Mentat DLPI Driver Template for Open Transport

The Mentat DLPI Driver Template is device independent code which may be used by STREAMS device driver writers to simplify the task of writing a DLPI compliant driver which supports DL_CSMACD and DL_ETHER media. Under Open Transport this template is provided as a Shared Library and ANSI C header files.

To use the template, a driver writer needs to supply only hardware specific code. The DLPI Driver Template provides full support for

- DLPI Version 2.0 Local Management Service Primitives and Connectionless-mode Service Primitives (Sections 4.1 and 4.3 of the DLPI 2.0 Specification[1]).
- 802.2 TEST and XID operations (Section 4.4 of DLPI 2.0 Specification and Section 5 of IEEE 802.2 Standard[2]).
- DIX Ethernet, 802.x, 802 SNAP, and raw IPX encapsulations on a per-stream basis.
- Promiscuous modes and Multicast.
- Mentat "Fast Path" extension.
- Interface MIB statistics per RFC 1573 and Ethernet MIB statistics per RFC 1643.
- Mentat Network Monitor extensions.

In particular, the common code efficiently handles all the details of inbound packet delivery to possibly multiple streams. The hardware specific code need only present buffers containing received data to the common code for correct delivery. On outbound packets the hardware specific code will be handed fully-formed data frames ready to be placed in hardware transmit buffers.

As a starting point for writing the hardware code, a working loopback driver is provided in a heavily commented ANSI C source file. The sample

---

1. Data Link Provider Interface Specification, UNIX International, OSI Work Group, Revision 2.0.0 (August 20, 1991).
2. Information processing systems—Local area networks—Part 2: Logical link control, ISO 8802.2: 1989, ANSI/IEEE Std 802.2-1989, IEEE, New York, 1989.

driver and this document combine to provide the necessary details on how to write a STREAMS device driver using the Mentat DLPI Driver Template.

1 Driver Template Overview

The common code for the DLPI Driver Template is provided as an Open Transport Shared Library. A header file, dlpiether.h, defines the interface to the common code. A separate header file, dlpiuser.h, defines the user interface to the driver, including MIB structures and network monitor extensions.

The STREAMS interface to the driver (open, close, put, procedures, service routines, and streamtab entry) must be supplied in the hardware-specific code. This design was chosen for two reasons. First, this gives the hardware code control over all its entry points, rather than having some hidden in the common code. Second, and most importantly, this permits the hardware code to optimize the main transmit and receive data paths for its particular hardware architecture.

The common code[1] provides a small number of entry points called by the hardware code, for such operations as initialization and inbound packet delivery. Likewise, the common code expects the hardware code to provide a small number of functions to perform such actions such as enabling the hardware and creating address filters for multicast addresses.

The relationship between open streams and the hardware code is maintained in data structures defined in dlpiether.h. These structures are incorporated into structures defined in the hardware code. Most of the fields in these structures are maintained by the common code, but a few are useful to and maintained by the hardware code, e.g., the Ethernet address associated with a board.

The common code handles all the details of the various DLPI binding semantics for Ethernet, 802.x, 802 SNAP, and raw IPX binding. For enabling promiscuous mode and multicast addresses, the common code handles the DLPI semantics and calls the hardware code to perform any board-specific operations, such as enabling hardware address filtering. The multicast support will function correctly whether or not the board provides multicast address filtering.

The key to high performance for data packets is Mentat's Fast Path extension to DLPI. The basic idea is to create the necessary frame header

1. We will use"common code" to refer to the DLPI Driver Template code in dlpiether.c the Shared Library and "hardware code" to refer to the hardware specific code you must write.

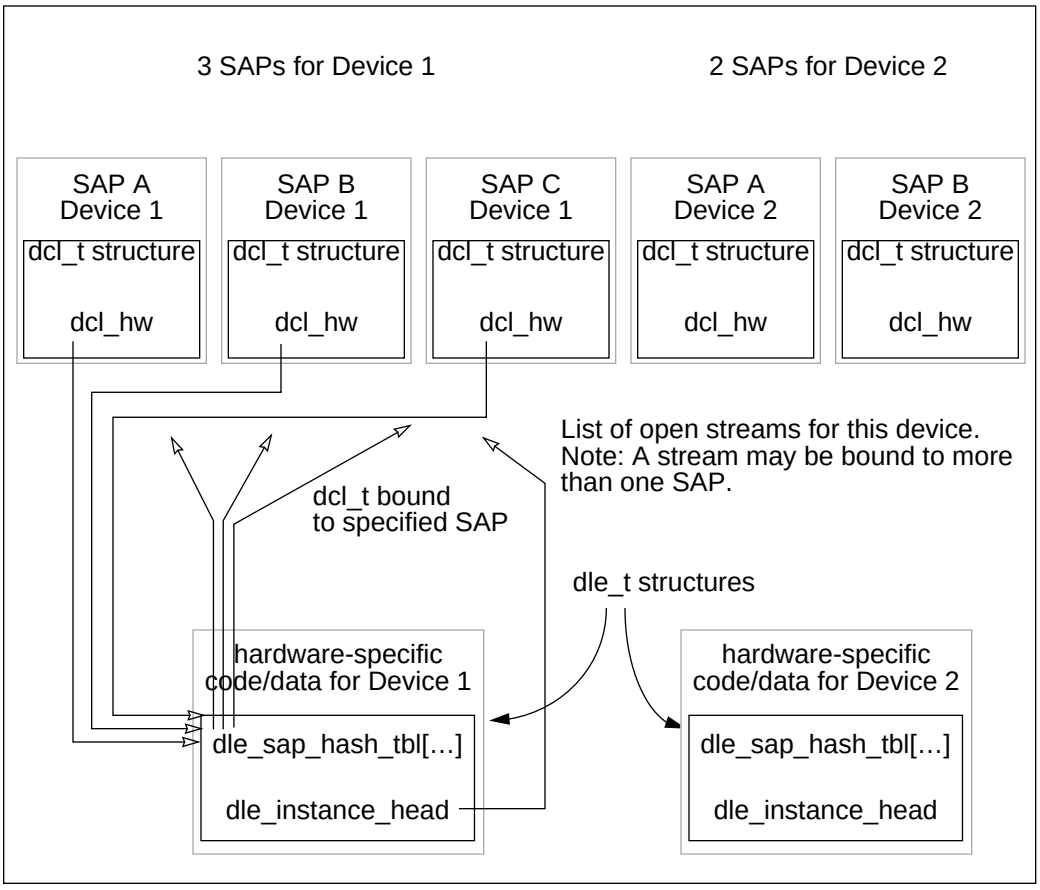

Figure 1: Data Structures Linking Open Streams, SAPs, and Driver Code

for packets only once. Once Fast Path has been negotiated for a particular source/destination address pair, the upper-level protocol can send all data packets to the driver as M_DATA messages with a complete frame header in place in front of the actual data. The result is a substantial reduction in processing overhead for data packets. Both Open Transport's Appletalk and TCP/IP protocol stacks make use of this optimization. The mechanism by which Fast Path is enabled and used by upper level protocol modules is fully described in Section 6.

The file loopback.c included with the DLPI Driver Template is a working loopback driver which is heavily commented and designed to be a starting point for writing hardware-specific driver code. Most features described in this document are illustrated in this sample driver.

1.1 Principal Data Structures

The common code completely controls the association between open streams, bound SAPs, and the hardware code. This relationship is shown in Figure 1.

The instance data for the driver (pointed to by the queue's q_ptr field) includes a dcl_t structure at its beginning. The dcl_t structure has a pointer, dcl_hw, to the dle_t structure for the corresponding hardware code. This association is established when the hardware code calls the common code's *dle_open* function. If there are multiple hardware cards (two are shown in Figure 1) each will have its own dle_t structure.

The common code maintains a hash table, dle_sap_hash_tbl[…] in the dle_t structure. This table contains an element (bind_t structure) for each bound SAP. Each bind_t structure contains a pointer to the dcl_t structure of the stream bound to the SAP represented by that bind_t element. Table 3 and Figure 1 (below) provide additional details.

To repeat, the relationship between these structures is fully maintained by the common code. Most fields in the structures are also maintained by the common code; a few fields are useful to and maintained by the hardware code, e.g., the Ethernet address associated with a board.

1.2 Common Code Structures

The common code requires the instance data for the device driver to begin with a dcl_t structure starting at q_ptr. If the driver requires additional per-stream data, these can be added to the instance data following the dcl_t structure, e.g.,

```
typedef struct mydev_s {
        dcl_t           mydev_dcl;
        int             mydev_extra1;
        int             mydev_extra2;
        …
} mydev_t;
```

The common code itself allocates the instance data when *dle_open* is called from the stream's open procedure (see description of *dle_open* in Section 2).

The fields of the dcl_t structure are described in the following table. Unless noted otherwise, these fields may not be altered by hardware code; in addition, those marked "private" have no function outside of dlpiether.c.

Table 1: Fields in the dcl_t Structure

Field	Description
dcl_addr_list	List of dle_addr_t structures containing the multicast Ethernet (MAC) addresses associated with this device instance.

Table 1: Fields in the dcl_t Structure (Continued)

Field	Description
dcl_flags	Private, except for the F_DCL_RAWMODE_OK bit (see xtra_hdr_len discussion in Section 6).
dcl_framing_type	Framing type as set by I_OTSetFramingType *ioctl*. See dcl_mac_type.
dcl_hw	Pointer to dle_t structure for this drivers hardware code. Hardware-specific driver code should use this pointer to retrieve per-board data structures.
dcl_mac_type	MAC type as returned by DL_INFO_ACK. This is DL_ETHER unless framing type has been set to kOTFraming8022 with an *ioctl*, in which case DL_CSMACD is returned.
dcl_pad[3]	(private)
dcl_rq	This stream's read-side queue.
dcl_sap	The 2-byte SAP from a DL_BIND_REQ.
dcl_snap[5]	SNAP header from a hierarchical DL_SUBS_BIND_REQ.
dcl_state	DLPI state.

Hardware code must maintain a dle_t structure for each separate hardware port. Thus, if a card supports four ports, the hardware code will provide four dle_t structures. If the driver requires additional per-board data, additional fields can be added to the board data following the dle_t structure, e.g.,

```
typedef struct board_s {
       dle_t          board_dle;
       int            board_extra1;
       int            board_extra2;
       …
} board_t;
```

The fields of the dle_t structure are described in the following table. Unless noted otherwise, these fields may not be altered by hardware code; in addition, those marked "private" have no function outside of dlpiether.c; those marked "hardware" must be maintained by the hardware code as described in Section 9.

Table 2: Fields in the dle_t Structure

Field	Description
dle_bind_list	(private) Pointer to list of all bind_t structures for all dcl_t's associated with this dle_t. See Table 3 and Figure 1 for details.
dle_bound_count	(private) Number of entries in dle_bind_list.
dle_current_addr[6]	(hardware) Current physical Ethernet address. This is the local address used for matching inbound packets and the source address for outbound packets.
dle_estatus	(hardware) One of two embedded MIB structures; see Section 8.
dle_factory_addr[6]	(hardware) Factory Ethernet address. This value is returned by DL_PHYS_ADDR_REQ, but otherwise unused by the common code.
dle_hw	(hardware) Structure defining the three hardware-specific functions (*start*, *stop*, and *reset*) called from common code.
dle_hw_addr_list	List of all physical and multicast addresses, with reference count of streams using them. This list of addresses is passed to the hardware code "reset" function whenever the number of multicast addresses changes.
dle_instance_head	Head of the list of all open streams referencing this dle_t.
dle_intr_active	(private) Set to 1 while code is running in interrupt context (*dle_inbound* has been called).
dle_istatus	One of two embedded MIB structures; see Section 8.
dle_match_any	(private) List of promiscuous binds. See DL_PROMISCON_REQ discussion in Section 4.
dle_match_any_802	(private) List of promiscuous 802.2 binds.
dle_match_any_count	Number of dle_match_any entries; passed to hardware "reset" function.

Table 2: Fields in the dle_t Structure (Continued)

Field	Description
dle_match_any_ multicast	(private) List of promiscuous multicast binds. See DL_PROMISCON_REQ discussion in Section 4.
dle_match_any_ multicast_count	Number of dle_match_any_multicast entries; passed to hardware "reset" function.
dle_match_matched	(private) List of binds for packets that have matched at least one other binding. See DL_ PROMISCON_REQ discussion in Section 4.
dle_min_sdu	Minimum transmit size from clients. Set to zero by *dle_init*.
dle_refcnt	Number of open streams referencing this dle_t.
dle_sap_hash_tbl[64]	(private) Hash table of SAP binds; see Figure 1 and Figure 1.
dle_xtra_hdr_len	(hardware) Extra space the hardware code needs in front of the Ethernet frame header in outbound M_DATA messages.

All bind information is maintained in bind_t structures stored in the dle_ sap_hash_tbl[…] or in one of the dle_match_xxx lists. These lists are shown schematically in Figure 1 and the fields of the bind_t are shown in the following table. Note that the bind_t structure is private to the common code; it is presented here only to provide a better understanding of how the common code works.

Table 3: Fields in the bind_t Structure

Field	Description
bind_dcl	Pointer to the dcl_t structure for the driver instance which is bound to bind_sap.
bind_flags	Bit mask identifying the type of SAP.
bind_hash_next	Pointer to the next bind_t in this hash chain.
bind_next	Pointer to the next bind_t in the list of all bind_t structures for this dle_t. List head is dle_bind_list.
bind_sap	The SAP value for this bind_t entry.

Table 3: Fields in the bind_t Structure (Continued)

Field	Description
bind_sap_next	Pointer to the next bind_t with "equivalent" SAP value. For entries in the hash table, "equivalent" means the same SAP value (the two "SAP C" entries in Figure 1). For the various dle_match_xxx lists, all entries in a list are "equivalent", so all entries are connected through this field. For inbound packets, all streams on the same bind_sap_next list will get copies of the same packets.

Two things are worth noting about the lists in Figure 1.

1. The hash chains in dle_sap_hash_tbl[…] linked by bind_hash_next will contain a single element for all the commonly used SAP values. This is ensured by choice of hash function, DLE_SAP_HASH_VALUE.

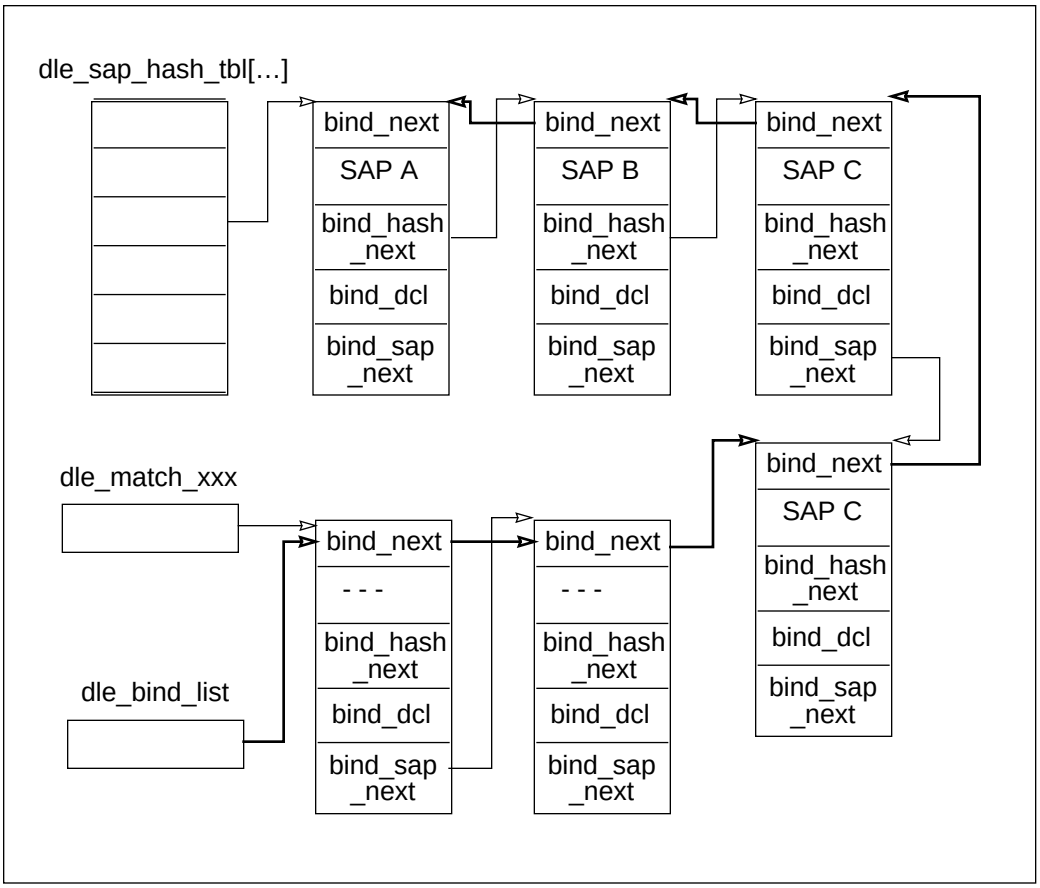

Figure 2: SAP Hash Table bind_t Lists.

2. Whether a “matching” SAP is found in the dle_sap_hash_tbl[…] or in one of the dle_match_xxx lists, following the chain through the bind_sap_next field will give pointers to all streams to which inbound packets must be delivered. In particular, this design requires no “special cases” for delivering packets to promiscuous streams.

2 Common Code Entry Points

dlpiether.c exports nine functions for use by the hardware code.

Table 4: Common Code Entry Points

Function	Description
dle_close	Removes a stream from common code structures.
dle_inbound	Performs all common code processing of inbound packets.
dle_inbound_error	Delivers bad packets to “interested streams”.
dle_init	Informs common code that board is now available.
dle_open	Adds a stream to common code structures.
dle_rsrv_ctl	802.2 XID and TEST packet processing.
dle_terminate	Informs common code that board is no longer available.
dle_wput	Called from write-side put procedure to process outbound packets.
dle_wput_ud_error	Formats DL_UDERROR_IND messages.

The remainder of this section describes these functions and summarizes important points about their use. Additional details are provided in Section 9.

dle_close

This function must be called from the device close procedure.

Prototype

```
int
dle_close (
                queue_t         * q);
```

Argument	Description
q	The queue argument passed to the close procedure, i.e., this module instance's read-side queue.

Return Value

Returns 0 on success or an error code on failure.

Notes

1. *dle_close* frees the queue's instance data and removes this instance from the list of open instances. Therefore, any references to instance data fields, e.g., to free a dynamically allocated memory block, must be done before *dle_close* is called.
2. If this is the last stream bound to a particular multicast address, the hardware code "reset" function is called to change the on-board filter.
3. The dle_refcnt field in the dle_t is decremented.

dle_inbound

dle_inbound is called for every inbound packet. It performs all the necessary operations for delivering the packet to streams registered to receive them, including those bound to a particular SAP or for which multicast and promiscuous modes have been enabled.

Prototype

```
void
dle_inbound (
        dle_t       * dle,
        mblk_t      * mp);
```

Argument	Description
dle	Pointer to the dle_t structure for this board / port.
mp	Message containing the inbound data packet. The message may contain multiple message blocks. The Ethernet frame header is assumed to start at b_rptr of the first message block.

Notes

1. For hardware devices, *dle_inbound* is called from the hardware code's inbound interrupt routine in interrupt context. All packets to be delivered to a stream are placed on the driver's read-side service queue for that stream. The operating system will schedule the service routine to run after the hardware code returns from interrupt context. A typical service routine is shown in the description of *dle_rsrv_ctl* below.
2. If multiple streams are to receive the packet, copies of mp are created with *dupmsg*.
3. The message format for delivering packets upstream depends upon the stream's mode, as defined in the following table:

Mode	Message Type Delivered Upstream
Fast Path: DL_IOC_HDR_INFO *ioctl* has been issued on the stream. See Section 6 for details.	All inbound packets sent to our local address are passed upstream as M_DATA messages. All inbound messages for multicast or broadcast addresses are passed upstream as M_PROTO / DL_UNITDATA_IND messages with dl_group_address properly set.
Raw Mode: I_OTSetRaw-Mode *ioctl* with subcommand kOTSetRecvMode has been issued on the stream, disabling Fast Path. See Section 7 for details.	All inbound packets are passed upstream as M_DATA messages with a dl_recv_status_t structure inserted in front of the packet. The *ioctl* requests this behavior for normal packets only, or for both normal and error packets.
Default: Stream just opened and neither Fast Path nor Raw Mode has been set.	All inbound packets are passed upstream as M_PROTO/DL_UNITDATA_IND messages.

4. If xtra_hdr_len passed to *dle_init* is nonzero, it is the responsibility of the caller to ensure that mp–>b_rptr points at the start of the Ethernet frame and not any extra data in front of the frame header.

---

dle_inbound_error

This function is called by hardware code to deliver packets containing errors to promiscuous streams that have registered to receive such packets. The function is optional and only required if the driver wishes to support the optional network monitor extensions (Section 7).

Prototype

```
void
dle_inbound_error (
        dle_t           * dle,
        mblk_t          * mp,
        unsigned long   flags);
```

Argument	Description
dle	Pointer to the dle_t structure for this board / port.
mp	Message containing the corrupt packet.
flags	A bit mask containing the result of OR'ing together all the values in Table 9 which apply to the packet. DL_ERROR_STATUS will be set by the common code.

Notes

1. The hardware code need only call this function after being called in *dlehw_address_filter_reset* with accept_errors > 0, indicating that at least one stream has registered to receive error packets.
2. The dlehw_recv_error_flags field in the dle_hw field of the dle_t structure must be initialized by the hardware code to indicate which errors can be received by the underlying hardware. The bit mask values are defined in dlpiuser.h and described in Table 9 of Section 7.

---

dle_init

This function must be called from the hardware code's initialization routine. It allows the common code to initialize several dle_t fields.

Prototype

```
void
dle_init (
        dle_t           * dle,
        uint            xtra_hdr_len);
```

Argument	Description
dle	Pointer to the dle_t structure for this board / port.
xtra_hdr_len	The number of bytes to be reserved between db_base and b_rptr in M_DATA messages created by *dle_wput*.

Notes

1. This function must be called once per board (dle_t instance) before the first stream open of the device is performed. For most drivers this is most conveniently done from the *InitStreamModule* function.
2. Several dle_t fields maintained by the hardware (see Table 10, Section 9) must also be initialized before the first stream open. It is strongly recommended that these fields be initialized in *Init-StreamModule* prior to calling *dle_init*.
3. *dle_init* initializes the following fields of the dle_t structure:

Field	Value Set by *dle_init*
dle_istatus.speed	10000000 (10Mbits / second)
dle_istatus.mtu	1514 (frame size including Ethernet header, but not trailer).
dle_min_sdu	0 (assume hardware or hardware code will pad short packets as needed).

   If any of these values is unacceptable, they must be set by the hardware code after calling *dle_init*.
4. If xtra_hdr_len == 0, M_DATA messages returned from *dle_wput* contain a complete Ethernet frame header beginning at b_rptr of the first block in the message. Some boards may require space in front of the frame header for other information; for example, ATM LAN Emulation requires a 6-byte area. *dle_wput* will allocate xtra_hdr_len bytes at b_rptr in front of the Ethernet frame header.

dle_open

This function must be called from the device open procedure. It allocates and initializes the fields of the dcl_t structure in the queue's instance data.

Prototype

```
int
dle_open (
        dle_t       * dle,
        queue_t     * q,
        dev_t       * devp,
        int         flag,
        int         sflag,
        cred_t      * credp,
        int         dcl_len);
```

Argument	Description
dle	Pointer to the dle_t structure for this board / port.
q	The queue argument passed to the open procedure, i.e., this module instance's read-side queue.
devp	Device pointer argument passed to the open procedure.
flag	flag argument passed to the open procedure.
sflag	sflag argument passed to the open procedure.
credp	credp argument passed to the open procedure.
dcl_len	Size of the instance data to be allocated.

Return Value

Returns 0 on success or an error code on failure.

Notes

1. *dle_open* allocates the queue's instance data (pointed to by the queue's q_ptr field). A dcl_t structure **must** be the first field of the instance data. The total length to allocate, sizeof(dcl_t) + sizeof(other_stuff), is passed in the dcl_len argument. There is a single instance data allocated, and both the read and write queue q_ptr fields point to it.
2. *dle_open* adds this queue instance to the list of open instances for this board. The head of this list is the dle_instance_head field of the dle argument. Section 9.3 describes the *mi_next_ptr* function which may be used to traverse this list.

3. If there are multiple boards/ports, it is the responsibility of the hardware code to determine the correct dle_t to pass as first argument.
4. The dle_refcnt field in the dle_t is incremented.

---

dle_rsrv_ctl

This function is called from the read-side service routine to process M_CTL messages put on the queue as part of 802.2 XID and Test packet processing.

Prototype

```
void
dle_rsrv_ctl (
        queue_t         * q,
        mblk_t          * mp);
```

Argument	Description
q	The read-side queue.
mp	Message containing the XID/TEST packet.

Notes

dle_inbound is called in interrupt context. XID/TEST packets must be passed to the driver's write-side put procedure, but this cannot be safely done in interrupt context. To resolve this problem, the XID/TEST logic places an M_CTL message on the read-side service queue. When the service routine runs, it must call *dle_rsrv_ctl* to deliver the message to the put procedure. Code for your read-side service routine should look like the following (see Section 9.1 for additional details):

```
int driver_rsrv (queue_t * q)
{
        mblk_t * mp;

        while (mp = getq(q)) {
                if (mp->b_datap->db_type == M_CTL)
                        dle_rsrv_ctl(q, mp);
                else if (canputnext(q))
                        putnext(q, mp);
                else
                        freemsg(mp);
        }
        return 0;
}
```

dle_terminate

dle_terminate must be called by hardware code when the board / port is no longer available for use. This will normally be done from *TerminateStreamModule*.

Prototype

```
void
dle_terminate (
                dle_t           * dle);
```

Argument	Description
dle	Pointer to the dle_t structure for this board / port.

dle_wput

dle_wput is called by the drivers write-side put procedure to process all packets received from upstream, except for M_DATA messages. M_DATA messages are assumed to contain packet data with Fast Path header included, and the hardware code can transmit these directly.

Prototype

```
mblk_t *
dle_wput (
                queue_t         * q,
                mblk_t          * mp);
```

Argument	Description
q	queue argument from the (write-side) put procedure which is calling *dle_wput*.
mp	Message pointer from the (write-side) put procedure which is calling *dle_wput*.

Return Values

There are three distinct "return values" from *dle_wput*:

Return Value	Meaning
NULL	The message passed to *dle_wput* has been consumed, e.g., it has been freed or a *qreply* has already been sent upstream.
non-NULL, db_type != M_DATA	This message must be sent upstream by the caller, typically by calling *qreply*.
non-NULL, db_type == M_DATA	This message contains a complete data packet with Ethernet frame beginning at b_rptr + xtra_hdr_len.

Notes

1. The M_DATA messages returned from *dle_wput* are identical to Fast Path M_DATA messages received from upstream. See Section 6 for details on how to use Fast Path.
2. Hardware code is responsible for computing the data length of packets and inserting the length into the 802.3 Length / Type field **if this field is zero** on return from *dle_wput*. (The Length / Type field are the 2 bytes at offset 12 from the start of the Ethernet frame.) The sample loopback driver (loopback.c) demonstrates an efficient way to compute this total while copying data into hardware transmit buffers.
3. Hardware code must also check for packets which are too long; *dle_wput_ud_error* may be called to return an error for packets which are too long. Short packets must be padded; if not an automatic feature of the hardware, the driver must perform padding (with zero bytes).
4. The start of the Ethernet frame is &mp–>b_rptr[xtra_hdr_len], where xtra_hdr_len is the second argument passed to *dle_init*.

5. Drivers must be prepared to handle messages containing multiple mblks.

---

dle_wput_ud_error

dle_wput_ud_error is used internally by *dle_wput* to return DL_UDERROR_IND messages upstream. It is made available to the hardware code as a simple convenience.

Prototype

```
mblk_t *
dle_wput_ud_error (
        mblk_t      * mp,
        int         dlpi_err
        int         unix_err);
```

Argument	Description
mp	Message passed to write-side put procedure which caused this error condition.
dlpi_err	Value to return in the dl_errno field of the DL_UNITDATA_IND.
unix_err	Value to return in the dl_unix_errno field of the DL_UNITDATA_IND.

Return Values

Returns a completed DL_UDERROR_IND message to send upstream, or NULL if an internal error occurred (in which case mp has been freed).

Notes

1. Only DL_UNITDATA_REQ messages and M_DATA Fast Path messages may be passed to this routine; NULL is returned if another message type is passed in.

3 Hardware-specific Functions and dlehw_t

The hardware code must supply all STREAMS entry points, e.g., streamtab entry, open and close procedures, write-side put procedure, and read-side service routine. In addition the common code requires three entry points into the hardware code: a “start” function, a “stop” function, and a “reset” function. An optional fourth function must be provided if the driver plans to support the network monitoring feature that allows for

sending of packets containing errors (see Section 7). These functions are referenced through function pointers defined within the dle_hw field in the dle_t. This structure (defined in dlpiether.h) is:

```
typedef struct dlehw_s {
        void    (*dlehw_start)(void *);
        void    (*dlehw_stop)(void *);
        void    (*dlehw_address_filter_reset)(
                        void * hw,
                        dle_addr_t * addr_list,
                        ulong addr_count,
                        ulong promisc_count,
                        ulong multi_promisc_count,
                        ulong accept_broadcast,
                        ulong accept_errors);
        int     (*dlehw_send_error)(
                        void * hw,
                        mblk_t * mp,
                        unsigned long flags);
        unsigned long  dlehw_recv_error_flags;
} dlehw_t;
```

The following paragraphs describe these fields.

---

dlehw_address_filter_reset

dlehw_address_filter_reset is called from the common code whenever a "bind" or "unbind" operation changes the promiscuous setting or the list of enabled multicast addresses.

Prototype

```
void
dlehw_address_filter_reset (
        void            * dle,
        dle_addr_t      * addr_list,
        ulong           addr_count,
        ulong           promisc_count,
        ulong           multi_promisc_count,
        ulong           accept_broadcast
        ulong           accept_errors);
```

Argument	Description
dle	The same dle_t structure passed as the first argument to *dle_open*.
addr_list	Head of list of multicast and physical addresses currently enabled.
addr_count	Number of entries in addr_list.
promisc_count	Nonzero if promiscuous mode must be enabled.
multi_ promisc_ count	Nonzero if promiscuous multicast mode must be enabled. (Promiscuous multicast mode means that all multicast packets, regardless of destination address, must be accepted.)
accept_ broadcast	Nonzero if broadcast packets must be accepted.
accept_errors	Nonzero if at least one stream is registered to receive error packets in raw mode.

Notes

1. addr_list contains addr_count dle_addr_t structures:

```
typedef struct dle_addr_s {
        struct dle_addr_s     dlea_next;
        unsigned char         dlea_addr[6];
} dle_addr_t;
```

   This list contains all of the multicast addresses currently enabled on any stream referencing this board / port (dle_t instance).

2. *dlehw_address_filter_reset* may be called frequently, but only in response to user actions, and may be intermixed with other board actions. The hardware code must be prepared to handle this call at any time, e.g., before or after *dlehw_start* is called.
3. Board level filters don't need to be perfect; in fact, except for processing overhead, the common code functions correctly whether or not any filtering is done. In particular, if the board has no hardware filtering capability, all packets should be passed directly to *dle_inbound*. In a worst case, this may require enabling promiscuous mode so that all multicast addresses will be accepted by the hardware.

4. If accept_errors is nonzero, the hardware code must call *dle_inbound_error* if a corrupt packet is received and the hardware code supports the network monitor extension for passing corrupt packets upstream. dlehw_recv_error_flags must be set to indicate the types of error packets the hardware code can handle.

dlehw_recv_error_flags

This field is a bit mask formed by OR'ing together the values in the following table (see dlpiuser.h and Section 7) that indicate the type(s) of errors the hardware is capable of recognizing and reporting.

Value	Description
DL_CRC_ERROR	Hardware can return CRC error packets.
DL_RUNT_ERROR	Hardware can return short packets (< 64 bytes).
DL_FRAMING_ERROR	Hardware can return packets with Ethernet framing errors.

This field only needs to be set if the hardware code chooses to support this network monitoring extension. This field is independent of the *dlehw_send_error* function.

dlehw_send_error

dlehw_send_error must be provided only if the driver plans to support the kOTSendErrorPacket primitive in the I_OTSetRawMode *ioctl*.

Prototype

```
int
dlehw_send_error (
        void            * dle,
        mblk_t          * mp,
        unsigned long   flags);
```

Argument	Description
dle	The same dle_t structure passed as the first argument to *dle_open*.

Argument	Description
mp	The packet to send.
flags	Bit mask defining the type of error packet to be sent. See Table 9 in Section 7 for details.

Return Value

Returns 0 on success. If flags requests a type of error packet that the hardware or hardware code cannot support, EINVAL must be returned.

Note

1. *dlehw_send_error* must be provided only if the driver plans to support the kOTSendErrorPacket primitive in the I_OTSetRaw-Mode *ioctl*. If this entry in the dlehw_t structure is NULL, the *ioctl* will be failed with EINVAL.
2. If DL_CRC_ERROR is set in flags, then the packet data in mp is assumed to contain the CRC value as the last four bytes. For other types of error transmits, the CRC value must be calculated and appended to the data as for normal packets.

---

dlehw_start

Hardware initialization function called from the common code at the time the first "bind" on any stream using the board/port is performed. "bind" includes enabling of promiscuous modes.

Prototype

```
void
dlehw_start (
                void            * dle);
```

Argument	Description
dle	The same dle_t structure passed as the first argument to *dle_open*.

Notes

1. The hardware code should not enable the board to receive packets until this function is called.
1. *dlehw_start* is called after possibly calling *dlehw_address_filter_reset* in case any physical address or multicast filters changed.

dlehw_stop

dlewh_stop is called from the common code after the last "unbind" on any stream using the board / port is performed. "unbind" includes disabling of promiscuous modes.

Prototype

```
void
dlehw_stop (
        void            * dle);
```

Argument	Description
dle	The same dle_t structure passed as the first argument to *dle_open*.

Notes

1. The hardware code must shut off receive interrupts when this function is called.
2. *dlehw_stop* is called after possibly calling *dlehw_address_filter_reset* in case any physical address or multicast filters changed.

4 Supported DLPI Primitives

This section lists in alphabetic order all the DLPI primitives supported by the common code. Ambiguities in the DLPI specification and important implementation details are discussed. It is assumed that the reader has access to the specification, since the bulk of that information is not repeated here.

4.1 DL_BIND_REQ

Since only connectionless primitives are supported, dl_service_mode must be DL_CLDLS and dl_max_conind must be zero. dl_sap values are treated as shown in Table 5.

Table 5: SAP Bindings

SAP Value/Range	Description
1501 ≤ SAP ≤ 0XFFFF	DIX Ethernet SAP.
odd value, SAP ≤ 0XFE	Illegal 802.2 Group SAP; DL_SUBS_BIND_REQ is needed for these SAPS. DL_BADADDR error returned.
even value, SAP ≤ 0XFE	Valid 802.2 SAP.
SAP == 0XAA	This is a SNAP stream and a DL_SUBS_BIND_REQ must follow to specify the 5-byte SNAP header.
SAP == 0xFF	IPX binding.

The DL_BIND_ACK will always return a dl_addr_length of 8: 6 for Ethernet (physical) address plus 2 for the SAP. See Section 4.5 for additional information on address and SAP lengths.

4.2 DL_DISABMULTI_REQ

The hardware code's *dlehw_address_filter_reset* function is called if there are no other streams referencing the address being disabled.

4.3 DL_ENABMULTI_REQ

The hardware code's *dlehw_address_filter_reset* function is called if this is the first stream to enable the specified address.

4.4 DL_GET_STATISTICS_REQ

The DL_GET_STATISTICS_ACK returned by the common code contains two structures, dle_interface_status_t and dle_ethernet_status_t at dl_stat_offset. See Section 8 and dlpiuser.h for more information.

4.5 DL_INFO_REQ

The DLPI specification contains ambiguities in its description of the DL_INFO_REQ and DL_INFO_ACK. The following summarizes how the common code resolves these points.

1. dl_addr_length and dl_addr_offset.

   Correct DLPI behavior requires that dl_addr_length be returned as zero prior to binding a DLSAP. However, several protocol implementations have misused this field and expect dl_addr_length to be the physical address length (6 for Ethernet) before a DLSAP is

bound. Therefore, the common code knowingly violates the DLPI specification to permit these protocols to work.

When in DL_IDLE state, the length returned depends upon the current bind state: 13 after a successful SNAP DL_SUBS_BIND_REQ, 8 otherwise.

2. dl_sap_length.

   The value returned in dl_sap_length depends upon both state and the type of bind(s) which have been performed. In DL_UNBOUND state, the value zero is returned. In DL_IDLE state, the value -2 is returned after a successful DL_BIND_REQ, and -7 is returned after a successful SNAP DL_SUBS_BIND_REQ. The SAP length is negative, showing that the SAP follows the physical address.

3. dl_max_sdu, dl_min_sdu.

   The returned value of dl_max_sdu is the number of bytes available for data, after subtracting out the Ethernet header length, which in turn depends upon the current bind state: 1492 (1514 – 22) after a successful SNAP DL_SUBS_BIND_REQ; 1497 (1514 – 17) for 802.2 SAP; 1500 (1514 – 14) otherwise (DIX Ethernet).

   dl_min_sdu is returned as zero on the assumption that the hardware will automatically pad short frames to the Ethernet minimum of 64 bytes (including trailer).

4. dl_qos_xxx fields.

   The current implementation does not support QOS fields. Zero is returned for all of them.

5. dl_provider_style.

   Only DL_STYLE1 drivers are supported (the DL_ATTACH_REQ and DL_DETACH_REQ primitives are not supported). Additional information about address formats is given in Section 4.14.

4.6 DL_PHYS_ADDR_REQ

The address returned is taken from the dle_current_addr or dle_factory_addr fields of the dle_t structure, which are maintained by the hardware code. Note that DL_SET_PHYS_ADDR_REQ is not currently supported by the common code.

4.7 DL_PROMISCOFF_REQ

The dl_level field of this request must match that of a previous DL_PROMISCON_REQ. The corresponding promiscuous behavior (see next section) is disabled.

4.8 DL_PROMISCON_REQ

The common code action depends upon the value of dl_level in the request and on the current bound state of the stream. The following table summarizes.

Table 6: Promiscuous Mode Processing

dl_level	state == DL_UNBOUND	state == DL_IDLE
DL_PROMISC_PHYS	All packets are delivered; both SAP and destination address are ignored.	Destination address is ignored, but only packets for SAPs already bound to this stream are delivered. Attempts to bind after setting this mode will fail. Stream must bind to all desired SAPs before issuing this request.
DL_PROMISC_SAP	All packets that match any bound SAP (on any other stream) are delivered.	All packets for the bound SAP are delivered. This is effectively a “no-op”.
DL_PROMISC_MULTI	All packets for any multicast destination address are delivered.	All multicast packets which match any SAP bound to this stream are delivered. Attempts to bind after setting this mode will fail. Stream must bind to all desired SAPs before issuing this request.

The common code will call *dlehw_address_filter_reset* if necessary.

4.9 DL_SUBS_BIND_REQ

The common code action depends upon the value of dl_subs_bind_class in the request. There are two cases:

dl_subs_bind_class	Action
DL_HIERARCHICAL_BIND	Hierarchical binds may only be used on streams bound to the SNAP SAP (0xAA). In addition, only one subsequent bind per stream is permitted and the dl_subs_sap_length value must be 5.

dl_subs_bind_class	Action
DL_PEER_BIND	Peer binds may only be used on streams bound to non-SNAP SAPs. The specified dl_subs_sap_length must be 2. Subsequent SAP values ≤ 0xFE (802 binds) cannot be mixed on the same stream with values > 0xFE. Values ≤ 0xFE are 802 values; a value of 0x00 is a special Open Transport 802 promiscuous mode which causes all 802 packets for this machine to be delivered to this stream. For 802 binds, the dl_xidtest_flg value from the original DL_BIND_REQ is inherited by all subsequent peer binds.

4.10 DL_SUBS_UNBIND_REQ

The value of dl_subs_sap_length must be 2 or 5 for removing a hierarchical or peer bind, respectively.

4.11 DL_UDERROR_IND

The common code returns a DL_UDERROR_IND if

- DL_UNITDATA_REQ address fields are malformed.
- an *allocb* fails while processing a request.

In addition, the hardware code must return a DL_UDERROR_IND if it receives a DL_UNITDATA_REQ or M_DATA containing a packet larger than the driver's maximum SDU (dle_istatus.mtu).

4.12 DL_UNBIND_REQ

This request removes all bound SAPs, including those specified in DL_SUBS_BIND_REQ requests.

4.13 DL_UNITDATA_IND

DL_UNITDATA_IND messages are used for all packets for streams in promiscuous mode and for broadcast and multicast packets on Fast Path streams; otherwise inbound packets are sent upstream as M_DATA messages. The source and destination address fields contain the complete DLSAP, i.e., physical address followed by a complete SAP. The address length is 8, except for 802 SNAP. For 802 SNAP, the length is 13 and the 5-byte SNAP header is included in both the source and destination address fields.

4.14 DL_UNITDATA_REQ

The common code is extremely flexible in its processing of DL_UNITDATA_REQ primitives. This flexibility is mandated by protocols which have misinterpreted the DLPI specification with regard to address format. Even though the DLPI specification states clearly that the address is a "the full DLSAP address returned in the DL_BIND_ACK," some protocols are known not to follow this rule and to provide only a 6-byte Ethernet destination address. The following three cases are supported:

dl_dest_addr_length	SAP Value(s) Used
6 (No SAP specified)	The bound SAP for DIX Ethernet, IPX or 802 SNAP. For simple 802, the bound SAP is used in both the DSAP and SSAP fields.
8 (2-byte SAP follows)	The 2-bytes following the 6-byte physical address is used for DIX Ethernet SAP. For 802, the 2-bytes are used as the DSAP and the stream's bound SAP is used as the SSAP.
13 (5-byte SNP header and 2-byte SAP follows)	The assumption is that this packet uses 802 SNAP encapsulation. Bytes 12 and 13 are treated as the DSAP, and bytes 9-11 are the SNAP header. Bytes 6 and 7, which are presumably both 0xAA, are ignored (0xAA is used for both SSAP and DSAP).

4.15 XID and TEST Primitives.

The common code provides complete support for both XID and TEST primitives. The semantics are as described in Section 4.4 of the DLPI 2.0 Specification. The various LLC information, SAP values and packet formats are as specified by IEEE 802.2[1]. Note that the "auto response" flags in the dl_xidtest_flg field of DL_BIND_REQ impact the operation of these primitives. In particular, if automatic handling has been set, these primitives are never used.

1. Information processing systems—Local area networks—Part 2: Logical link control, ISO 8802.2: 1989, ANSI/IEEE Std 802.2-1989, IEEE, New York, 1989.

5 Driver *ioctl*s

The common code recognizes three M_IOCTL messages as summarized in the following table.

Table 7: Supported I_STR *ioctl*s

Command	Description
DL_IOC_HDR_INFO	This is the Fast Path *ioctl*; see Section 6.
I_OTSetFraming-Type	If the unsigned long argument is kOTFraming8022, then dcl_mac_type in subsequent DL_INFO_ACK messages will be set to DL_CSMACD. For any other argument, DL_ETHER is set. The *ioctl* return value is the framing type specified in the argument.
I_OTSetRawMode	Enable special network monitoring extensions; see Section 7.

These M_IOCTLs may be created directly by upper-level modules or as I_STR *ioctl*s from the application. These may not be issued as transparent *ioctl*s.

6 Mentat Fast Path Extension to DLPI

Mentat has for many years utilized an extension to DLPI in its protocols and drivers. This extension, called Fast Path, substantially reduces the processing overhead for data packets being transmitted over a virtual circuit, e.g., an established TCP connection. Fast Path support in the common code is completely transparent to the hardware code.

As summarized in Section 1, the basic idea of Fast Path is to create the necessary frame header for packets only once. To do this, the upper level module sends an M_IOCTL message with ioc_cmd == DL_IOC_HDR_INFO. The M_DATA message block attached to this *ioctl* contains a complete DL_UNITDATA_REQ identical to what the upper layer would supply for data packets (Figure 3a).

The common code processes this *ioctl* and returns an M_IOCACK message containing the Ethernet Frame Header corresponding to the destination address in the DL_UNITDATA_REQ. This header is returned in a third message block as shown in Figure 3b. The DL_UNITDATA_REQ in the second block is unchanged; it is returned to simplify the task of re-establishing the sender's context.

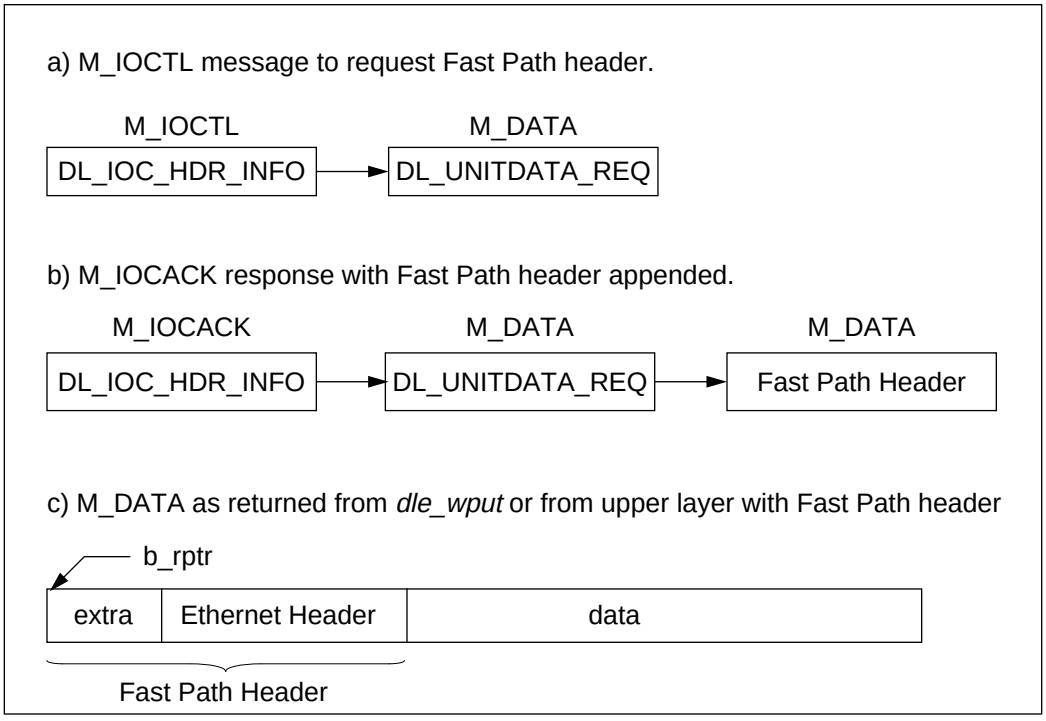

Figure 3: Fast Path Message Formats

The b_rptr for the third M_DATA points xtra_hdr_len bytes in front of the Ethernet Header, where xtra_hdr_len is the second argument to *dle_init* (Section 2).

After Fast Path negotiation, upper layer modules send data in M_DATA messages with the layout shown in Figure 3c. Everything between b_rptr and b_wptr in the returned Fast Path header (second M_DATA in Figure 3b) must be inserted at the front of subsequent M_DATA messages sent to the driver. Protocols using Fast Path must ensure that the Fast Path Header portion of all M_DATAs sent to the driver is "writable". The hardware code is free to modify any part of the Fast Path Header; in fact for 802 framing, it must insert the length of the data into the Length/Type field.

If xtra_hdr_len is nonzero, the hardware code must perform an additional test before processing M_DATA messages. Drivers must support a "raw mode" in which complete Ethernet packets are sent from an upper layer with no knowledge of the xtra_hdr_len bytes. The test involving the dcl_flags field of the dcl_t structure which is at the start of the driver's instance data (see Section 1.2) is shown in the following code fragment:

```
dp = (dcl_t *)q->q_ptr;
...
/* driver write-side put procedure M_DATA processing */
case M_DATA:
       if (dp->dcl.dcl_flags & F_DCL_RAWMODE_OK) {
              /* raw mode - mp->b_rptr at Ethernet frame */
              ...
       } else {
              /* Fast Path - &mp->b_rptr[xtra_hdr_len] is
               * start of Ethernet frame.
               */
              ...
       }
```

7 Network Monitor Extensions

The common code provides an extension to standard DLPI semantics which permits sending and receiving packets containing errors, e.g., CRC errors. Since such capabilities are a requirement of network monitoring software, we refer to them as Network Monitor Extensions.

Support of these features is dependent upon the hardware code, since the underlying hardware must have a mechanism for transmitting and receiving corrupt packets. Because of the hardware requirements and because support for network monitors may not be a requirement of many drivers, this feature is completely optional and hardware code may simply choose not to implement the necessary support code.

To support the network monitor extensions, the hardware code must supply a *dlehw_send_error* function if it will support sending of corrupt packets. It must set dlehw_recv_error_flags and call *dle_inbound_error* to deliver received corrupt packets to registered streams. See Section 2 and Section 3 for details on these functions. The hardware code may support either, both, or neither of these features.

7.1 Raw Mode *ioctl*

Network monitor extensions are accessed by the I_OTSetRawMode *ioctl*. This *ioctl* has two subcommands, one to control which packets to receive and one to send packets which deliberately contain errors. The structures and definitions for this *ioctl* are contained in dlpiuser.h. The subcommands and related structures are summarized in the following table.

Subcommand	*ioctl* Structure	Description
kOTSetRecvMode	dl_recv_control_t	Enable receipt of raw mode packets, either normal or both normal and with errors.
kOTSendErrorPacket	dl_send_control_t	Request the hardware to send a packet with specified errors, e.g., CRC or short packet.

In both cases, the *ioctl* structure defines the subcommand and arguments to it.

7.2 Receiving Raw Packets

In order for a stream to receive raw packets that include network headers, an I_OTSetRawMode *ioctl* with a kOTSetRecvMode subcommand must be issued on that stream. The ic_dp field of the strioctl structure passed as argument to the I_STR *ioctl* points at a buffer containing a dl_recv_control_t structure:

```
typedef struct dl_recv_control_s {
       unsigned long  dl_primitive;
       unsigned long  dl_flags;
       unsigned long  dl_truncation_length;
} dl_recv_control_t;
```

The dl_primitive field must be set to kOTSetRecvMode.

Once raw mode is set, only packets of the requested type(s), as specified in the dl_flags field, will be delivered upstream. Each packet will be in an M_DATA message with a dl_recv_status_t structure inserted in front of the packet header.

```
typedef struct dl_recv_status_s {
       unsigned long  dl_overall_length;
       unsigned long  dl_flags;
       unsigned long  dl_packet_length_before_truncation;
       unsigned long  dl_pad;
       OTTimeStamp    dl_timestamp;
} dl_recv_status_t;
```

On success, the return value from the OTSetRawMode *ioctl* is the size of this dl_recv_status_t structure, i.e., the offset to the start of the packet data.

The dl_flags field of the dl_recv_control_t structure is used to specify the type(s) of packets to receive. The value passed in the *ioctl* is a bit-mask formed by OR'ing together the desired values from Table 8.

Table 8: Selecting Raw Mode Packets

dl_flags Value	Description of Packets Received
DL_NORMAL_STATUS	If DL_NORMAL_STATUS is set, then every properly formatted packet will be passed upstream with a dl_recv_status_t structure inserted before the actual packet data. The packet data will include the complete network header, but will not include the CRC value. DL_NORMAL_STATUS mode may be set on any DLPI stream, whether or not the stream is bound to a particular SAP or whether it is enabled for promiscuous mode.
DL_ERROR_STATUS	If DL_ERROR_STATUS is set, then corrupt packets will be passed upstream with a dl_recv_status_t structure inserted before the actual packet. The types of error packets available (Table 9) will be indicated by the dl_flags return value. This mode is only valid for streams that are enabled for DL_PROMISC_PHYS operation. The mode may be set for any stream, but only promiscuous streams will actually receive error packets. **If DL_ERROR_STATUS is set, then DL_NORMAL_STATUS must also be set**.
DL_TRUNCATED_ PACKET	If DL_TRUNCATED_PACKET is set, then all messages passed upstream with "receive status" will be truncated to either dl_truncation_length or the maximum size of packets for the board, whichever is smaller. DL_TRUNCATED_PACKET is only meaningful in conjunction with DL_NORMAL_STATUS and DL_ERROR_STATUS. This flag value does not affect regular DL_UNITDATA_IND or Fast Path M_DATA messages.

On successful return from the *ioctl*, dl_flags will be set to show the types of packets the driver is able to return. In particular, if error packets have been requested (DL_ERROR_STATUS was set in the *ioctl*), the return value will include one or more additional bit values from Table 9. This feature relies heavily on the capabilities of the underlying hardware; not all drivers will be able to deliver all types of corrupted packets. The hardware code sets the driver's capabilities in the dlehw_recv_error_flags field in the board's dle_t structure (see Section 3).

Table 9: dl_flags Return Values

dl_flags Value	Description
DL_CRC_ERROR	CRC error packets are returned.
DL_RUNT_ERROR	Short packets (< 64 bytes) are returned.
DL_FRAMING_ERROR	Packets with Ethernet framing errors are returned.
DL_BAD_802_3_ LENGTH	802 packets whose Length field doesn't match the actual packet length are returned. These error packets are detected by the common code and are always available, i.e., if DL_ERROR_STATUS is set, this bit will always be set in the return value.

If DL_TRUNCATED_PACKET is set in the *ioctl*, dl_truncation_length includes both the dl_recv_status_t structure and the packet data. For example, if dl_truncation_length is set to 64, then packets passed upstream will contain 24 bytes (==sizeof(dl_recv_status_t)) of status information, followed by 40 bytes of packet data.

Raw Mode Packet Format.

For each raw mode packet sent upstream, the dl_recv_status_t structure in front of the actual packet identifies the packet characteristics. The dl_flags field (of the dl_recv_status_t structure) indicates the packet type, DL_NORMAL_STATUS or DL_ERROR_STATUS. If DL_ERROR_STATUS is set, additional bits from Table 9 will included to identify the error.

If DL_TRUNCATED_PACKET was set in the I_OTSetRawMode *ioctl*, then it will also be set in each dl_recv_status_t dl_flags field, and dl_packet_ length_before_truncation will indicate the original size of the packet.

Whether or not the message is truncated, the dl_overall_length field contains the number of bytes in the full message passed upstream, including the size of the dl_recv_status_t structure.The packet, truncated if requested, follows the dl_recv_status_t structure, and the message containing them is padded to an 8-byte boundary. This format allows application software to read multiple messages in one read request and to know where the packet boundaries are located. (Recall that a STREAMS *read* operation will pull multiple M_DATA messages from the Stream head read-side queue until the *read* request is satisfied.)

The dl_timestamp field is set with *OTGetTimeStamp*. Note that the message is typically created at interrupt time, so this time stamp is as accurate as we can get.

7.3 Sending Raw packets

Raw packets may be sent using either

- the I_OTSetRawMode *ioctl* with subcommand kOTSendError-Packet, or
- simply by passing a plain M_DATA message downstream.

If raw packets will be sent on a stream, then Fast Path must not be used on the same stream, i.e., no DL_IOC_HDR_INFO *ioctl* can be issued on that stream.[1]

To send normal packets, it is simplest to use M_DATA messages. In this case, the driver hardware code will transmit the packet without modification, i.e., exactly as it is formatted in the message. The driver assumes that the full network header is already attached to the message. The destination address will not be changed. The source address may be updated to the board's local address if this is done automatically by the underlying hardware. If the length field of an 802.3 header is 0, then the driver will insert the actual length of the packet in this field. Packets smaller than the minimum size will be padded with zeros (see Section 9 for hardware code requirements for processing M_DATA messages).

To send error packets, the I_OTSetRawMode *ioctl* with subcommand kOTSendErrorPacket must be used. The ic_dp field of the strioctl structure passed as argument to the I_STR *ioctl* points at a buffer containing the full packet to transmit preceded by a dl_send_control_t structure:

```
typedef struct dl_send_control_s {
       unsigned long  dl_primitive;
       unsigned long  dl_flags;
} dl_send_control_t;
```

The dl_flags field may contain any combination of values from Table 9. Which, if any, of these are supported depends upon the hardware code's *dlehw_send_error* function. If this function is not provided, or a request is made for an error type the driver cannot transmit, the *ioctl* will fail with EINVAL. However DL_BAD_802_3_LENGTH is always supported since this is handled by the common code.

If DL_CRC_ERROR is set in dl_flags, then the packet data is assumed to contain the CRC value as the last four bytes. For other types of error transmits, the CRC value will be calculated and appended to the data as for normal packets.

---

1. Actually this restriction applies only to drivers for which xtra_hdr_len argument to *dle_init* is nonzero.

The length of the transmitted packet will be exactly the number of bytes passed as the *ioctl* data, minus the length of the dl_send_control_t structure. This length may be smaller or bigger than legal sizes. The destination and source addresses will not be changed, unless the hardware updates the source address automatically. The 802.3 length field in the header will not be updated, even if the value passed is 0.

8 MIB Statistics

The common code defines two structures for maintaining interface statistics. These structures, dle_interface_status_t and dle_ethernet_status_t are defined and extensively commented in dlpiuser.h. The fields in these structures correspond to Interface MIB statistics (RFC 1573) and Ethernet MIB statistics (RFC 1643), respectively.

These two structures are embedded in the dle_t structure as dle_istatus and dle_estatus. The common code maintains only the receive statistics in dle_istatus. Since Fast Path packets are not passed to the common code, the common code cannot update most of the transmit and Ethernet statistics. Hence these statistics must be maintained by the hardware code. Example code is given in loopback.c.

The DL_GET_STATISTICS_REQ is used to retrieve these structures. The format of the returned DL_GET_STATISTICS_ACK is:

```
TOptionHeader structure
dle_interface_status_t structure
TOptionHeader structure
dle_ethernet_status_t structure
```

The value of length in the TOptionHeader structure is the length of the TOptionHeader structure plus the length of the status structure that follows. The level is DLPI_XTI_LEVEL; the option name is one of DL_INTERFACE_MIB or DL_ETHERNET_MIB.

9 Writing a Driver Using the Template

This section describes how to use the DLPI Driver Template to write a STREAMS device driver for Apple's Open Transport. As a starting point, "hardware code" for a fully operational loopback driver is provided in the file loopback.c. This file contains all the necessary STREAMS interface code and provides detailed comments on what needs to be changed for a hardware device driver.

9.1 Checklist for the Hardware Code

Most of the information for writing the hardware code has been presented in previous sections. The following paragraphs summarize the key points.

- **Additions to dle_t and dcl_t.**

  The board_t and loop_t structures for the loopback driver require no fields in addition to those of the dle_t and dcl_t. Most drivers will need to add board-specific fields to the board_t structure. However, additions to the streams instance data, loop_t, are generally not required.

  Table 10, a subset of Table 2, contains those fields in the dle_t structure which must be maintained by the hardware code.

Table 10: dle_t Fields Maintained by Hardware Code

Field	Description
dle_current_addr[6]	The boards current Ethernet address is stored here. This field is initially the same as dle_factory_addr. This is the address used by the common code.
dle_estatus	Hardware code must increment event counters defined in this structure.
dle_factory_addr[6]	The board's factory-set address.
dle_hw	This structure defines the four hardware entry points. See Section 3 for details.
dle_istatus	The hardware code must maintain the transmit statistics in this structure. See also description of *dle_init* in Section 2.
dle_xtra_hdr_len	This field is set from the value passed to *dle_init.*

- **streamtab and install_info Structures.**

  Synchronization level in the install_info structure must be SQLVL_MODULE. Other fields are set as described in the Open Transport documentation.

- **Modify Installation Functions.**

  Three functions, *GetOTInstallInfo*, *InitStreamModule*, and *TerminateStreamModule* may require additional board-specific code. Comments in these functions give suggestions.

- **Modify STREAMS open and close routines.**

  The loopback driver open and close routines simply call *dle_open* and *dle_close*, respectively. Most of the board-specific initialization is done when the driver is loaded by Open Transport and *InitStreamModule* is called. If additional per-stream data is required, the necessary fields can be added to the instance data structure (loop_t for the loopback driver) and initialized at stream open time.

  If there are multiple board_t structures for your driver, the associated board_t structure must be determined before calling *dle_open*. Open Transport functions such as *OTFindPortByDev* may be used to determine this association.

- **Hardware Start, Stop, Reset Functions.**

  These functions have been described in Section 3. The loopback driver versions of these functions contain extensive comments that show a typical implementation.

- **Transmit Code and Write-side Put and Service Procedures.**

  All message types except M_DATA are passed to *dle_wput* for processing. Data to be transmitted will either arrive in write-side put procedure (*loop_wput* in the loopback driver) as M_DATAs or will be returned from *dle_wput* as an M_DATA. See Section 2 for details on calling *dle_wput*, and Section 6 for Fast Path.

  For 802 packets, the hardware code must compute the length of the data and insert it into the Length/Type field of the Ethernet frame header. The loopback driver provides sample code that efficiently computes the length while copying data from mblk's into hardware transmit buffers.

  The hardware code must provide for STREAMS flow control. STREAMS drivers exert flow control placing messages on the write-side service queue (q_first field is the head), so that the total length of all queued messages is greater than q_hiwat. By removing messages with *getq*, the upstream flow control point (the first upstream queue with a service routine) will be back-enabled (the service routine will run) when the total amount of queued data falls below q_lowat.

  The choice on how much data to queue, whether or not a service routine should be used, etc., depends upon the hardware's capabilities:

  – Can the board transmit directly from message blocks or must the data be copied to transmit buffers?
  – How much data can be queued on the board?

– Can the board transmit multiple buffers without driver intervention?

Whatever design choice is made, it is important that the transmit interrupt logic and enabling of any service routine overlap to minimize latency.

- **Receive Code and Read-side Service Routine.**

  The receive interrupt handler must be written. The basic logic is shown in template form in the *board_intr* function of the loopback driver. Essentially all that is required is to create a STREAMS M_DATA message containing the received data and pass it to *dle_inbound* for processing.

  dle_inbound will call *putq* for each stream (queue) that must receive a copy of the message. After return from interrupt, the read-side service routine of all queues with data will be called. The service routine must be supplied in the hardware code (no STREAMS entry points are included in the common code). The loopback driver's *loop_rsrv* routine can be used without change.

- **Add Statistics Gathering code.**

  The interface transmit statistics and the Ethernet statistics must be maintained by the hardware code. Consult dlpiuser.h to determine which event counters are required.

9.2 Adding Support for Other DLPI Primitives and *ioctl*s

There are a few DLPI primitives for Local Management (Section 4.1 of the DLPI Specification) and Connectionless-mode (Section 4.3) which are not supported by the common code. DL_ATTACH_REQ is not supported because Open Transport does not support Style 2 DLPI Drivers. Two other primitives, DL_UDQOS_REQ and DL_SET_PHYS_ADDR_REQ, are not currently supported by Open Transport protocols and have not been implemented in the common code. Nonetheless, the design of the common code makes it fairly easy for the hardware code to add support for these or any other DLPI primitive the driver might require.

To support additional primitives, the hardware code must process M_PROTO messages for the added DLPI primitive **before** calling *dle_wput*. If the primitive matches an added primitive, *dle_wput* is not called. Otherwise the message is just passed to *dle_wput*.

Driver-specific *ioctl*s are most easily processed **after** calling *dle_wput*, which will return an M_IOCNAK for any unrecognized *ioctl*. Since only the message type and ioc_error fields have changed, the hardware code can

examine the ioc_cmd field and trailing M_DATA messages and take appropriate action. Comments in the loopback driver show how to do this.

9.3 List of Open Modules

A list of driver instances currently open is maintained by the common code. This list is maintained on a per board basis; the head of the list is dle_instance_head in each board's dle_t structure. The kernel utility function *mi_next_ptr* must be used to traverse this list.

---

mi_next_ptr

This function is used to traverse the linked list of open module instances.

Prototype

```
char *
mi_next_ptr (
                char            * ptr );
```

Argument	Description
ptr	Pointer to the instance data (q_ptr field) for which the "next" instance is desired.

Return Value

Returns a pointer to the instance data of the next module instance in the linked list. Returns NULL if ptr is from the last instance in the list.

Notes

1. Assuming the instance data is of type dcl_t, the list head variable is the dle_instance_head field of the dle_t structure dle, and dclp is of type dcl_t *, then list traversal using this function takes the following form:

```
for (dclp = (dcl_t *)dle.dle_instance_head
; dclp
; dclp = (dcl_t *)mi_next_ptr((char *)dclp) ) {

        ...
}
```

2. This function may not be called from interrupt context.