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. 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. Review this section before beginning development of the 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 the Driver Development Kit, to create the 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 6-1, “Three Layered HP-UX Interface to the PCI Bus,” and is briefly described in the following four subsections:

“Data Link Interface Layer”

“Network Protocol Layer”

“Protocol Interface Layer”

“STREAMS Environment”

Figure 6-1 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

TCP/IP Networking Commands and Applications

Driver Networking Commands and Utilities

User Space

Kernel Space

Protocol Interface Layer

STREAMS Head and Device Files

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 LAPB

FDDI Card
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

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 developers 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:

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

Other References:

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

The flowchart in Figure 6-2, "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.

Figure 6-2 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

2. Auxiliary Code? Y N
   Auxiliary Code

3. Network Management Support? Y N
   Network Management Support

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

5. Auxiliary Files? Y N
   Auxiliary Files

OPTIONS

6. SAM Support? Y N
   SAM Support

7. LAN Commands Support? Y N
   LAN Commands Support

DRIVER COMPLETE
Creating Networking Device Drivers
Networking Driver Structure

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 Overview”.

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” for detailed information about these data structures.

Protection and Synchronization
Describes the OSF/Encore spinlock protection model. Refer to “Protection and Synchronization” 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” 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” 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”.

DLPI Interface
Describes how upper layers are linked to the network drivers via the DLPI. Refer to “DLPI Interface”.

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 Overview” 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” for more detailed information.

Select or go to the next option.
4. Network Tracing and Logging Support
A detailed discussion of the topic is provided in *Chapter 8, “Tracing and Logging in LAN Drivers.”*
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
Refer to *Chapter 9, “SAM Support for LAN Drivers,”* for detailed information on adding SAM support in IHV network interface drivers.
Select or go to the next option

7. LAN Commands Support
Refer to *Chapter 7, “LAN Commands,”* for more information on LAN commands and how to add support for them in IHV network interface drivers. This includes discussion regarding any required shared libraries.
The driver is now complete.

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

---

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

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

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

Each device driver may maintain its `hw_ift_t` and `hw_dlpi_t` structure as part of a larger structure, the driver control block `enet_ift_t`, shown in Figure 6-3, “Networking Driver Control Block and Structures.” The driver control block provides information used in driving and controlling the interface hardware.

**Figure 6-3 Networking Driver Control Block and Structures**

- `enet_ift_t` Private driver data
- `hw_ift_t`
- `hw_dlpi_t`

**hw_ift_t Structure**

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:

```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

- **llc_flags**  
  **Link Level Control** (LLC) encapsulation method.

- **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.

- **name**  
  Driver device name; used for naming shared libraries for `lanscan` and `driveradmin`.

---

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

- **hdw_path**  
  Hardware path obtained by calling `io_node_to_hw_path()` followed by `io_hw_path_to_str()`.

- **hdw_state**  
  Hardware state of the device; zero, if the device is OK. If the device is not available, `hdw_state` is set to `LAN_DEAD`.

- **mac_addr_len**  
  Number of bytes of `mac_addr[]` for MAC address.

- **mac_addr**  
  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.

- **features**  
  Features supported by the device. The following flags are provided:

  - **DRV_MP**  
    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 more information. When this flag is set, the driver cannot use any `spl*` calls.

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

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

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

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

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

- **arpmod_name**  
  The name of ARP STREAMS helper module. This 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” 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_if_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->hdw_state = LAN_DEAD;
hw_ift_ptr->mac_addr_len = 6;
bcopy((caddr_t)(mac_addr), (caddr_t)(hw_ift_ptr->mac_addr), 6);

hw_ift_ptr->features = DRV_MP | DRV_MBLK;

hw_ift_ptr->arpmod_name = (u_char *)"";

hw_ift_ptr->watch_timer = 0;

hw_ift_ptr->ppa = hw_ift_ptr->instance_num;

hw_ift_ptr->reserved2 = 0;
```

**hw_dlpi Structure**

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

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 3, “Multiprocessing,” of this manual and spinlock (KER2) in the HP-UX 11i v1 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:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HW_IFT_LOCK(hw_ift_ptr)</td>
<td>Acquire a spinlock on hwift_lock. hw_ift_ptr: pointer to an hw_ift structure.</td>
</tr>
<tr>
<td>HW_IFT_UNLOCK(hw_ift_ptr)</td>
<td>Release previously acquired hwift_lock spinlock. hw_ift_ptr: pointer to an hw_ift structure.</td>
</tr>
</tbody>
</table>

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:

- **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 clas */
        DRV_CHAR|DRV_PSEUDO|DRV_MP_SAFE,
        -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 = {
        0, enet_rsrv, enet_open, enet_close, 0, &enet_rminfo
    };
    static struct qinit enet_winit = {
        enet_wput, enet_wsrv, 0, 0, 0, &enet_wminfo
    };
    struct streamtab enet_info = {&enet_rinit, &enet_winit};
    streams_info_t enet_str_info = {
        "enet" /* name */
        -1, /* dynamic mj# */
        { &enet_rinit, &enet_winit, NULL, NULL},
        /* streamtab */
        STR_IS_DEVICE|MGR_IS_MP|STR_SYSV4_OPEN,
        /* stream flags */
        SQLVL_QUEUE, /* sync level */
    };
```

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.

```c
#define __LP64__
int driver_attach(uint32_t product_id,
                 struct isc_table_type *isc_ptr)
#else driver_attach(PCI_ID product_id,
                    struct isc_table_type *isc_ptr)
#endif
driver_install()         Four bytes of PCI product ID.
isc_ptr                  Pointer to `isc_table_type` structure.
void driver_init(struct isc_table_type *isc_ptr)
driver_isr(struct isc_table_type *isc_ptr,
caddr_t cb_ptr)
```

The `driver_attach()` and `driver_install()` initialization procedures are common to all HP-UX device drivers. More details of each step are presented in Chapter 4, “Writing a Driver.”

**Calling driver_install()**

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

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

The following is a call to `driver_install()`:

```c
static drv_ops_t enet_drv_ops = {
    NULL, /* open */
    NULL, /* close */
    NULL, /* strategy */
    NULL, /* dump */
    NULL, /* psize */
    NULL, /* reserved */
    NULL, /* read */
    NULL, /* write */
    NULL, /* ioctl */
    NULL, /* select */
    NULL, /* option1 */
    NULL, /* reserved1 */
    NULL, /* reserved2 */
    NULL, /* reserved3 */
    NULL, /* link */
    0, /* device flags */
};
```
Creating Networking Device Drivers
Initializing Networking Device Drivers

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");
    if(!(rv=wsio_install_driver(&enet_drv_info,
                               &enet_drv_ops))){
        if(rv=str_install(&enet_str_info)) {
            wsio_uninstall_driver(&enet_drv_info);
            msg_printf("enet:install failed\n");
        }
    }
    return rv;
}

Calling driver_attach()

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

NOTE

The driver_attach() routine may be called many times before a match is found. For the device in the first slot, the associated driver_attach() routine is called by the number of devices in the PCI backplane. For the device in the last slot of the PCI backplane, the associated driver_attach() routine is called only once.
When the `driver_attach()` routine recognizes the device ID it allocates and initializes its driver control blocks and PCI I/O registers. The `driver_attach()` routine also sets up a driver initialization routine and calls the `isc_claim()` to claim the device. The following is a sample `driver_attach()` routine:

```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
   */
  ...
  ...
```
Initializing Networking Device Drivers

Chapter 6

/*
 * 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,
CFCS_MEMORY_SPACE_ACCESS |
CFCS_MASTER_OPERATION |
CFCS_PARITY_ERROR_RESPONSE |
CFCS_SYSTEM_ERROR_ENABLE |
CFCS_I_O_SPACE_ACCESS);

/* Init and reset the controller */

/* Perform general enet_ift initialization */

/* Setup hwift structure */

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

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

Chapter 6

Initializing Networking Device Drivers

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

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

If initialization is successful, the _init() routine proceeds with the following steps:

- Initializes the MIB structure and the hw_dlpi and hw_ift structures (see the preceding sections hw_ift_t Structure and hw_dlpi Structure 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:

  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 6-1, “Protocol Kind and Value,” summarizes the information a networking driver requires to demultiplex inbound packets for corresponding upper layer protocols. More detailed information is provided in the section DLPI Interface.

<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.

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” section for details).

Protocol Binding

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.

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.
- 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.
- 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.

The relationships of protocol kind, protocol value, and protocol processing for different types of packets are shown in Table 6-1, “Protocol Kind and Value.”

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.
Promiscuous Inbound and Outbound

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 6-2, “Promiscuous Mode Matrix,”* explains each promiscuous mode.

**Table 6-2  Promiscuous Mode Matrix**

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

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

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

The list of commonly used message block functions:

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

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

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

streams_put(), streams_put_release()

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

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 Information

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

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

# ll /dev/enet*

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

The `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” for details of `hw_ift_attach()`.

*Table 6-3, “lanscan Output,”* shows how `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.
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` as previously shown, the IP and ARP streams are set up as listed in the following steps:

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 steams are linked under dummy multiplexer.

### Table 6-3  lanscan Output

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

The DLPI Sequence in Figure 6-4, “STREAMS DLPI Network Driver Sequence,” 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”.

Figure 6-4  STREAMS DLPI Network Driver Sequence
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/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_dlp_t**

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

typedef struct _enet_dlp_t {
    enet_ift_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;
} enet_dlp_t;
uint32_t xidtest_flag;
int mac_type; /
int mac_mtu;
dlsap_t *dlsap_ptr;
uint8_t ssap;
uint16_t sxssap;
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 6-4, “enet_dlpi_data_t Data Fields,” 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>
<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_flag from DL_BIND_REQ; indicates to the driver that XID and/or TEST responses for this stream are to be generated by DLPI driver.</td>
</tr>
<tr>
<td>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>sxssap</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>
</tbody>
</table>
Table 6-4  enet_dlpi_data_t Data Fields (Continued)

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>noloopback_flag</td>
<td>Set when the application wants to handle loopback. This flag is set when</td>
</tr>
<tr>
<td></td>
<td>DLPI_SET_NOLOOPBACK ioctl is issued. DLPI turns on the MSGNOLOOP flag</td>
</tr>
<tr>
<td></td>
<td>in mblk message on every outbound message so driver won't loop back the</td>
</tr>
<tr>
<td></td>
<td>packet.</td>
</tr>
<tr>
<td>no_src_routing</td>
<td>Set when DLPI_NO_SRC_ROUTING is issued.</td>
</tr>
<tr>
<td>arp_stream</td>
<td>Set if this is ARP stream.</td>
</tr>
<tr>
<td>ip_stream</td>
<td>Set if this is IP stream.</td>
</tr>
<tr>
<td>fast_path</td>
<td>Set if application requests to set up fast path.</td>
</tr>
<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</td>
</tr>
<tr>
<td></td>
<td>driver; when the request is complete the streams can be set to the</td>
</tr>
<tr>
<td></td>
<td>correct state.</td>
</tr>
</tbody>
</table>

enet_dlpi_data_ptr_arr[]

This array holds enet_dlpi_data_t pointers to keep track of the open streams.

Opening and Closing a Driver

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

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

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

The actual major number of the enet driver is 239.

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

However, a clone device file for the enet driver is created as follows:

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

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

The function `enet_wput()`, the STREAMS driver's “put” procedure, calls various control functions to service DLPI `M_PROTO` and `M_PCPROTO` messages with local management primitives (information reporting, attach, bind, and others such as multicast and promiscuous). This function consists of a switch table that calls the service function based on message `dl_primitive`. The following Table 6-5, “Message Service Functions,” is a list of service functions:

<table>
<thead>
<tr>
<th>Function Name [prefixed by <code>enet_dlpi</code>]</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>_attach()</td>
<td>The information for PPA to be attached is found from <code>hw_if_t</code> list; <code>dlpi_ioctl()</code> is issued to the driver with primitive <code>DL_HP_HDW_INIT</code>. The <code>enet_dlpi_data_t</code> for this stream is updated with network interface information and the stream DLPI state.</td>
</tr>
<tr>
<td>_bind()</td>
<td><code>DL_BIND_REQ</code> primitive request indicates to bind a DLSAP to the stream. Protocol kind (<code>LAN_TYPE</code>, <code>LAN_SNAP</code> or <code>LAN_SAP</code>) is determined by SAP value in the request. The <code>enet_log_sap_value()</code> function is called. Once driver bind is successful, <code>dlsap_t</code> is allocated and initialized with protocol type and value of SAP. The <code>enet_dlpi_data_t</code> structure for this stream is updated with these bind details.</td>
</tr>
<tr>
<td>_control()</td>
<td>The primitives serviced by this function are — <code>DL_ENABMULTI_REQ</code>, <code>DL_DISABMULTI_REQ</code>, <code>DL_SET_PHYS_ADDR_REQ</code>, <code>DL_PROMISCON_REQ</code>, <code>DL_PROMISCOFF_REQ</code> and <code>DL_HP_HW_RESET_REQ</code>. The respective <code>ioctl</code> commands are issued to the driver via <code>enet_dlpi_ioctl</code>. If the request didn’t complete immediately, this routine sleeps on the address of the sleep object of the <code>dlpi_ioctl()</code>.</td>
</tr>
<tr>
<td>_detach()</td>
<td>Disable all multicasts that were enabled through this stream by issuing <code>dlpi_ioctl()</code>s to the network driver. If promiscuous mode was enabled by this stream, disable it. The <code>clean_str_spu_sw_q()</code> routine is called to clean up any requests in the STREAMS/UX. Finally, update the state in <code>enet_dlpi_data_t</code> to <code>DL_UNATTACHED</code>.</td>
</tr>
<tr>
<td>_get_mib_req()</td>
<td>Services <code>MC_GET_MIB_REQ(sys/mci.h)</code>. The driver <code>ioctl</code> <code>DL_GET_STATISTICS</code> is issued to get current MIB statistics.</td>
</tr>
<tr>
<td>_get_mibstats()</td>
<td>Calls <code>enet_hw_req()</code> function to get the standard MIB statistics from the driver structures.</td>
</tr>
<tr>
<td>_getphyaddr()</td>
<td>The <code>enet_hw_req()</code> function is called, which selects the permanent ROM physical address of the network interface, to service <code>DL_PHYS_ADDR_REQ</code>.</td>
</tr>
<tr>
<td>_info()</td>
<td>A service function for <code>DL_INFO_REQ</code>. The information is returned upstream in structure <code>dl_info_ack_t</code>. If the PPA is not attached yet, mac type and mtu is set to <code>DL_CSMACD</code> and <code>IEEE8023_MTU</code>.</td>
</tr>
<tr>
<td>_multicast_list()</td>
<td>This function is called to service the <code>DL_HP_MULTICAST_LIST_REQ</code> primitive. In turn, this function calls driver <code>dlpi_ioctl()</code> to get the list by passing the command <code>DL_HP_GET_MIB_STATS</code>.</td>
</tr>
</tbody>
</table>
Creating Networking Device Drivers
STREAMS DLPI Overview

Chapter 6 191

Table 6-5 Message Service Functions (Continued)

<table>
<thead>
<tr>
<th>Function Name [prefixed by enet_dlpi]</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>_ppa_req()</td>
<td>Receipt of DL_HP_PPA_REQ results in this function being called. The hw_ift_t list is searched for this PPA and the information from hw_ift_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_ift-&gt;hw_state upstream in response to the DL_HP_HW_STATUS_REQ request.</td>
</tr>
<tr>
<td>_subs_bind()</td>
<td>When DL_SUBS_BIND_REQ is received, this function is called. If the dl_subs_bind_class is DL_PEER_BIND, a new dlsap_t is allocated and initialized with protocol type and value of SAP. With DL_HEIRARCHICAL_BIND the dlsap_addr information in enet_dlpi_data_t is updated with bind details.</td>
</tr>
<tr>
<td>_subs_unbind()</td>
<td>For each dlsap_t bound, compare the unbind request SAP. If there is a match, routine enet_unlog_sap_value() is called.</td>
</tr>
<tr>
<td>_unbind()</td>
<td>The function enet_unlog_sap_value() is called. dlsap_t is deallocated and the bind information in enet_dlpi_data_t is set to the default value.</td>
</tr>
<tr>
<td>_xidtest_out()</td>
<td>This function services DL_TEST_REQ, DL_TEST_RES, DL_XID_REQ and DL_XID_RES. It builds the test/xid packet and sends it to the driver using dlpi_output().</td>
</tr>
</tbody>
</table>

IOCTL Processing

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

The sample driver implements DLPI_IOC_HDR_INFO, DLPI_IOC_DRIVER_OPTIONS, and DLPI_SET_NOLOOPBACK.

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

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

Depending on the device capabilities, the driver has to reset the device features which are assumed to be true by default by the transport stack. The features include driver checksum offload (DRIVER_CKO), copy on write (DRIVER_COW), long fat pipe (DRIVER_LFP) and long narrow pipe (DRIVER_LNP). The current version of the HP-UX 11i v1 Driver Development Guide (DDG) does not provide any details on implementing support for the previously listed features. So follow the implementation as given in enet_dlpi_process_ioctl() routine in the sample enet driver to inform the transport stack that the driver does not support any of these features.
DLPI_SET_NOLOOPBACK ioctl causes the enet_dli_data->nolloopback_flg to be set to the value specified in the ioctl parameter.

**Transmission of Message Blocks**

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

**Regular Data Path**

The regular data path message transmission works as follows. The streams “put” function enet_wput() receives the DL_UNITDATA_REQ primitive request from the application to send a message to a destination specified in the unitdata message. The enet_wput() function calls the enet_dli_unitdata_out() function to service the request. The enet_dli_unitdata_out() function applies sanity checks for the stream’s DLPI state and request parameters and builds the LLC header. The LLC header message block is linked with the first M_DATA (with DL_UNITDATA_REQ) and calls the driver’s output routine enet_hw_req().

**Fast Path**

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

**Reception of Message Blocks**

The message is received by the enet_dli_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_dli_unitdata_in() or enet_dli_fast_in(), based on whether fast path is set or not.

The enet_dli_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_dli_fast_in() was discussed in “Fast Path” earlier in this section.

**DLPI Primitives and IOCTLs**

The following Table 6-6, “DLPI Primitives and IOCTLs,” 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 the “Driving the NIC” section.
### Table 6-6  DLPI Primitives and IOCTLs

<table>
<thead>
<tr>
<th>DLPI Primitive or IOCTL</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DLPI PRIMITIVES DEFINED BY DLPI 2.0</strong></td>
<td></td>
</tr>
<tr>
<td>DL_ERROR_ACK</td>
<td>Information reporting</td>
</tr>
<tr>
<td>DL_INFO_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_INFO_ACK</td>
<td></td>
</tr>
<tr>
<td>DL_ATTACH_REQ</td>
<td>Attach</td>
</tr>
<tr>
<td>DL_DETACH_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_ERROR_ACK</td>
<td></td>
</tr>
<tr>
<td>DL_OK_ACK</td>
<td></td>
</tr>
<tr>
<td>DL_BIND_REQ DL_BIND_ACK</td>
<td>Bind</td>
</tr>
<tr>
<td>DL_ERROR_ACK DL_OK_ACK</td>
<td></td>
</tr>
<tr>
<td>DL_SUBS_BIND_REQ DL_SUBS_BIND_ACK</td>
<td></td>
</tr>
<tr>
<td>DL_SUBS_UNBIND_REQ DL_UNBIND_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_DISABMULTI_REQ DL_ENABMULTI_REQ DL_ERROR_ACK</td>
<td></td>
</tr>
<tr>
<td>DL_GET_STATISTICS_REQ DL_OK_ACK DL_PHYS_ADDR_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_PROMISCOFF_REQ DL_PROMISCON_REQ</td>
<td></td>
</tr>
<tr>
<td>DL_UNITDATA_IND DL_UNITDATA_REQ</td>
<td>DLPI Ver 2.0 Connection less Data transfer</td>
</tr>
<tr>
<td><strong>HP EXTENDED DLPI PRIMITIVES</strong></td>
<td>(These are HP extensions to DLPI 2.0 and may change. They are defined in sys/dlpi_ext.h)</td>
</tr>
<tr>
<td>DL_HP_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><strong>(These are HP specific IOCTLs and may change. They are defined in sys/mci.h)</strong></td>
<td></td>
</tr>
<tr>
<td>MC_GET_MIB_REQ</td>
<td>Get MIB statistics</td>
</tr>
<tr>
<td>MC_SET_MIB_REQ</td>
<td>Set MIB statistics</td>
</tr>
<tr>
<td><strong>HP IOCTLs</strong></td>
<td></td>
</tr>
<tr>
<td>DLPI_IOC_DRIVER_OPTIONS</td>
<td>To get driver features.</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.

Data Structures

enet_ift_t

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

typedef struct enet_ift {
    enlan_ift   lancift;
    /*************************************************************/
    * PCI Configuration information - PCI CONF
    *************************************************************/
    ...
    ...;
    /*************************************************************/
    * PCI Control and Status registers. Each field contains the
    * HPA + offset for the network contrl. registers - DEV REG
    *************************************************************/
    ...
    ...
    /*************************************************************/
    * Device Specific Section - DEV SPEC
    *************************************************************/
    struct isc_table_type *isc;
    enet_srom_t *srom;/* Serial ROM layout*/
    ubit32 drv_state;/* Driver state info*/
    ubit32 reset_state;/* Driver reset state*/
    ...
    ...
    /*************************************************************/
    * Transmit Section - TX SECT
    *************************************************************/
    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 */
    ...
    ...

Table 6-6 DLPI Primitives and IOCTLs (Continued)

<table>
<thead>
<tr>
<th>DLPI Primitive or IOCTL</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<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>
For a description of the data fields, refer to Table 6-7, “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>
<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>
</tbody>
</table>
This structure holds generic LAN information for the network interface. It is shown here, and Table 6-8, “enlan_ift Data Fields,” explains the fields.

```c
typedef struct{
    hw_ift_t hwift;
    lan_timer lan_timer;
    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 hw_initialized;
    uint8_t mcast[96];
    uint32_t mcast_ref_cnt[16];
    mib_xEntry *mib_xstats_ptr;
    lock_t* enlanc_lock;
} enlan_ift;
```

### Table 6-7 enet_ift Data Fields (Continued)

<table>
<thead>
<tr>
<th>Field Name/Generic Description</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<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 here, and Table 6-8, “enlan_ift Data Fields,” explains the fields.

```c
typedef struct{
    hw_ift_t hwift;
    lan_timer lan_timer;
    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 hw_initialized;
    uint8_t mcast[96];
    uint32_t mcast_ref_cnt[16];
    mib_xEntry *mib_xstats_ptr;
    lock_t* enlanc_lock;
} enlan_ift;
```

### Table 6-8 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>
Creating Networking Device Drivers
STREAMS DLPI Overview

Chapter 6

Table 6-8  

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

logged_info

The 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;
};
```

Refer to Table 6-9, “Bound SAP Data Fields,” for additional information.

Table 6-9  

<table>
<thead>
<tr>
<th>Field Name / Generic Description</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_if)</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_if, the ioctl command, and the message block with request data.

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

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

```
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.

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

The `enet_dlpi_control()` calls `enet_media_control()` function to process `DL_HP_ENABMULTI` command. `enet_media_control()` function checks validity of multicast address and calls macro `ENET_UPDATE_EXT_MCAST` to update the entry for the multicast address.

This macro checks all multicast entries for the interface. If there is already an entry of the requested multicast address, then it just updates `ref_cnt` and returns.

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

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

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

DL_HP_DISABMULTI

`enet_dlpi_control()` calls `enet_media_control()` function to process the `DL_HP_DISABMULTI` command. `enet_media_control()` function calls the `ENET_DEL_EXT_MCAST` macro.

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

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

DL_HP_PROMISCON

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

```
typedef struct {
    int (*func_ptr) __((struct lan_ift *,
                      void *, void *, u_int));
    /* Function to call for promiscuous packets */
    caddr_t     data_ptr;
    /* queue pointer of the promiscuous stream */
    uint32_t    filter_cnt;
} p_entry_t;
```
/* 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;

The `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`. Also refer to Figure 6.5, "Control Flowchart for Outbound Path."
enet_dlpi_unitdata_out()

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

enet_hw_req()

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

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

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

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

**ENET_TRANSMIT_FRAME**

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

enet_transmit_complt()

This routine processes transmit complete interrupts. Call `enet_slow_complt()` to process non-unicast frames or setup frames. Transmit error handling is done by calling the `enet_trans_error()` routine. If there are frames queued for transmission, call `enet_transmit_pended_frames()` to restart transmission.

enet_transmit_pended_frames()

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

**Inbound Path**

For additional information, refer to `Figure 6-6, “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_receivekts()**

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().

The 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().

The 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()

The 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.

The 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.

The enet_isr() can be called even when the NIC is suspended, see Chapter 15, “On-Line Addition/Replacement.” This is 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 15, “On-Line Addition/Replacement.”

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

Following enum is a field in enet_if 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() or 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 noncoherent 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 2, “HP-UX I/O Subsystem Features.”

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 6-7, “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.

Figure 6-7  Master Agent/Subagents Relationship

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.
Creating Networking Device Drivers
Network Management Support

When replacing the /dev/netman, the following ioctls 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.

void 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:

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

return_nmid

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

int return_nmid (int ifIndex)

IfIndex

The 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 ifOutOctests;
counter ifOutNUcastPkts;
counter ifOutDiscards;
counter ifOutErrors;
gauge ifOutQlen;
int ifSpecific;
} mib_ifEntry;

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