routines for all of the interface drivers in the system.
#ifdef __LP64__
int driver_attach(uint32_t product_id,
                struct isc_table_type *isc_ptr)
#else
int 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 \texttt{isc\_table\_type} structure.

void driver_init(struct isc_table_type *isc_ptr)

cb_ptr Pointer to the driver control block; it is driver developer
defined and passed as a pointer through the isrlink() routine during the \texttt{driver\_attach()} or
\texttt{driver\_init()} routines.

The \texttt{driver\_attach()} and \texttt{driver\_install()} initialization procedures
are common to all HP-UX device drivers. More details of each step are
presented in Chapter 5, \textit{Writing a Driver}.

### Calling \texttt{driver\_install()}

When the HP-UX system is configured through the \texttt{config} command, a
table of \texttt{driver\_install()} entry points is created from information in
/stand/system.

When \texttt{driver\_install()} is called by the I/O system configuration
process through the \texttt{driver\_install()} entry point configured in the
system, the \texttt{driver\_install()} routine places the \texttt{driver\_attach()} entry in a table of drivers to be called at configuration time. The
\texttt{driver\_install()} routine calls the \texttt{wsio\_install\_driver()} routine to
register the driver with the I/O subsystem and returns any error.

The following is a call to \texttt{driver\_install()}:

```c
static drv_ops_t enet_drv_ops = {
    NULL,       /* open */
    NULL,       /* close */
    NULL,       /* strategy */
    NULL,       /* dump */
    NULL,       /* psize */
    NULL,       /* reserved */
    NULL,       /* read */
    NULL,       /* write */
};
```
Chapter 7

Creating Networking Device Drivers

Initializing Networking Device Drivers

```c
NULL, /* ioctl */
NULL, /* select */
NULL, /* option1 */
NULL, /* reserved1 */
NULL, /* reserved2 */
NULL, /* reserved3 */
NULL, /* link */
0, /* device flags */

};

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

};

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

};

static wsio_drv_info_t enet_wsio_info = {
&enet_drv_info, /* driver info */
&enet_drv_ops, /* driver ops */
&enet_data, /* driver data */
WSIO_DRV_CURRENT_VERSION

};

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

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

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

NOTE

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

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

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

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

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

/*
* Obtain memory for Transmit and Receive Descriptor
* Rings and any additional driver data structures */
...

/*
* Get/Set PCI configuration
*/
pci_read_cfg_uint32_isc(isc,SSID,&ssid);
enet_iftp->sub_id = (ubit16)(ssid >> 16) ;
enet_iftp->sub_vendor_id = (ubit16)(ssid & 0x0000ffff) ;

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

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

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

/* Read the Configuration Driver Area information */
pci_read_cfg_uint32_isc(isc,CFDA,&cfda);

... Turn on PCI memory access and bus master capability
* on host */
pci_write_cfg_uint8_isc(isc, CFCS,
CPCS_MEMORY_SPACE_ACCESS |
CPCS_MASTER_OPERATION |
CPCS_PARITY_ERROR_RESPONSE |
CPCS_SYSTEM_ERROR_ENABLE |
CPCS_I_O_SPACE_ACCESS);

...
Creating Networking Device Drivers

Chapter 7 231

Initializing Networking Device Drivers

*/

/ *
* Perform general enet_ift initialization
* /

...  

/* Setup hwift structure */

...  

...  

/* Attach hwift to global list */

hw_ift_attach(&enet_iftp->lancift.hwift);

...  

...  

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

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

/* Allocate shared memory for Tx-descriptor ring */

if( wsio_allocate_shared_mem(isc,  
enet_iftp->tdr_DMA_handle,size,  
    (caddr_t *)& enet_iftp->tdr, 0) != WSIO_MAP_OK) {
    msg_printf("enet - TDR allocation failed...
    return -1;
}

...  

}  

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

Calls the `hw_ift_attach()` routine to link the `hw_ift` structure to a global list of `hw_ift` structures of active interfaces. The `hw_ift_attach()` routine is defined as:

```c
hw_ift_attach(hw_ift_t * hw_ift_ptr)
```

- `hw_ift_ptr`: pointer to the password `hw_ift` structure.
Protocol Binding and Demultiplexing

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

To correctly demultiplex inbound packets, a networking driver must:

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

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

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

Protocol kind is the type of protocol to bind. Interpretation of the protocol value field depends on the protocol kind value. “Protocol Kinds and Values” in Appendix A lists the names of the available kinds of protocols to bind and some of the available protocol values.
When the networking driver binds a protocol with protocol kind and type values, the driver knows what kind of packets to handle for that bind. The networking driver processes inbound packets on the interrupt control stack (ICS) for all the protocol binds by calling an associated STREAMS queue. To do this, it calls `putnext()` (see the `STREAMS/UX for the HP 9000 Reference Manual`) in the device driver's interrupt service routine. The driver must use the protocol ID that was carried in the `dl_sap` field of the `DL_BIND_REQ` to pass the packet to the right Stream that is logged (see “DLPI Interface to Upper Layers” on page 239 for details).

**Protocol Binding and Unbinding**

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

**Protocol Demultiplexing**

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

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

2. Locate the proper inbound packet service routine or queue for each valid incoming packet.
Distinguishing Packet Protocol Format

The following information can be used to determine the protocol format and type. To determine whether the packet is an Ethernet type packet:

- If the value of the TYPE field of an inbound packet is equal to or greater than 0x600, the packet is an Ethernet type packet. The protocol kind of the packet is `LAN_TYPE`, and the protocol value is the TYPE field specified in the packet (see “Protocol Kinds and Values” in Appendix A).
- If the value of the TYPE field is less than 0x600, the packet could be an IEEE 802.2 LLC type packet, SNAP or non-SNAP type.

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

- The packet is considered to be a SNAP packet (defined in IEEE 802.1a) if both the DSAP and the SSAP values are 0xAA. The protocol kind of the packet is `LAN_SNAP`, the protocol value is 0xAA, and the protocol value extended is the five-byte SNAP protocol data specified in the SNAP header (see “Protocol Kinds and Values”, in Appendix A).
- Otherwise, it is an IEEE 802.2 LLC non-SNAP type packet. The protocol kind is `LAN_SAP` and the protocol value is the DSAP field that is specified in the packet (see “Protocol Kinds and Values”, in Appendix A).

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

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

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

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

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

Table 7-2 explains each promiscuous mode.

<table>
<thead>
<tr>
<th></th>
<th>PROMISC_PHY</th>
<th>PROMISC_MULTI</th>
<th>PROMISC_SAP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unbound</strong></td>
<td>The stream gets all outbound packets transmitted on the interface. (broadcast, multicast, self addressed and non self addressed unicast packets)</td>
<td>The stream gets all outbound multicast, broadcast packets transmitted on the interface. No outbound unicast packets will be seen</td>
<td>The stream gets all outbound packets when the “source” SAP matches one of the protocols enabled on the interface.</td>
</tr>
<tr>
<td><strong>promiscuous</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>stream monitors</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>outbound</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>traffic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Unbound</strong></td>
<td>The stream gets all packets on the wire regardless of SAP or address.</td>
<td>The promiscuous stream gets all multicast, broadcast packets on the wire regardless of SAP or SNAP. No unicast packets will be seen on an inbound traffic.</td>
<td>The promiscuous stream gets all packets which pass the physical level filtering (local MAC, broadcast, or multicast addresses) for the interface and passes the protocol filtering (SAP type or SNAP enabled on that interface).</td>
</tr>
<tr>
<td><strong>promiscuous</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>stream monitors</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>inbound</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>traffic</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Inbound Promiscuous and Outbound Promiscuous

Table 7-2 Promiscuous Mode Matrix (Continued)

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

Message Block and Queue Functions and Macros

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

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

The list of commonly used message block functions.

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

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

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

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

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

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

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

Device file, Interface name and PPA number

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

```
# lsdev
 
	...............................................
	...............................................
239          -1         enet            lan

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

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

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

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

Table 7-3  
**lanscan Output**

<table>
<thead>
<tr>
<th>H/W Path</th>
<th>Station Address</th>
<th>Card Im. #</th>
<th>H/W State</th>
<th>Net I/F, Name</th>
<th>PPA</th>
<th>MAC Type</th>
<th>HP- DL PT</th>
<th>DLPI Mjr#</th>
</tr>
</thead>
<tbody>
<tr>
<td>8/16/6</td>
<td>0x0060B0 7EDBF0</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>0x0060B0 7A221E</td>
<td>1</td>
<td>UP</td>
<td>enet1</td>
<td>2</td>
<td>ETHER</td>
<td>No</td>
<td>*</td>
</tr>
<tr>
<td>8/0/2/0</td>
<td>0x0060B0 B2D850</td>
<td>2</td>
<td>UP</td>
<td>enet2</td>
<td>3</td>
<td>ETHER</td>
<td>No</td>
<td>*</td>
</tr>
</tbody>
</table>

**IP and ARP Configuration**

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

1. `ifconfig` opens device file `/dev/enet` and senses PPA configured is 1
2. `ifconfig` issues an `ioctl` to push IP module to top of `enet` driver
3. `ifconfig` issues another `ioctl` to issue attach and bind requests for PPA 1
4. `ifconfig` opens device file `/dev/enet` and issues `ioctl` to push ARP to top of `enet` driver
5. `ifconfig` again performs step 3 for ARP/enet stream.

6. `ifconfig` opens `/dev/ip` and uses it as dummy multiplexer; IP/enet and ARP/enet streams are linked under dummy multiplexer.
STREAMS DLPI and Network Driver Overview

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

putmsg()          Application Layer
                 DLS User
getmsg()          ____________________________

User Space

Kernel Space

Network Protocol Layer

STREAMS Head
/dev/enet and /dev/enetX

Data Link Layer

PCI BUS

/driver_open/
driver_close()  
driver_install() 
  driver_attach()  
    driver_init()  

_driver_wput()  _driver_wsrv()  _control()  
INFO, ATTACH,  
BIND, PPA_REQ

_fast_in()  _control()  _fast_out()  
_unitdata_out()  _intr()  

_driver_isr()  _build_hdr()  _proc_ioctl()  
_unitdata_out()  

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

**Device Driver/DLPI Driver Synchronization**

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

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

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

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

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

The following rules summarize actions each networking device driver must take in dealing with such DLPI ioctl requests:
1. The control request does one of the following:

- If the control request completes immediately with no error, the enet_dlpi_process_ioctl() routine immediately returns zero to DLPI.

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

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

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

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

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

**STREAMS Synchronization**

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

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

Driver Support for DLPI

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

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

Major Data Structures

NOTE

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

`enet_dli_data_t` This data structure contains STREAMS DLPI driver information for a Stream that is open currently with the driver.
typedef struct _enet_dlpi{
    enet_if_t *enetiftp;
    cred_t *cred;
    queue_t *queue_ptr;
    dev_tenet_dev;
    uint32_t dlsap_addr_length;
    uint8_t dlsap_addr[MAX_DLSAP_LEN];
    uint16_t service_mode;
    int curr_state;
    uint32_t xidtest_flag;
    int mac_type;
    int mac_mtu;
    dlsap_t *dlsap_ptr;
    uint8_t ssap;
    uint16_t sxsap;
    enet_mcast_list_t *enet_mcast_list;
    int promiscuous_flg;
    int promisc_filter;
    uint32_t noloopback_flg;
    uint32_t no_src_routing;
    uint32_t arp_stream;
    uint32_t ip_stream;
    int fast_path;
    int fast_path_pkt_type;
    int fast_path_llc_length;
    int pre_state;
} enet_dlpi_data_t;

The following table explains the fields.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>enetiftp</td>
<td>The interface that is associated with this open stream</td>
</tr>
<tr>
<td>cred</td>
<td>Credential structure of the user who opened this stream</td>
</tr>
<tr>
<td>queue_ptr</td>
<td>Queue pointer to the read queue of the stream</td>
</tr>
<tr>
<td>enet_dev</td>
<td>enet device number</td>
</tr>
<tr>
<td>dlsap_addr_length</td>
<td>Length of DLSAP address</td>
</tr>
</tbody>
</table>
### enet_dpni_data_t Data Fields (Continued)

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

Opening and Closing a Driver

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

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

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

The actual major number of the enet driver is 239.

# lsdev
..............
..............
239 -l enet lan

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>fast_path</td>
<td>Set if application requests to set up fast path</td>
</tr>
<tr>
<td>fast_path_pkt_type</td>
<td>The fast path packet type</td>
</tr>
<tr>
<td>fast_path_llc_length</td>
<td>The LLC header length used in the fast path</td>
</tr>
<tr>
<td>pre_state</td>
<td>Retains the state before a pending ioctl or control request with the driver; when the request is complete the streams can be set to the correct state</td>
</tr>
</tbody>
</table>
However, a clone device file for the `enet` driver is created as follows:

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

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

**Control Functions**

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

<table>
<thead>
<tr>
<th>Function Name [prefixed by enet_dlpi]</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>_attach()</td>
<td>The information for PPA to be attached is found from hw_if_t list; dlpi_ioctl() is issued to the driver with primitive DL_HP_HDW_INIT. The enet_dlpi_data_t for this stream is updated with network interface information and the stream DLPI state.</td>
</tr>
<tr>
<td>_bind()</td>
<td>DL_BIND_REQ primitive request indicates to bind a DLSAP to the stream. Protocol kind (LAN_TYPE, LAN_SNAP or LAN_SAP) is determined by SAP value in the request. The enet_log_sap_value() function is called. Once driver bind is successful, dlsap_t is allocated and initialized with protocol type and value of SAP. The enet_dlpi_data_t structure for this stream is updated with these bind details.</td>
</tr>
<tr>
<td>_control()</td>
<td>The primitives serviced by this function are - DL_ENABMULTI_REQ, DL_DISABMULTI_REQ, DL_SET_PHYS_ADDR_REQ, DL_PROMISCON_REQ, DL_PROMISCOFF_REQ and DL_HP_HW_RESET_REQ. The respective ioctl commands are issued to the driver via enet_dlpi_control(). If the request didn’t complete immediately, this routine sleeps on the address of the sleep object of the dlpi_ioctl().</td>
</tr>
<tr>
<td>_detach()</td>
<td>Disable all multicasts that were enabled through this stream by issuing dlpi_ioctl()s to the network driver. If promiscuous mode was enabled by this stream, disable it. The clean_str_spu_sw_q() routine is called to clean up any requests in the STREAMS/UX. Finally, update the state in enet_dlpi_data to DL_UNATTACHED.</td>
</tr>
</tbody>
</table>
Creating Networking Device Drivers
STREAMS DLPI and Network Driver Overview

Table 7-5: Message Service Functions (Continued)

<table>
<thead>
<tr>
<th><strong>Function Name [prefixed by enet_dlpi]</strong></th>
<th><strong>Functionality</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>_get_mib_req()</td>
<td>Services MC_GET_MIB_REQ (sys/mci.h). The driver ioctl DL_GET_STATISTICS is issued to get current MIB statistics.</td>
</tr>
<tr>
<td>_get_mibstats()</td>
<td>Calls enet_hw_req() function to get the standard MIB statistics from the driver structures.</td>
</tr>
<tr>
<td>_getphyaddr()</td>
<td>The enet_hw_req() function is called, which selects the permanent ROM physical address of the network interface, to service DL_PHYS_ADDR_REQ.</td>
</tr>
<tr>
<td>_info()</td>
<td>A service function for DL_INFO_REQ. The information is returned upstream in structure dl_info_ack_t. If the PPA is not attached yet, mac_type and mtu is set to DL_CSMACD and IEEE8023_MTU.</td>
</tr>
<tr>
<td>_multicast_list()</td>
<td>This function is called to service the DL_HP_MULTICAST_LIST_REQ primitive. In turn, this function calls driver dlpi_ioctl() to get the list by passing the command DL_HP_GET_MIB_STATS.</td>
</tr>
<tr>
<td>_ppa_req()</td>
<td>Receipt of DL_HP_PPA_REQ results in this function being called. The hw_if_t list is searched for this PPA and the information from hw_if_t is returned.</td>
</tr>
<tr>
<td>_set_mib_req()</td>
<td>This function services MC_SET_MIB_REQ. The driver ioctl DL_HP_RESET_STATS is issued to reset the MIB statistics.</td>
</tr>
<tr>
<td>_status()</td>
<td>This function sends the hw_if-&gt;hdw_state upstream in response to the DL_HP_HW_STATUS_REQ request.</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.

---

Table 7-5 Message Service Functions (Continued)

<table>
<thead>
<tr>
<th>Function Name [prefixed by enet_dlpi]</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>_subs_bind()</td>
<td>When DL_SUBS_BIND_REQ is received, this function is called. If the dl_subs_bind_class is DL_PEER_BIND, a new dlsap_t is allocated and initialized with protocol type and value of SAP. With DL_HEIRARCHICAL_BIND the dlsap_addr information in enet_dlpi_data_t is updated with bind details.</td>
</tr>
<tr>
<td>_subs_unbind()</td>
<td>For each dlsap_t bound, compare the unbind request SAP. If there is a match, routine enet_unlog_sap_value() is called.</td>
</tr>
<tr>
<td>_unbind()</td>
<td>The function enet_unlog_sap_value() is called. dlsap_t is deallocated and the bind information in enet_dlpi_data_t is set to the default value.</td>
</tr>
<tr>
<td>_xidtest_out()</td>
<td>This function services DL_TEST_REQ, DL_TEST_RES, DL_XID_REQ and DL_XID_RES. It builds the test/xid packet and sends it to the driver using dlpi_output().</td>
</tr>
</tbody>
</table>
The application sends DLPI an \texttt{M_IOCTL} message with the \texttt{ioctl} command \texttt{DLPI_IOC_HDR_INFO}. The \texttt{M_IOCTL} message block is linked with the \texttt{M_PROTO} message block with the \texttt{DL_UNITDATA_REQ} primitive. The LLC header format is built for the specific interface in a new \texttt{M_DATA} message block and linked to \texttt{M_PROTO}; the whole complex message is sent back to the application.

The \texttt{ioctl} \texttt{DLPI_IOC_DRIVER_OPTIONS} routine is processed by sending \texttt{hw_if_t} information for the request stream.

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

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

### Transmission of Message Blocks

The message block transmission has two paths in the sample implementation. The regular data path uses the \texttt{DL_UNITDATA_REQ} primitive and the fast path. The regular path is defined in the DLPI standards. The fast path uses \texttt{DLPI_IOC_HDR_INFO} \texttt{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 \texttt{enet_wput()} receives the \texttt{DL_UNITDATA_REQ} primitive request from the application to send a message to a destination specified in the unitdata message. The \texttt{enet_wput()} function calls the \texttt{enet_dlpi_unitdata_out()} function to service the request. The \texttt{enet_dlpi_unitdata_out()} function applies sanity checks for the stream’s DLPI state and request parameters and builds the LLC header. The LLC header message block is linked with the \texttt{first M_DATA} (with \texttt{DL_UNITDATA_REQ}) and calls the driver’s output routine \texttt{enet_hw_req()}.  
Fast Path  For better performance, fast path is used to transmit and receive data. The DLPI user sends DLPI ioctl DLPI_IOC_HDR_INFO to set up the fast path on the stream. The DLPI builds an LLC header template and sends it back to the user. For an outbound packet, the user prepends the link header to the data, based on the template of the link header, and sends M_DATA messages to DLPI. DLPI passes this packet to the network driver without building the link header. For an inbound packet on the fastpath stream, DLPI strips off the LLC header and passes it to the user without building and prepending the DL_UNITDATA_IND primitive to the data.

Reception of Message Blocks

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

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

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

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

The function implemented in enet_dlpi_fast_in() was discussed in “Fast Path” earlier in this section.
**Summary of DLPI Primitives and IOCTLs**

The following table summarizes the DLPI primitives and IOCTLs that have been dealt with in the sample drivers, along with appropriate comments. The processing of most DLPI primitives and IOCTLs involves driver interaction, which is discussed in “Driving the NIC” on page 257.

<table>
<thead>
<tr>
<th>DLPI Primitive or IOCTL</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DLPI PRIMITIVES DEFINED BY DLPI 2.0</strong></td>
<td>Information reporting</td>
</tr>
<tr>
<td>DL_ERROR_ACK</td>
<td>Attach</td>
</tr>
<tr>
<td>DL_INFO_REQ</td>
<td>Bind</td>
</tr>
<tr>
<td>DL_INFO_ACK</td>
<td>Other</td>
</tr>
<tr>
<td>DL_ATTACH_REQ</td>
<td>DLPI Ver 2.0 Connection less</td>
</tr>
<tr>
<td>DL_DETACH_REQ</td>
<td>Data transfer</td>
</tr>
<tr>
<td>DL_ERROR_ACK</td>
<td>DL_UNITDATA_IND</td>
</tr>
<tr>
<td>DL_OK_ACK</td>
<td>DL_UNITDATA_REQ</td>
</tr>
<tr>
<td>DL_BIND_REQ</td>
<td>DL_BIND_ACK</td>
</tr>
<tr>
<td>DL_BIND_ACK</td>
<td>DL_ERROR_ACK</td>
</tr>
<tr>
<td>DL_OK_ACK</td>
<td>DL_OK_ACK</td>
</tr>
<tr>
<td>DL_SUBS_BIND_REQ</td>
<td>DL_SUBS_BIND_ACK</td>
</tr>
<tr>
<td>DL_SUBS_BIND_ACK</td>
<td>DL_SUBS_UNBIND_REQ</td>
</tr>
<tr>
<td>DL_SUBS_UNBIND_REQ</td>
<td>DL_OK_ACK</td>
</tr>
<tr>
<td>DL_DISABMULTI_REQ</td>
<td>DL_PROMISCOFF_REQ</td>
</tr>
<tr>
<td>DL_ENABMULTI_REQ</td>
<td>DL_PROMISCON_REQ</td>
</tr>
<tr>
<td>DL_ERROR_ACK</td>
<td>DL_UNITDATA_IND</td>
</tr>
<tr>
<td>DL_OK_ACK</td>
<td>DL_UNITDATA_REQ</td>
</tr>
</tbody>
</table>
Driving the NIC

This section briefly explains the code flow of the lower part of the driver. This portion of the driver handles device interrupts, sends and receives frames, handles control requests from the upper part that require interaction with the device, and so forth. The objective here is to present the code flow of the sample driver `enet` as a background to the sample driver code.

Table 7-6 DLPI Primitives and IOCTLs (Continued)

<table>
<thead>
<tr>
<th>DLPI Primitive or IOCTL</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>HP EXTENDED DLPI PRIMITIVES</td>
<td></td>
</tr>
<tr>
<td>(These are HP extensions to DLPI 2.0 and may change. They are defined in sys/dlpi_ext.h)</td>
<td></td>
</tr>
<tr>
<td>DL_HP_HW_RESET_REQ</td>
<td>Hardware reset. Used by enetadmin.</td>
</tr>
<tr>
<td>DL_HP_HW_STATUS_REQ</td>
<td>Get hardware status req</td>
</tr>
<tr>
<td>DL_HP_MULTICAST_LIST_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_HP_PPA_REQ</td>
<td>Used by commands, enetadmin, enetlikloop, etc.</td>
</tr>
<tr>
<td>DL_HP_RESET_STATS_REQ</td>
<td>Reset statistics. Used by enetadmin.</td>
</tr>
<tr>
<td>(These are HP specific IOCTLs and may change. They are defined in sys/mci.h)</td>
<td></td>
</tr>
<tr>
<td>MC_GET_MIB_REQ</td>
<td>Get MIB statistics</td>
</tr>
<tr>
<td>MC_SET_MIB_REQ</td>
<td>Set MIB statistics</td>
</tr>
<tr>
<td>HP IOCTLs</td>
<td></td>
</tr>
<tr>
<td>DLPI_IOC_DRIVER_OPTIONS</td>
<td>To get driver features.</td>
</tr>
<tr>
<td>DLPI_IOC_HDR_INFO</td>
<td>To get LLC header for fast path.</td>
</tr>
<tr>
<td>DLPI_SET_NOLOOPBACK</td>
<td>Do not loopback the message.</td>
</tr>
</tbody>
</table>
Data Structures

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

typedef struct enet_if_t {
    enlan_if      lancift;
    /***********************************************************/
    * PCI Configuration information - PCI CONF
    *******************************************************
    ...
    ...
    /***********************************************************/
    * PCI Control and Status registers. Each field contains the
    * HPA + offset for the network contlr. registers - DEV REG
    *******************************************************
    ...
    ...
    /***********************************************************/
    * Device Specific Section - DEV SPEC
    *******************************************************
    struct isc_table_type *isc;
    enet_srom_t *srom;/* Serial ROM layout*/
    ubit32 drv_state;/* Driver state info.*/
    ubit32 reset_state;/* Driver reset state*/
    ...
    ...
    /***********************************************************/
    * Transmit Section - TX SECT
    *******************************************************
    enet_tb_t*tbr; /* Transmit buffer Ring */
    enet_td_t*tdr;/* Transmit Descriptor Ring */
    void *tdr_DMA_handle; /* DMA handle for Tx-desc ring */
    ...
    ...
    /***********************************************************/
    * Receive Section - RX SECT
    *******************************************************
    enet_rd_t*rdr; /* Receive Descriptor Ring */
    enet_rb_t*rbr;/* Receive buffer Ring */
    void *rdr_DMA_handle; /* DMA handle for Rx-desc ring */
STREAMS DLPI and Network Driver Overview

 Chapter 7

Table 7-7  enet_ift Data Fields

<table>
<thead>
<tr>
<th>Field Name/Generic Description</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>lancift</td>
<td>Contains generic LAN information</td>
</tr>
<tr>
<td>PCI INFO</td>
<td>Has PCI configuration information</td>
</tr>
</tbody>
</table>
### enet_ift Data Fields (Continued)

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

```c
typedef struct{
  hw_ift_t hwift;
  lan_timer lantimer;
  int ptr_t (*hw_req)();
  int (*dma_time)();

  /* Status and statistics Data Area - STATUS & STAT*/
  uint32_t BAD_CONTROL;
  uint32_t UNKNOWN_PROTO;
  uint32_t RXD_XID;
  uint32_t RXD_TEST;
  uint32_t RXD_SPECIAL_DROPPED;
  short int is_scaninterval;

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

**Table 7-8 enlan_ift Data Fields**

<table>
<thead>
<tr>
<th>Field Name/Generic Description</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>hwift</td>
<td>Generic Hardware information</td>
</tr>
<tr>
<td>lantimer</td>
<td>DMA/Control timer to track if a DMA or control operation is taking too long</td>
</tr>
<tr>
<td>hw_req()</td>
<td>h/w interface request function pointer</td>
</tr>
<tr>
<td>dma_time()</td>
<td>DMA timeout error handling</td>
</tr>
<tr>
<td>STATUS &amp; STAT</td>
<td>More statistics</td>
</tr>
<tr>
<td>num_multicast_addr</td>
<td>Number of multicast addresses active</td>
</tr>
</tbody>
</table>
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];
    caddr_t ift_ptr;
    queue_t *q_ptr;
    int flags;
    
};
```

**Table 7-9  Bound SAP Data Fields**

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>protocol_val</td>
<td>SAP, Type, or Canonical value</td>
</tr>
<tr>
<td>ift_ptr</td>
<td>Driver control block (enet_ift)</td>
</tr>
<tr>
<td>q_ptr</td>
<td>Queue pointer of the stream which did the bind</td>
</tr>
<tr>
<td>flags</td>
<td>LANC_ONICS and LANC_STRIP_HEADER bits</td>
</tr>
</tbody>
</table>
The following structure is used to link the logged_infos.

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

### Control Functions

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

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

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

```c
typedef struct {
    mcast_list_t *hw_mcast;
    int mc_threshold;
    /* Threshold for mcast addresses */
} hw_mcast_entry_t;
```

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

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

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

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

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

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

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

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

enet_media_control() function calls the ENET_DEL_EXT_MCAST macro.

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

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

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

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

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

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

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

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

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

**Figure 7-5**  Control Flowchart for Outbound Path

- **enet_wput**
  - Regular data path
  - Fast Path
    - `enet_hw_req`
      - Multicast, Broadcast, loopback packets and packets with more than 7 mblks or page crossings
        - `enet_slow_hw_req`
          - `enet_unit_data_out`
            - `enet_dlpi_build_hdr`
              - `enet_process_packet`

- **ENET_TRANSMIT_FRAME**
  - Outbound promiscuous
    - `enet_filter_pkts`
      - Calls `enet_ether_ics` or `enet_802_2_ics` depending on packet type and mblk
        - finally sent to read queue

- **ENET_MAP_DATA**
  - `enet_fill_td`
    - `OPROMISC_CLONE_LOOPBACK_PKT`
      - Issues Transmit poll request to the card

- **ENET_SETUP_TRANSMIT_DESCRIPTOR**

- Issues Transmit poll request to the card
enet_dlpi_unitdata_out()  This function calls enet_hw_req() to handle the write request.

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

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

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

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

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

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

Inbound Path

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

For PROMISC_PHYS, all packets are passed to the stream.

**Interrupt Service Routine - enet_isr()**

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

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

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

**Releasing any Pending Timeouts**

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

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

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

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

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

DMA Mapping

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

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

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

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

V Class

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

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

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

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

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

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

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

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

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

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

...
Network Management Support

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

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

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

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

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

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

- **u_int32 get_nmid()**  
  Allocates a system unique ifIndex value for use in the MIB-II ifEntry object. Any kernel entity that required an entry in the ifTable should use this service for retrieving the value of the ifIndex field.

  > 0 indicates the call succeeded and the value returned is the ifIndex value.

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

Example code in enet driver:

```c
enet_iftp->lancift_hwift.nm_id = get_nmid();
```

- **u_int32 return_nmid (u_int32 ifIndex)**  
  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.

```c
u_int32 return_nmid (u_int32 ifIndex)
```
ifIndex ifIndex value to be returned to the pool of available ifIndex values.

<0 Indicates the ifIndex value being returned was not the previously assigned ifIndex value

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

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

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

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

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

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

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

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

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

Introductory Overview of HP-UX Tracing and Logging

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

- A mechanism for recording log events and trace data
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 and kernel routines to access the services.

Figure 7-8, Network Tracing, Logging Elements and Data Flow I, shows the data flow among the following elements of the HP-UX trace and log system:

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

- Storage buffer in shared memory
- Subsystem Management Table in shared memory
- Storage buffer in kernel
- Subsystem Management Table in kernel
- `ntl_reader` daemon
- `nktl_daemon` daemon
These elements are explained in the following sections of this chapter.
nettlgen.conf(4)

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

NOTE

You must obtain this subsystem ID from Hewlett-Packard (see “Assign Subsystem ID” on page 285.

nettlconf(1M)

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

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

nettl(1M)

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

Network Tracing and Logging Support for Troubleshooting

netfmt(1M)

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

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

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

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

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

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

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

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

Using HP-UX Logging and Tracing for Troubleshooting Support

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

- Log only what is needed to solve problems.
- Record all information to diagnose the problem in the log.
- Provide a hex dump to the troubleshooter only as the last resort.
Make each product do as much self diagnosis and repair as possible, and do it quietly. Notify the end user only when intervention is required or requested.

Give the customer what is needed to solve their problems, not the developer's problems.

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

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

**Assign Subsystem ID**

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

To do so, Email a request for a unique subsystem ID for your product to Hewlett-Packard at nettl_support@india.hp.com. In the message identify a suggested interface subsystem name for your product. Check /usr/include/sys/subsys_id.h in your system prior to selecting the name. Do not request names such as lan, lo, ni, X25, and others that are already assigned. You will be assigned this name if it is not already being used.

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

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

**NOTE**

Use the file you receive from Hewlett-Packard as /usr/include/sys/subsys_id.h in your HP-UX device driver development system when you compile your networking device driver.
Classify Trace Data

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

Trace kinds are defined as:

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

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

Refer to the following guidelines when implementing your tracing routines.

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

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

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

Classify Log Data

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

Logged events are considered in the following classes:

- **Disaster**: Signals that the software detected a severe and irrecoverable error condition that typically affects multiple user applications or connections and may jeopardize system integrity. For example, the condition may cause a system crash or corrupt a system table. Another example is when a condition implies that an action generated by one process may damage other processes.
### Creating Networking Device Drivers

**Network Tracing and Logging Support for Troubleshooting**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Error</strong></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 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.</td>
</tr>
<tr>
<td><strong>Warning</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Informative</strong></td>
<td>Describes infrequent operations and current system activities, such as protocol module initiation and termination sufficiently important to post.</td>
</tr>
</tbody>
</table>

### What and When to Log

The most important part of logged messages is the ASCII string describing the event, which is the first item a system operator might see on the system console following an event in the network operation. Deciding what to log and when it should be logged often involves trade-offs in terms of usability, performance, schedule constraints, and management and peer pressure. Other than the items outlined in the preceding tracing or logging sections, some general guidelines include:

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

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

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

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

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

Encapsulate logging calls in functions or macros.

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

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

Identify error recovery procedures for Disaster and Error class events.

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

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

```c
#include <net_diag.h>
Contains macro calls to check that tracing and logging is enabled for the subsystem.
```

```c
#include <subsys_id.h>
Contains subsystem information and definitions for log classes and trace kinds.
```

The function calls for kernel subsystems capture trace and log data.

**KTRC_CK()**

This macro is used to trace on an all interface device basis. It allows the calling process to verify if tracing is enabled for the current subsystem. The returned value is one (1) if tracing is enabled. It is defined as:

```c
KTRC_CK(subsys_id, trace_kind)
```

- **subsys_id** Unique subsystem ID of the calling subsystem. The number is assigned by Hewlett-Packard; see “Assign Subsystem ID” on page 285.
- **trace_kind** Defines trace kind; these are defined in `subsys_id.h` header file and are detailed in “Classify Trace Data” on page 286, as follows:
  - **HDR_IN_BIT** Inbound header tracing mask
  - **HDR_OUT_BIT** Outbound header tracing mask
  - **PDU_IN_BIT** Inbound PDU tracing mask
  - **PDU_OUT_BIT** Outbound PDU tracing mask
  - **PROCEDURE_TRACE_BIT** Procedure entry/exit trace
  - **ERROR_TRACE_BIT** Error tracing mask
  - **LOGGING_TRACE_BIT** Log call tracing mask
  - **LOOP_BACK_BIT** For loopback
  - **PTOP_BIT** For point to point
NOTE

There are some alias or redefine the trace_kind functions in the net_diag.h header file mentioned earlier:

```
#define TR_LINK_LOOP PDU_OUT_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:

```
if (KTRC_CK(ENET_ID, TR_LINK_INBOUND))
{
    ktrc_write(...);
}
```

**ktrc_write()**

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

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

```
ktrc_write ( int subsys_id, int trace_kind, int path_id,
            int device_id, caddr_t tl_packet,
            int tl_packet_cnt)
```

<table>
<thead>
<tr>
<th>subsys_id</th>
<th>Unique subsystem ID of the calling subsystem (number assigned by Hewlett-Packard; see the “Assign Subsystem ID” on page 285.</th>
</tr>
</thead>
<tbody>
<tr>
<td>trace_kind</td>
<td>Defines the kind of trace. All kinds are defined in the header file subsys_id.h. The following is the defined trace kind values (see “Classify Trace Data” on page 286). They can be OR'ed to produce the combination of trace kinds.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HDR_IN_BIT</th>
<th>Inbound header tracing mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>HDR_OUT_BIT</td>
<td>Outbound header tracing mask</td>
</tr>
<tr>
<td>PDU_IN_BIT</td>
<td>Inbound PDU tracing mask</td>
</tr>
<tr>
<td>PDU_OUT_BIT</td>
<td>Outbound PDU tracing mask</td>
</tr>
</tbody>
</table>
PROCEDURE_TRACE_BIT  
Procedure entry/exit trace

STATE_TRACE_BIT  
State machine tracing mask

ERROR_TRACE_BIT  
Error tracing mask

LOGGING_TRACE_BIT  
Log call tracing mask

LOOP_BACK_BIT  
For loopback

PTOP_BIT  
For point to point

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

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

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

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

As with logging, developers should encapsulate tracing calls in functions or macros. The code scenario in the following section shows a typical use of tracing calls.
Tracing Code Sample  The following example shows a trace of an outbound packet whose various parts are located in distinct memory locations. The trace uses the vectored data capability of the ktrc_write() call. The same could be accomplished using an mbuf chain as well.

```c
#include "../h/netdiag1.h"
#include "../h/net_diag.h"
#include "../h/subsys_id.h"
#include "../h/uio.h"
...
#define MAX_BUF 3 /* any number of vectors are allowed */
#define TRACE 0
#define FALSE 0
...

int trace_pdu_out(pdu_hdr, pdu_hdr_len, pdu_data, pdu_data_len)
char *pdu_hdr;
int pdu_hdr_len;
char *pdu_data;
int pdu_data_len;
{
    int kind;
    int device_id;
    int path_id;
    short subsys_id;
    struct iovec tl_buf[MAX_BUF];
    int tl_buf_cnt;

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

    if (KTRC_CHECK(subsys_id, kind, device_id))
    {
        /*
         * Tracing is enabled for this subsystem
         * and kind combination.
         */
    }
```


41  tl_buf[0].bufptr = pdu_hdr;
42  tl_buf[0].buflen = pdu_hdr_len
43  tl_buf[1].bufptr = pdu_data;
44  tl_buf[1].buflen = pdu_data_len;
45  tl_buf[2].bufptr = NULL;
46  tl_buf[2].buflen = 0;
47  tl_buf_cnt = 2;
48
49  ktrc_write(subsys_id,
50    kind,
51    path_id,
52    device_id,
53    &tl_buf,
54    tl_buf_cnt);
55  }
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 one (1) if logging is enabled. It is defined as:

**KLOG_CK(subsys_id, log_class)**

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

*log_class*  Defines the classification of event. All classes are defined in the header file *subsys_id.h* (see also “Classify Trace Data” on page 286). Four classes are defined for logging messages:

- Informative  Normal messages only
- Warning  Warning messages
- Error  Error condition messages
- Disaster  Critical error messages
**kget_log_instance()**

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

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

**klogg_write()**

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

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

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

- **subsys_id** Unique ID (number assigned by Hewlett-Packard) of the calling subsystem.
- **class** Defines the classification of event. All classes are defined in the header file `subsys_id.h` (see also “Classify Trace Data” on page 286). Four classes are defined for logging messages:
  - Informative Normal messages only
  - Warning Warning messages
  - Error Error condition messages
  - Disaster Critical error messages
- **device_id** Device ID number (for example, `if_unit`) of the calling subsystem message. If this is a nonapplicable parameter, pass in −1.
Creating Networking Device Drivers

Network Tracing and Logging Support for Troubleshooting

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

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

**tl_packet_cnt**
If −1, tl_packet points to an mbuf chain. If the value is greater than zero, it is the number of the iovec structure (as defined in uio.h) the tl_packet points to.

**Logging Code Sample**
The following scenarios describe the intrinsic calls of HP-UX logging facilities. These are typical fragments of code that a subsystem might include to perform logging calls.

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

... 

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

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

int log_disaster()
{
    int class;
    int device_id;
```
Creating Networking Device Drivers

Network Tracing and Logging Support for Troubleshooting

```c
23 event_data_type event_data;
24 short subsys_id;
25 struct iovec tl_buf[MAX_BUF+1];
26 int tl_buf_cnt;

28 /*
29 * Set up variables for call to KLOG_CK()
30 */
31 subsys_id = MY_SUBSYS_ID_NUMBER; /* assigned by HP */
32 class = DISASTER;
33 device_id = -1; /* -1 means not applicable */
34 if (KLOG_CK(subsys_id, class)
35 { /*
36 * Logging enabled for this subsystem
37 * and class combination.
38 */
39 
40 if (log_instance == 0)
41 {
42 /*
43 * There was no previous log instance
44 * associated with this event. This is
45 * the first module to encounter the
46 * problem, so it gets the log instance.
47 * Log instance should be available to
48 * all modules in the subsystem and to
49 * other subsystems.
50 */
51 log_instance = kget_log_instance();
52 }
53 }
54
55 event_data.event_number = THIS_EVENT_NUMBER;
56 event_data.event_type = THIS_EVENT_TYPE;
57 /*
58 * Additional data about the event can be
59 * placed in the data structure. This
60 * data structure is entirely up to the
61 * local developer to design. The
62 * subformatter for this subsystem must
63 * be able to decode the data structure,
```
* but other than that there are no
* restrictions on what gets passed. The
* local developer may choose to use a
* single mbuf chain to hold all the
* event information, or pass a vectored
* buffer to the klogg_write() call to
* hold individual pieces of information.
* *
* Callers should NOT pass strings in this
* function; the event number as shown in
* this example should be used to
* generate an NLS string from a message
* catalog in the subformatter.
* */

tl_buf[0].bufptr = *event_data;
tl_buf[0].buflen = sizeof(event_data_type);
tl_buf[1].bufptr = NULL;
tl_buf[1].buflen = 0;
tl_buf_cnt = 1;

klogg_write(subsys_id,
class,
device_id,
log_instance,
&tl_buf,
&tl_buf_cnt);

return(0);
Formatting Networking Trace/Log Messages

The following sections detail facilities and network device driver developer responsibilities for formatting trace/log data output. Some sections provide code and output examples. “Designing a Subformatter” in Appendix B, shows a generic style subformatter that can handle the preceding logging and tracing examples using the basic calls of netfmt.

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

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

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

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

The formatter filter handles or discards records based on data in each message header. Such filtering is described in the netfmt (1M) manpage. See also netfmt (1M) in “Introductory Overview of HP-UX Tracing and Logging” on page 280.

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

During filtering, the formatter checks the message to make sure it contains good information. If the formatter finds a corrupted message header, an unknown subsystem, a message that is too long to handle, and so forth, it prints an informative message, formats the message header, and discards the remainder of the data. It then continues with the rest of the file.
The formatter provides utility functions that subformatters can call to perform common tasks, such as formatting the message header in a standard fashion, dumping raw data, and outputting the formatted data. These functions are discussed in the following sections, with recommendations for usage. Figure 7-8, “Network Tracing, Logging Elements and Data Flow I,” identifies important tracing/logging subroutines and shows their relationship to tracing/logging facilities shown in Figure 7-7, “Master Agent/Subagents Relationship.” You can add additional functions, if necessary.

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

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

How to design a Subformatter and related issues are included in Appendix B.
Figure 7-9  Network Tracing, Logging Elements and Data Flow II

[Diagram showing network tracing and logging elements and data flow]

- **ktl driver**
- **Subsys Table**
- **Port**
- **netfnt**
- **ntl_reader**
- **Subsystem Table**
- **ntl daemon**

**ASCII records**
- Initial Log Class
- Message Catalog
- nettgen.conf
- Subformatter Library
- Subformatter Function
- Subformatter Options
- Subsystem ID
- Subsystem Name

**Report**

**Trace or Log Data File**

**ktlogg_write()**
**ktrc_write()**

**KLOG_CK()**
**KTRC_CHECK()**
Alternative Means of Development Debugging

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

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

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

System Administration Manager (SAM) is an HP-UX system administration tool. For the HP-UX system administrator, it provides both GUI and cursor based interfaces to configure the system's resources like file systems, network etc. Starting from HP-UX version 11i, SAM has included support for configuring network interface cards controlled by third party device drivers. This is a desirable feature for a third party driver since the system administration for both native HP drivers and the third party drivers falls under the same umbrella of SAM, easing the job of a system administrator.

This section presents an overview of the various components involved in lending SAM support for third party drivers, generic networking configuration supported by SAM, additional configuration and init script files required for integrating with SAM, control flow of get/set requests from SAM to the driver, and defines required at the driver.
SAM Interface to IHV Network Drivers

The following diagram shows various components involved in SAM support to a third party network driver.

**Figure 7-10  Support Components for Third Party Network Driver**

The network driver is assumed to be a STREAMS DLPI networking driver. For further information on writing a STREAMS DLPI networking driver, please refer to earlier sections in this chapter. The remaining document discusses only the issues that are relevant to modifying an existing STREAMS DLPI network driver for SAM support.

A network driver that requires SAM support has to provide a driver configuration file, a driver init script, a driveradmin tool and additional support at the driver.
To interact with a driver, first SAM needs to know the driver name and the device special file name of the driver. The driver configuration file located in the directory `/etc/rc.config.d` contains the required information. The driver configuration file also lists parameters of the driver which can be configured. Specific details on the driver configuration file along with an example are provided in later sections.

Based on the driver name, SAM locates a driver `init` script located in the directory `/usr/sbin`. The driver `init` script is a nexus between SAM and the driver and is called by SAM to modify settings of the parameters that the driver supports. SAM always calls a driver's `init` script in a particular format. It is the responsibility of the `init` script to correctly parse SAM request and issue commands to the driver to modify the NIC settings. Specific details to the driver `init` script file along with an example are provided in later sections.

A driver `init` script passes the request to the `driveradmin`, which issues a request to the driver to set a value for a parameter passed by the `init` script. For more details on writing a `driveradmin` tool for a network driver, refer to “Shared Library Examples for the `driveradmin` and `lanscan` commands” in Appendix C.

A driver has to support additional `ioctl` requests to process requests originated at SAM to modify settings of the driver parameters. A driver also needs to support additional HP-DLPI primitives to process requests originating from SAM to get current settings of the driver. Specific details on the changes required at a driver are given in later sections.

SAM uses an internal SAM executable `/usr/sam/lbin/laninfo` to list all the network interfaces on a host. `laninfo` issues HP-DLPI requests and `ioctl` requests to the driver to get the current settings from the driver. The control flow of `get` and `set` requests from SAM to the driver is discussed in later sections.
Generic Network Configuration Supported by SAM

SAM supports GUI based configuration of Network Interface Cards (NICs). Various card parameters of a NIC are set through SAM. Under basic configuration, SAM supports configuration of IP Address, Subnet Mask, Host Name Aliases, speed, duplex and autonegotiate. Under advanced options, it supports the configuration of Station Address, Broadcast Address and the MTU size.

SAM uses the `ifconfig` utility for setting the IP address, Subnet mask and Broadcast address. If user has given a host name and an alias, SAM updates the `/etc/hosts` files with host name and alias information.

For configuring speed, duplex, autonegotiate, station address and MTU size, SAM calls the driver `init` script.

The following is an example of how SAM calls the driver `init` script to set the various parameters:

```
/usr/sbin/hpenet_init start -1 0 enet0 0 0x001083F60E72 1500 10hd
```

Driver Configuration File and Init Script

SAM requires a driver configuration file to obtain necessary information to access the driver, and a driver `init` script to modify the driver parameters.

Driver Configuration File

The driver specific configuration should be saved under the directory `/etc/rc.config.d` with a file name of `hp<driver_name>conf`. For example, if the driver name is `enet`, the driver configuration file name is `hpenetconf`.

SAM gets two important pieces of information from this file: configuration parameters, and the device special file name.

- Obtaining Configuration Parameters

The `HP_<DRIVER>_INIT_ARGS` statement in this file defines the parameters that are configurable on this card. SAM configures only those values that are in this statement. If a parameter does not apply here, SAM does not support its configuration. It is set as follows:
Creating Networking Device Drivers
Configuring a Networking Driver Through SAM

HP_<DRIVER>_INIT_ARGS="HP_<DRIVER>STATION_ADDRESS
HP_<DRIVER>_MTU
HP_<DRIVER>_SPEED"

This is a fixed setting and should be followed in the driver configuration file.

- Obtaining Device Special File Name

The configuration file should also contain the third party device special file name, which is specified as:

IHV_DLPI_DEVICE_NAME=/dev/<driver_name>

for example,

IHV_DLPI_DEVICE_NAME=/dev/enet

The hp driver_nameconf is created and owned by the driver.

SAM updates the modified configuration parameters in this driver specific configuration file by using the ch_rc command. For example, after setting the interface network speed settings to 100 full-duplex, MTU size to 1200, and MAC address to 0x000629BE051C, the following is a set of entries that SAM places after each operation on the NIC.

HP_<DRIVER>_INTERFACE_NAME[1]=enet1
HP_<DRIVER>_SPEED[1]=100FD
HP_<DRIVER>_MTU[1]=1200
HP_<DRIVER>_STATION_ADDRESS[1]=0X000629BE051C

---

**NOTE**

SAM modifies this file whenever network parameters are modified. Changes to this file should not be performed after initialization to prevent erroneous configurations.

---

Refer to the sample driver_nameconf file, hpenetconf provided in the DDK, to build a driver conf file for the third party driver.

**Driver INIT Script**

The driver specific init script should be saved under the directory /usr/sbin with a file name of hp driver_name_init. For example, if the driver name is enet the driver init script file name is hpenet_init.

This init script is for the exclusive use of SAM, and should be different from any other init scripts a driver might have.
A driver's `init` script is the glue between SAM and the driver. SAM sends requests to modify network configuration parameters to the driver via the `init` script. SAM always calls the driver `init` script with a fixed sequence of parameters. The calling convention of the `init` script by SAM is:

```
/usr/sbin/hp<driver_name>init start <major#> <instance#> <interface name>
<nmid> <station address> <mtu> <interface speed>
```

The parameter definitions are:

- **major#**: Major number of the driver. Typically this is not used by the `init` script. Ignore this field.
- **instance#**: Instance of the NIC on which the `set` operation is to be performed.
- **interface name**: Network interface name, as shown in `lanscan` output. Typically this is not used by the `init` script. Ignore this field.
- **station address**: New MAC address; set the specified NIC's MAC address with this.
- **mtu**: New MTU value; set the specified NIC's MTU with this.
- **interface speed**: New link speed setting for NIC. The valid values for speed settings are:
  - `10HD`: 10Mbps, half-duplex
  - `10FD`: 10Mbps, full-duplex
  - `100HD`: 100Mbps, half-duplex
  - `100FD`: 100Mbps, full-duplex
  - `1000HD`: 1000Mbps, half-duplex
  - `1000FD`: 1000Mbps, full-duplex
  - `AUTO_ON`: Autonegotiation ON

For example, to set interface network speed settings to 100 Full-duplex, MTU size to 1200, and MAC address to 0x000629BE051C, SAM can call the corresponding driver's `init` script as:
Creating Networking Device Drivers

Configuring a Networking Driver Through SAM

```
/usr/sbin/hp<driver_name>_init start 119 1 lan1 6
0x000629BE051C 1200 100FD
```

Any message from the init script after successfully carrying out the set operation is to be redirected to /dev/null. If the init script fails, the error message provided by the script will be displayed by SAM.

Typically a driver init script will call the driver_nameadmin tool to set the values at the driver.

Refer to the sample driver init script hpenet_init in the DDK to construct the hpdriver_name_init script for the third party driver.

SAM/Driver get/set Request Flow

The following section discusses the control flow of get and set requests from SAM to the driver. Implementation details are given to process SAM requests. Where required, example code is provided from the sample driver enet for clarification.

To get The Current Settings of the Driver

SAM does not interact with the driver's init script to get current settings from the driver. Instead, it directly issues HP-DLPI and ioctl requests to the driver.

1. SAM issues DL_HP_PPA_REQ to the driver to get driver configuration information.

   DL_HP_PPA_REQ processing in the sample enet driver provides a good example of implementing support for it in the driver. The driver has to set the MTU field of the dl_ppa_info_t structure to the current MTU size at the NIC. SAM gets the current MTU size from the field.

2. To get the speed, duplexity and autonegotiation values, SAM issues an ioctl request of command type DLPI_LINK_SPEED to the driver. DLPI_LINK_SPEED is defined in the /usr/include/sys/dlpi_ext.h file.

   The process is:

   a. SAM allocates a variable of type struct strioctl.
b. SAM issues an `ioctl` (2) system call with the request code set to `I_STR` and passes the variable allocated in step a as an argument.

The following fields are passed by the `ioctl`:

- `ic_cmd` field set to the constant `DLPI_LINK_SFEED`
- `ic_dp` field points to a structure of type `struct fis`
- `ic_len` field set to `sizeof(struct fis)`

In the structure `struct fis`, the `reqtype` field is set to `IHV_REQ_LINK_SETTING` to get the current speed/duplex/autoneg settings of the link.

While processing this `ioctl` request, the driver sets the `value.i` member of the `struct fis` to one of the following values depending on the link status:

- `IHV_SPEED_10HD` (= 1) for Speed: 10 Duplex: Half
- `IHV_SPEED_10FD` (= 2) for Speed: 10 Duplex: Full
- `IHV_SPEED_100HD` (= 3) for Speed: 100 Duplex: Half
- `IHV_SPEED_100FD` (= 4) for Speed: 100 Duplex: Full
- `IHV_SPEED_1000FD` (= 5) for Speed: 1000 Duplex: Full

In addition, if autonegotiation is turned on at the NIC, the driver should bit-wise OR the above value with `IHV_AUTONEG_ON` (=0x10). When SAM gets this value, it performs a bit-wise AND of the returned `value.i` member with `IHV_AUTONEG_MASK` (=0xF0) to decide the status of autonegotiation.

Some devices or their drivers may or may not support autonegotiation. SAM has to know whether the device/driver supports it. Bit four in the `ioctl` return value indicates autonegotiation status. Bit five indicates autonegotiation support. If the support bit is not set, the autonegotiation status can be ignored by SAM. If the support bit is set, the autonegotiation status value should be considered.

SAM expects this format for the `ioctl` return value in the `value.i` member:

<table>
<thead>
<tr>
<th>bit #</th>
<th>X⁷</th>
<th>X⁶</th>
<th>X⁵</th>
<th>X⁴</th>
<th>X³</th>
<th>X²</th>
<th>X¹</th>
<th>X⁰</th>
</tr>
</thead>
</table>

where X = a value associated with the bit position.
<table>
<thead>
<tr>
<th>Bits 3 - 0</th>
<th>Indicate speed and duplexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 4</td>
<td>Indicates autonegotiation status (1 = ON, 0 = OFF)</td>
</tr>
<tr>
<td>Bit 5</td>
<td>Indicates autonegotiation support (0 = Not supported, 1 = Supported)</td>
</tr>
</tbody>
</table>
The following pseudo code is from the sample enet driver:

```c
<driver>_ioctl_function() {
    
    /* Code to return speed, duplexity and autoneg values to SAM. */
    case IHV_REQ_LINK_SETTING:
        datap->vtype = INTEGERTYPE;
        /* Check the current speed settings at the link and set the ioctl return value. */
        if( enet_iftp->speed == 10 )
            if( enet_iftp->full_duplex)
                datap->value.i = IHV_SPEED_10FD;
            else
                datap->value.i = IHV_SPEED_10HD;
        if( enet_iftp->speed == 100 )
            if( enet_iftp->full_duplex)
                datap->value.i = IHV_SPEED_100FD;
            else
                datap->value.i = IHV_SPEED_100HD;
        /* Check for autonegotiation status */
        if( enet_iftp->conn_type == MII_AUTOSENSE ){
            /* Autonegotiation supported */
            /* Set autonegotiation status bit to ON */
            datap->value.i |= IHV_AUTONEG_ON;
            /* Set autonegotiation suport bit to ON */
            datap->value.i |= IHV_AUTONEG_SUPPORTED;
        }
        break;
    
    break;
}
```
If the speed is unavailable, the ioctl for \texttt{IHV\_REQ\_LINK\_SETTING} will return its error code (presently -1) and the global variable \texttt{errno} will be set to \texttt{ENXIO}. If SAM passes in any \texttt{reqtype} other than the above, the ioctl will return its error code and the variable will be set to \texttt{EINVAL}.

To Set the Current Settings of the Driver

To set modified network configuration parameters, SAM sends requests to the driver via the driver init script. The driver init script in turn calls the \texttt{driveradmin} to process the SAM request. Since the interface between the \texttt{driveradmin} and the driver is driver dependent, the way modified configuration parameters are passed to the driver can't be described here. In a typical implementation, a \texttt{driveradmin} tool may issue an ioctl to the driver for each of the modified configuration parameters. The sample driver \texttt{enet} implementation and the sample \texttt{enetadmin} tool provided with the DDK will provide the required information on handling ioctl requests.

Defines Required at the Driver

The following defines have to be included in a driver header file. It is very important that the values of these defines are not modified by the driver developer. SAM uses these values, and if the driver doesn't follow them the existing mechanism between SAM and a third party driver does not work.

\begin{verbatim}
/* * The following defines are for SAM support */
#define IHV_SPEED_10HD  1
#define IHV_SPEED_10FD  2
#define IHV_SPEED_100HD 3
#define IHV_SPEED_100FD 4
#define IHV_SPEED_1000FD  5
#define IHV_AUTONEG_ON        0x10
#define IHV_AUTONEG_SUPPORTED 0x20
#define IHV_REQ_LINK_SETTING  0x77
\end{verbatim}
Network Monitoring Commands

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

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

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

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

**driveradmin command**

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

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

The driveradmin routine then completes the steps below:

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

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

   The name of the driver returned in the attach originates from the string that is stored by the driver at initialization time in the name element of the hw_ift structure. See “hw_ift_t Structure Description and Initialization” on page 216 for a description of this structure.

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

   /usr/lib/lanadmin/libdsdriver_name.sl

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

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

   dsdriver_name

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

5. driveradmin then uses the handle returned by shl_findsym() to invoke the shared library.
After the user selects the Display command of the LAN Interface Test Mode menu, and just before invoking the shared library function, *driveradmin* displays the status display title, date and time, and the first line of the statistics, which is always the PPA. This output resembles the following:

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

**Arguments Passed to Shared Library Functions**

driveradmin passes the following arguments into the shared library:

- **int fd**: File descriptor of third party's special device filename used by the shared library function to get the statistics.
- **int cur_ppa**: PPA of interface whose statistics are to be displayed.
- **int termlines**: Number of terminal lines in the current screen/window; typically used by a shared library function to determine whether the number of statistics being displayed is greater than the screen length.

**Writing a driveradmin Shared Library**

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

- The shared library function must be located in `/usr/lib/lanadmin`.
- The shared library function must be named `dsdriver_name` (with `driver_name` as the string stored in name in the `hw_ift` structure).

There are no restrictions on what the shared library function can be written to do. For ease of use and consistency with other Hewlett-Packard networking data outputs, each shared library should be written to make the statistics display emulate the statistics displays of existing Hewlett-Packard shared libraries. To promote such consistency for all systems, *driveradmin* always displays the PPA as the first line of the statistics display.
Defining the Statistics

The statistics that the Hewlett-Packard LAN drivers maintain and that the Hewlett-Packard shared libraries display are the MIB-II statistics defined in RFC 1213. These statistics are common to all Hewlett-Packard LAN links, Ethernet, Token Ring, FDDI, and Fiber Channel. In addition, most Hewlett-Packard shared libraries and most Hewlett-Packard LAN drivers maintain the link specific MIB statistics. For example, Hewlett-Packard Ethernet/802.3 drivers maintain the Ethernet-like MIB statistics defined in RFC 1398.

Localizing Output Messages

All outputs from shared libraries should be localized. The shared libraries should use the Hewlett-Packard Native Language Support (NLS) message catalogs. Refer to the HP Native Language Support: User's Guide for further information.

Example for Writing a Shared Library

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

Shared Library Message Catalog

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

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

Getting the Interface Statistics

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

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

**lanscan Command**

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

**Displaying Encapsulation Methods**

A `lanscan` shared library can display the encapsulation methods supported by an interface. The shared libraries are invoked by `lanscan` when the user selects the “-v” (verbose) option on the command line. Since `lanscan` displays information about all LAN interfaces on the system, a different shared library is invoked for each interface. `lanscan` traverses the `hw_ift` linked list to find out what LAN interfaces are configured and what information is to be displayed. See “`hw_ift_t` Structure Description and Initialization” on page 216 for more information on the `hw_ift` structure.

When “-v” is selected:

1. `lanscan` gets the driver name out of the name element of the `hw_ift` structure to find the name of the shared library file to access.
2. `lanscan` does a `shl_load()` of the file with the following form:
   ```
   /usr/lib/lanscan/libpedriver_name.sl
   ```
   where `driver_name` is the string stored in the name element of the `hw_ift` structure. Every other part of the full pathname shown above is hardcoded in `lanscan`. 

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

   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/pedriver_name.cat

**NOTE**

Use this path and file name coding to avoid confusion and to conform to Hewlett-Packard shared libraries and other conventions.
Encapsulation Methods Support

To discover the checking and displaying of supported encapsulation methods, refer to the sample shared library peenet(), which is source code for libpeenet.sl.

The llc_flags element of the hw_ift structure for a given interface tells which encapsulation methods are supported by the driver.

The following example from /usr/include/sio/lan_dlpikrn.h shows the presentation of bit definitions:

/* LLC Encapsulation Types */
#define IEEE0x01 /* IEEE 8022*/
#define HP_EXT_IEEE0x02 /* HP Extended IEEE 8022*/
#define SNAP0x04 /* IEEE SNAP*/
#define ETHERTYPE0x08 /* Ethernet*/
#define NOVELL0x10 /* Ethernet */

driverlinkloop Command

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

Example:

enetlinkloop -i 1 -3 /dev/dev_enet 0x0060B07EAAFD