Chapter 10 261

10 CKO and Transport IOCTLs

This chapter explains the interaction between the HP-UX transport layer and the DLS providers to create an efficient data transfer mechanism between layers in a networking stack. The HP-UX transport layer supports DLS providers that adhere to DLPI 2.0 standard. Certain, HP specific extensions to DLPI 2.0 standard must also implemented when interaction with the HP-UX transport layer.
DLPI IOCTL

The HP-UX transport stack uses a set of IOCTLs for the in-kernel STREAMS Data Link Service (DLS) user (e.g., IP module) to negotiate the driver features and set up a Fastpath. These IOCTLs are HP-UX specific and were originally introduced in HP-UX 11.0. These IOCTLs are listed and described.

DLPI_IOC_DRIVER_OPTIONS

The DLPI_IOC_DRIVER_OPTIONS IOCTL is used by IP to obtain additional information from a DLS provider regarding the driver's capabilities. The DLS provider communicates to IP of the capabilities which IP may use to enhance performance.

While configuring an interface for IP, the IP sends an M_IOCTL message to the DLS provider. The M_IOCTL message block contains an iocblk structure with ioc_cmd == DLPI_IOC_DRIVER_OPTIONS, ioc_count == size of (driver_ops_t). Following the M_IOCTL message block, pointed by b_cont of the M_IOCTL message block is a M_DATA message block containing a driver_ops_t structure.

For information on the structure, refer to Table 10-1, “driver_ops_t Data Structure.”

Table 10-1  
driver_ops_t Data Structure

<table>
<thead>
<tr>
<th>Name</th>
<th>driver_ops_t</th>
</tr>
</thead>
<tbody>
<tr>
<td>Include File</td>
<td>&lt;sys/dlpi_ext.h&gt;</td>
</tr>
<tr>
<td>Purpose</td>
<td>DLPI/XPORT options negotiations structure. This structure will be passed as part of the DLPI_IOC_DRIVER_OPTIONS IOCTL from XPORT.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Members</th>
<th>Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>driver_ops_type</td>
<td>uint32_t</td>
<td>Features supported by XPORT. The features are bit wise flags.</td>
</tr>
<tr>
<td></td>
<td>driver_ops_type_1</td>
<td>uint32_t</td>
<td>Reserved. Must be set to “0”</td>
</tr>
<tr>
<td></td>
<td>driver_ops_type_2</td>
<td>uint32_t</td>
<td>Reserved. Must be set to “0”</td>
</tr>
</tbody>
</table>

For information on possible features supported by transport layer, refer to Table 10-2, “Capabilities Understood by IP.”
The `driver_ops_type` field is used by the IP module to pass a bit mask of flags, indicating the driver capabilities that IP is inquiring about. The driver is expected to look at this bit mask, perform whatever action it needs to enable the options asked for, and then to return an `M_IOCACK` containing the same data structure with `driver_ops_type` set to a bit mask showing the options the driver supports. The transport option `DRIVER_PRI` was added to enable VLAN. Currently, it is not supported for all non-HP network interface drivers, therefore all non-HP drivers must turn this bit off before replying to IP.

The following code snippet was taken from the ENET sample driver for HP-UX version 11i v1. It shows how network drivers must respond to the `DLPI_IOC_DRIVER_OPTIONS` IOCTLs. Note that transport options may be added to HP-UX at any time and therefore recommended that IHV developers make modifications to their network drivers such that it behaves as the following code. This code turns off all the options flags at the beginning. Only the options supported by the driver are turned on.

<< enet code snippet here >>
**DL_IOC_HDR_INFO**

A DLS provider normally expects to receive datagrams in the form of a `DL_UNITDATA_REQ` message. The `DL_UNITDATA_REQ` primitive contains the hardware source and destination addresses from which a DLS provider can construct a packet header and place in front of the IP packet before transmitting the packet. The creation of the packet header is not performance tuned as it has to be constructed for every packet that is sent out. Many connection-oriented protocols like TCP, once the connection has been established, will have identical information in every `DL_UNITDATA_REQ`. To avoid constructing a packet header every time, a DLS provider can support an IOCTL which permits the transport layer like TCP/IP to request a packet header template during configuration of TCP/IP on a per-interface basis. The packet header template will enable transport layer to add the packet header before sending the packet to a DLS provider. This mechanism is called **Fastpath**.

The negotiation of **Fastpath** happens when the transport layer sends a `M_IOCTL` type message with `ioc_cmd` set to `DLPI_IOC_HDR_INFO`. The `b_cont` of the `M_IOCTL` message block will link a `M_DATA` type message block which contains `dl_unitdata_req_t` structure which is used in the creation of `DL_UNITDATA_REQ` primitive.

For a driver that does not support Fastpath capability, a `M_IOCNAK` message should be sent back, in which case the transport layer will use only the `DL_UNITDATA_REQ` primitive for data transfer.

A Native STREAMS DLPI network interface driver must implement this IOCTL, if it intends to support the Fastpath capability. Refer to Figure 10-1, “Fastpath,” for Fastpath negotiation message formats.

**Figure 10-1   Fastpath**

![Diagram of Fastpath](image)

In response to the `DL_IOC_HDR_INFO` IOCTL, a DLS provider must return the packet header template which contains the CKO header and the Link level header template with information filled in. The response will be a `M_IOCACK` IOCTL as shown in Figure 10-1, “Fastpath.” The CKO header must be present only if the DLS provider supports the Checksum Offload feature. The second message block contains the `DL_UNITDATA_REQ` passed to DLS provider. The third message contains a `M_DATA` message block with the **LLC Header** template created for that particular stream.

The CKO header part of the packet header template is used by the transport layer to communicate checksum offload related information.
Check offload is a host assistance feature provided by a Network Interface Card (NIC). The NIC accelerates checksum generation and verification by performing this function for the host CPU (offloading it). This network card feature allows checksum calculation (including pseudo headers) for IP, TCP and UDP packets. Enabling checksum offload on adapters leads to a reduction in host CPU utilization and improved performance due to reduced cache misses (avoiding cache pollution with the data to be checksummed). The network card can also support checksum offload of fragmented packets; when, on the transmit side, the host sets up the buffers in such a way that fragmented frames can have their UDP or TCP checksums calculated over the entire datagram.

Three support modules are required for the host to exploit the NIC checksum offload feature:

1. Hardware: The NIC should have the feature to calculate checksum for a TCP/UDP/IP packet.
2. Driver: When checksum offload is enabled, it indicates to the upper layer that it is no longer required to calculate the checksum and places the necessary checksum calculating information in the NIC.
3. Transport Stack: Provides the checksum information (explained later) to the driver.

Adding Checksum Offload Support

There are two kinds of checksums that can be calculated by the NIC:

1. TCP/UDP checksum
2. IP header checksum

The NIC must provide both TCP and UDP checksum offload support. TCP/UDP checksum can be calculated with or without a pseudo header. Therefore, the NIC enables the driver to indicate which checksums need to be calculated and if a TCP/UDP checksum should be calculated with or without a pseudo header.

These options can be set differently for transmit and receive packets.

The behavior of HP-UX transport stack for checksum offload:

- Checksum offload is supported only in IP fast path.
- Transport stack cannot take advantage of IP header checksumming because the IP layer calculates the header checksum for all adapters.
- Transport stack calculates the pseudo-header checksum by default and places it in the TCP/UDP checksum field on outbound. It would be good if the adapter just used this field and did not compute the pseudo-header checksum again.

Initialization

Indication that the checksum offload feature is enabled is kept in a per instance structure, i.e., hwift structure. The DRV_CKO (0x8) flag is set in the features field of the hwift structure.

After binding the stream, the upper layer sends a DLPI_IOC_DRIVER_OPTIONS IOCTL request to the driver to check if the device instance supports checksum offload. The driver_dlpi_ioctl() function (driver routine which handles IOCTL requests) checks the features field of hwift and ACKs the request if DRV_CKO is set.
driver_dpli_ioctl()

Checks if the DRV_CKO flag is set in the features field of the hwift structure and returns back the driver_ops_t structure with DRVCKO flag set or cleared depending on whether checksum offload is enabled or not. This function must implement the DLPI_IOC_DRIVER_OPTIONS IOCTL.

Transmit Path

The NIC calculates the IP, TCP, and/or UDP checksum and inserts it into the outgoing frame. The NIC must provide a method for the driver to inform the NIC on a per frame basis. These flags can be provided in a transmit descriptor, where each transmit descriptor refers to a frame. The NIC can also provide the feature of calculating the TCP/UDP checksum for an IP fragmented packet by accumulating the checksums of each IP fragment and inserting the final checksum into the TCP/UDP header. For checksum calculation of fragmented packets, the NIC must provide extra bits to be set in the descriptor to mark the fragments to be considered and the end of the fragment etc. Also there would be a condition that all fragments must be placed in send ring consecutively and in the correct order.

For drivers with DRV_CKO enabled, the upper layer passes the Message Blocks (mblks) to the driver with the MSGCKO flag set in the b_flags field; this indicates that the NIC must calculate checksum for the packet. An IP fragment chain comes down to the driver as a series of mblks connected through the b_cont field with MSGCKO bit being set in the b_flag field of each mblk which is the start of the fragment.

For the mblks with the MSGCKO bit flag set, the first 8 bytes denote the Checksum Offload (CKO) information. So the driver_build_hdr() function (driver routine which builds the MAC header) builds the MAC header with an extra 8 bytes at the front of the MAC header before returning the header to the upper layer in fast path. Checksum offload structure (8 bytes) which comes from the upper layer contains the following information:

- Starting location (of the data to be checksummed)
- End location (pointing to the last byte of data)
- Seed information (pseudo-header checksum)
- Insert location (pointing to the location where the checksum is to be inserted)
- Insert flag (indicating whether the checksum is to be inserted in this fragment or not).

Checksum offload information (8 bytes) structure is defined in the header file <cko.h>.

```
struct cko_info {
    u_char    cko_type;   /* Checksum assist control field */
#define CKO_INSERT       0x0001  /* cko_seed value need to be used */
#define CKO_OUTBOUND     0x0002
#define CKO_CMD          0x0004
#define CKO_ALGO_NONE    0x0008
#define CKO_ALGO_UDP     0x0010   /* UDP packet */
#define CKO_ALGO_TCP     0x0020   /* TCP packet */
#define CKO_SUM_FLUSH    0x0040  /* Flush checksum in card */
#define CKO_ETC          0x0080 /* Embedded Trailer Cksum pkt */
    u_char    cko_start;  /* Checksum starting offset */
    u_short   cko_stop;   /* Checksum stop offset */
    u_short   cko_insert; /* Checksum insert offset */
    u_short   cko_seed;   /* Checksum seed value */
};
```
The driver analyzes the previous checksum offload information and saves the required information in a transmit descriptor of the packets in device specific manner. Then the driver strips off the first 8 bytes of each mblk with MSGCKO set before passing the packet to the NIC.

The following sections are some of the driver functions in the transmit path and the various purposes they fulfill.

**driver_wput()**

This function is the driver's write side put routine, which receives all the requests from DLPI user (upper layer). For Fastpath, this function calls `driver_dlpi_cko_fast_out()` if checksum offload is enabled (features &DRV_CKO), otherwise it calls `driver_if_resolved_output()`.

**driver_dlpi_build_hdr()**

This function builds the MAC header and if DRV_CKO is enabled it also allocates an extra 8 bytes in front of the MAC header. For fast path, this function is called when the upper layer sends an IOCTL request DLPI_IOC_HDR_INFO during the fast path setup. For regular path, this function is called by the driver to setup MAC header before sending a packet.

**driver_dlpi_cko_fast_out()**

This function is called by `driver_wput()` for fast path and also for an interface for which checksum offload is enabled. The following steps explain what the function does related to checksum offload:

1. IP fragment chain comes down to the driver as a series of mblks connected through the b_cont field with the MSGCKO bit set in the b_flag field of each mblk that is the start of the fragment. Every IP fragment is separated in such a way that the b_next field of the first mblk of each fragment points to the first mblk of the next fragment as shown Figure 10-2, “Fragments.”

2. For the message blocks which have MSGCKO set, the function checks if the CKO_INSERT bit is tuned on in the insert flag of the checksum information. If the CKO_INSERT bit is on, then it inserts the seed value (present in ckoi_info) which is the pseudo header checksum at the location pointed by cko_insert.

3. Calls `driver_if_resolved_output()`.
**Figure 10-2  Fragments**

![diagram](image)

**driver_dlpi_unitdata_out()**

This function is called by `driver_wput()` for all regular path data packets (DL_UNITDATA_REQ):

1. Calls `driver_dlpi_build_hdr()` to build the MAC header which allocates an extra 8 bytes at the front of the MAC header if `DRV_CKO` is enabled.
2. Calls `driver_if_resolved_output()`.

**driver_if_resolved_output()**

This function is called for both regular path and fast path packets:

1. Strips off checksum information (8 bytes) from all message blocks.
2. For any packets with the MSGCKO bit set, it saves the `cko_start` and `cko_insert` fields of the `cko_info` structure for later use.
3. For any packets with the MSGCKO bit set, it calculates the frame length using the `cko_stop` field of the `cko_info` structure.
4. Calls `driver_hw_req` to transmit packets.

**Receive Path**

The NIC calculates the IP, TCP and/or UDP checksum for each incoming frame. The calculated checksum can be kept in a receive descriptor from which the driver reads the values. Transport stack can use the TCP/UDP checksum value directly for unfragmented packets. For fragmented packets, the stack needs to add up the TCP/UDP checksums for all the fragments to obtain the final checksum.
The driver reads the TCP/UDP checksum value from the device registers or receive descriptor (depending on the NIC) and saves the value in \texttt{b_quad[3]} of mblk for which checksum corresponds. Also MSGCKO is set in the "\texttt{b\_flags}" of the mblk. Since the IP layer always calculates the IP header checksum, the IP header checksum calculation is not enabled on the device.

If \((\texttt{mblkp->b\_rptr} - \texttt{mblkp->b\_datap->db\_base}) > 8\). In other words, if there are 8 bytes of space between \texttt{db\_base} and \texttt{b\_rptr}, then the driver moves \texttt{b\_rptr} back by 8 bytes to cover checksum information (structure \texttt{cko\_info}). Otherwise, the driver allocates a new mblk for checksum information (\texttt{cko\_info}). The driver then saves the checksum value (\texttt{b\_quad[3]}) in the \texttt{cko\_insert} field of the \texttt{cko\_info} structure (which will be read by the IP layer). For fragmented packets, the transport stack adds up the TCP/UDP checksums for all the fragments to obtain the final checksum.

The device calculates the checksum for each frame and saves it in the receive descriptor. The \texttt{driver\_receive\_pkts()} function reads the checksum value from receive descriptor and saves it in \texttt{b\_quad[3]} of the message block and sets the MSGCKO bit in the \texttt{b\_flags} field of the message block.

\textbf{driver\_dlpi\_fast\_in()}

Reads checksum value from \texttt{b\_quad[3]}.

If \((\texttt{mblkp->b\_rptr} - \texttt{mblkp->b\_datap->db\_base}) > 8\), then it moves \texttt{b\_rptr} back by 8 bytes to cover the checksum information (structure \texttt{cko\_info}). Otherwise, it allocates a new mblk for checksum information. Then it saves the checksum value (\texttt{b\_quad[3]}) in the \texttt{cko\_insert} field of \texttt{cko\_info} structure, which will be read by the IP layer.

\textbf{Packet Flow Diagram}

The following diagram shown in \textit{Figure 10-3, “Packet Flow Diagram,”} shows how the packet traverses the driver.
Figure 10-3  Packet Flow Diagram

lan_dlpikrn.h:

#define DRV_CKO 0x8 /* Indicates that particular interface has checksum offload enabled */

stream.h:

#define MSGCKO 0x2000 /* Checksum offload is expected for particular packet */

dlpi.h:

#define DL_CLDLS 0x02 /* support connectionless data link service */
#define DL_ETHER 0x4 /* Ethernet Bus, LLI Compatibility */
```c
#define DL_CSMACD 0x0 /* IEEE 802.3 CSMA/CD network */

if_ether.h:
#define ETHER_HLEN 14 /* header length for ethernet packets */
#define ETHER_PKT 0
#define SNAP8021_PKT 2
#define IEEE8021_PKT 5

if_eee.h:
#define MIN_IEEE8022_HLEN 3 /* minimum length for 802.2 header */
#define SNAP_802_2_HLEN 8 /* minimum length for 802.2 snap header */
#define CKO_INFO_SIZE 8

int driver_wput (queue_t *q, mblk_t *mblkp)
{
    /* Get the pointer to the per stream structure */
    driver_dlpi_data_t *hp_dlpi_datap = (driver_dlpi_data_t *)q->q_ptr;

    if (mblkp->b_datap->db_type == M_DATA) { /* Fast Path */
        if ((hp_dlpi_datap->service_mode == DL_CLDLS) &&
            hp_dlpi_datap->fast_path &&
            (hp_dlpi_datap->curr_state == DL_IDLE)) {
            if(hp_dlpi_datap->hwiftp->features & DRV_CKO) {
                . . .
                driver_dlpi_cko_fast_out (hp_dlpi_datap, mblkp);
                . . .
            } else {
                . . .
                driver_if_resolved_output (hp_dlpi_datap, mblkp);
                . . .
            }
        } else if (hp_dlpi_datap->service_mode == DL_CODLS) {
            /* connection oriented part */
            . . .
        } else {
            /* Error : free messages */
            . . .
        }
    }
}
```
/** The case of M_PROTO and UNITDATA request (Regular path) */
if ((mblkp->b_datap->db_type == M_PROTO) &&
    (*((uint32_t *)mblkp->b_rptr) == DL_UNITDATA_REQ)) {
    . . .
    driver_dlpi_unitdata_out (hp_dlpi_datap, mblkp);
    . . .
}

/* Handle other messages (M_IOCTL, M_PCPROTO etc) */
.
.
}

mblk_t *
driver_dlpi_cko_fast_out (driver_dlpi_data_t *hp_dlpi_datap, mblk_t *mblkp)
{
    hw_ift_t                *hwiftp = hp_dlpi_datap->hwiftp;
caddr_t                 drv_datap;
intptr_t                error;
mblk_t                  *finger_mblkp;
mblk_t                  *cur_frag_mblkp, *next_frag_mblkp;

    finger_mblkp = cur_frag_mblkp = mblkp;

    /* Break IP fragment chain */
    while ( (finger_mblkp) ) {
        if ( (finger_mblkp->b_cont && (finger_mblkp->b_cont->b_flag & MSGCKO)) ) {
            /* b_cont here is the beginning of next frag */
            next_frag_mblkp = finger_mblkp->b_cont;
            finger_mblkp->b_cont = NULL;

            /* formulate the train */
            cur_frag_mblkp->b_next = next_frag_mblkp;
            VASSERT(next_frag_mblkp->b_next == NULL);

            /* reset finger_mblkp to the head of the rest */
            finger_mblkp = cur_frag_mblkp = next_frag_mblkp;
            continue;
        }
        finger_mblkp = finger_mblkp->b_cont;
    }
    cur_frag_mblkp = mblkp;

    /* Insert pseudo header checksum into TCP/UDP checksum field in packet */
    while ( (cur_frag_mblkp) ) {
        if (!driver_dlpi_add_xport_cksum(hp_dlpi_datap, cur_frag_mblkp)) {
            . . .
        }
    }
}
/* free the entire stream message */
cur_frag_mblkp=mblkp;
while (cur_frag_mblkp) {
    next_frag_mblkp=cur_frag_mblkp->b_next;
    freemsg(cur_frag_mblkp);
    cur_frag_mblkp=next_frag_mblkp;
}
return driver_dlpi_error_ack(NULL, DL_UNITDATA_REQ, EINVAL, DL_SYSERR);

cur_frag_mblkp = cur_frag_mblkp->b_next;
}
.
.
driver_if_resolved_output(hp_dlpi_datap, mblkp);

/* Inserts pseudo header checksum into checksum field for TCP/UDP */

int

driver_dlpi_add_xport_cksum (driver_dlpi_data_t * hp_dlpi_datap, mblk_t *mblkp)
{

    struct cko_info       *cko_info_in, cp_cko_info_in;
    uint16_t              *cksum_addr;
    uint32_t              insert_len, cko_offset;
    mblk_t               *mblkp, *tmp_mblkp;

    mblkp=(mblk_t *)datap;
    cko_info_in = (struct cko_info *)mblkp->b_rptr;
    cs_block = (fddi_xmit_cs_t *)mblkp->b_rptr;

    if (cko_info_in->cko_type & CKO_INSERT) {
        tmp_mblkp = mblkp;
        /* We must adjust the insert length to include the llc header
         * because the insert_len calculated by the transport does
         * not include the llc header length.
         */
        insert_len = cko_info_in->cko_insert + hp_dlpi_datap->fast_path_llc_length;
        while (insert_len > 0) { /* Find the TCP header’s Checksum field. */
            if (insert_len < (tmp_mblkp->b_wptr - tmp_mblkp->b_rptr)) {
                cksum_addr = (uint16_t *)(tmp_mblkp->b_rptr + insert_len);
                insert_len = 0;
            }else {
                insert_len = insert_len - (tmp_mblkp->b_wptr - tmp_mblkp->b_rptr);
                tmp_mblkp = tmp_mblkp->b_cont;
            }
}  
    *cksum_addr = cko_info_in->cko_seed;
    }
  }
}

int  
driver_dlpi_build_hdr (driver_dlpi_data_t *hp_dlpi_datap, mblk_t *mblkp,  
                      uint8_t **llc_ptr)
{
    mblk_t                  *llc_hdr_mblkp, *temp = mblkp;
    caddr_t                 llc_hdrp, llc_hdr_datap;
    ...

    if (!((llc_hdr_mblkp = allocb(128, BPRI_HI)))
        return (0);  
    llc_hdrp = (caddr_t)llc_hdr_mblkp;
    if (hwiftp->features & DRV_CKO) {
        llc_hdr_datap = (caddr_t)llc_hdr_mblkp->b_rptr + 8; /* 8 bytes for cko_info */
    } else {
        llc_hdr_datap = (caddr_t)llc_hdr_mblkp->b_rptr;
    }

    /* Then create MAC header using  llc_hdr_datap */

    mblk_t  *  
driver_dlpi_unitdata_out (driver_dlpi_data_t *hp_dlpi_datap, mblk_t *mblkp)
{
    caddr_t                  llc_hdrp;
    mblk_t                  *llc_hdr_mblkp;
    ...

    /* driver_dlpi_build_hdr() is called to build MAC header */

    if(!driver_dlpi_build_hdr(hp_dlpi_datap, mblkp, (uint8_t **)llc_hdrp)) {
        if(llc_hdrp)
            freeb((mblk_t *)llc_hdrp);
        return hp_dlpi_uderror_ind(hp_dlpi_datap, mblkp, 0, DL_BADADDR);
    }
    llc_hdr_mblkp = (mblk_t *)llc_hdrp;
    linkb(llc_hdr_mblkp, mblkp->b_cont);
    ...
    driver_if_resolved_output(hp_dlpi_datap, llc_hdr_mblkp);
}
int
driver_if_resolved_output (driver_dlpi_data_t * hp_dlpi_datap, mblk_t *mblkp)
{
    register caddr_t       datap;
    uint32_t             buf_len;
    struct cko_info     *cko_info;
    uint32_t           fast_path_pkt_type, fast_path_llc_length;
    uint8_t              css;            /* Checksum start */
    uint8_t              cso;           /* Checksum insert offset */
...
    datap = (caddr_t) mblkp->b_rptr;
    buf_len = ((int)(mblkp->b_wptr) - (int)(mblkp->b_rptr));
    fast_path_pkt_type = hp_dlpi_datap->fast_path_pkt_type;
    fast_path_llc_length = hp_dlpi_datap->fast_path_llc_length;

    if(buf_len != NULL) {
        if(mblkp->b_flag & MSGCKO){
            cko_info = (struct cko_info *)(datap);
            /* Move datap by 8 bytes to strip off cko_info (8 bytes) */
            datap += CKO_INFO_SIZE;
            mblkp->b_rptr = (uint8_t *)(datap);
            buf_len -= CKO_INFO_SIZE;    /* decrease buf_len by 8 bytes */
            if (cko_info->cko_type & CKO_INSERT) {
                /* First fragment contains the TCP/UDP header where the checksum
                 * is to be inserted.
                 */
                cso = (int) (fast_path_llc_length + cko_info->cko_insert - CKO_INFO_SIZE);
                css  = (int) (fast_path_llc_length + cko_info->cko_start - CKO_INFO_SIZE);
            } else {
                insert_len = 0;
            }
            /* calculate packet length using cko_info->stop */
            if (buf_len >= ETHER_HLEN) {
                if (fast_path_pkt_type == ETHER_PKT) {
                    if (fast_path_pkt_type == ETHER_PKT) {
                        else {
                            uint16_t *llc_lengthhp;
                            llc_lengthhp = (uint16_t *)(datap + LAN_LLC_LEN_OFFSET);
                            if (fast_path_pkt_type == IEEE8023_PKT)
                                *llc_lengthhp = cko_info->cko_stop + MIN_IEEE8022_HLEN + 1;
                            else if (fast_path_pkt_type == SNAP8023_PKT)
                                *llc_lengthhp = cko_info->cko_stop + SNAP_802_2_HLEN + 1;
                        }
                    }
                }
            }
        }
    }
}
status = EINVAL;
cso = css = 0;

} else { /* non MSGCKO mblk */
datap += CKO_INFO_SIZE;
    mblkp->b_rptr = (uint8_t *)(datap);
    buf_len -= CKO_INFO_SIZE; /* decrease buf_len by 8 bytes */
} else {
    /* first mblk has a zero length buffer */
    status = EINVAL;
}

/* Use css & cso for filling transmit descriptor with checksum information */
...

driver_hw_req (hp_dlpi_datap->enetiftp, LAN_REQ_WRITE, mblkp);

...

void
driver_dlpi_fast_in (driver_dlpi_data_t *hp_dlpi_datap, mblk_t *mblkp, int pkt_fmt)
{
    uint8_t  *llc_hdrp;
    mblk_t   *tmp_mblkp;
    uint32_t llc_hdr_size = 0;
    queue_t *q;

    llc_hdrp = (uint8_t *) mblkp->b_rptr
    uint32_t llc_hdr_size = 0;

    if (pkt_fmt == ETHER_PKT) {
        llc_hdr_size = ETHER_HLEN;
    } else {
        /* Determine the LLC header size to strip off */
        switch (pkt_fmt) {
            case IEEE8023_PKT:
                llc_hdr_size = IEEE8023_HLEN;
                break;
            case SNAP8023_PKT:
                llc_hdr_size = SNAP8023_HLEN;
                break;
            default:
                /* Unsupported packet type received, drop it. */
                freemsg(mblkp);
                break;
        }
    }
}
return;
}

/* Strip off LLC header for both MSGCKO and non-MSGCKO packets */
while( llc_hdr_size) {
    if ( (mblkp->b_wptr - mblkp->b_rptr) > llc_hdr_size ) {
        mblkp->b_rptr += llc_hdr_size;
        llc_hdr_size = 0;
    } else {
        llc_hdr_size -= (uint32_t)(mblkp->b_wptr - mblkp->b_rptr);
        tmp_mblkp = mblkp;
        mblkp = mblkp->b_cont;
        if ( !mblkp ) { /* let go pkts with just llc hdrs*/
            mblkp = tmp_mblkp;
            mblkp->b_rptr = mblkp->b_wptr ;
            break;
        }
        freeb(tmp_mblkp);
    }
}

/* Use the LLC header left over above to store the CKO data!!
 * Driver must guarantee that the LLC header is contained in the same mblk.
 */
/* Determine if this packet contains checksum information. */
if ((mblkp->b_flag & MSGCKO) && (hp_dlpi_datap->hwiftp->features & DRV_CKO)) {
    mblk_t              *cko_mblkp;
    struct cko_info     *cko_info_in;
    uint16_t            check_sum;

    check_sum = (uint16_t)mblkp->b_quad[3];
    if ((mblkp->b_rptr - mblkp->b_datap->db_base) >= sizeof(struct cko_info)) {
        mblkp->b_rptr -= sizeof(struct cko_info);
        cko_info_in = (struct cko_info *)mblkp->b_rptr;
        cko_info_in->cko_insert = check_sum;
    } else {
        /* Allocate mblk to hold checksum information. */
        cko_mblkp = allocb(sizeof (struct cko_info), BPRI_HI);
        if (!cko_mblkp) {
            freeb(mblkp);
            return;
        }
    }
}
CKO and Transport IOCTLs

Checksum Offload

cko_mblkp->b_datap->db_type = M_DATA;
cko_info_in = (struct cko_info *)cko_mblkp->b_rptr;
cko_info_in->cko_insert = check_sum;

/* put the new cko_mblkp as the head mblk */
cko_mblkp->b_cont = mblkp;
cko_mblkp->b_flag |= MSGCKO;

/* MSGCKO should be set only at the beginning mblk */
   mblkp->b_flag &= ~MSGCKO;

   mblkp = cko_mblkp;
}
}
/*
 * Send message directly to upstream module.
 */
(void)putnext(hp_dlpi_datap->queue_ptr , mblkp);
return;
}
}

void
driver_dlp_process_ioctl (driver_dlpi_data_t *hp_dlpi_datap, mblk_t *mblkp)
{
  struct iocblk   *ioctl_mblkp;
  int             cmd, len = 0, retval;

  ioctl_mblkp = (struct iocblk *)mblkp->b_rptr;
  cmd = ioctl_mblkp->ioc_cmd;

  ...
  switch(cmd){
    case DLPI_IOC_DRIVER_OPTIONS:
      ops = (driver_ops_t *)mblkp->b_cont->b_rptr);
      if((ops->driver_ops_type & DRIVER_CKO) &&
                  !(hwiftp->features & DRV_CKO))
        ops->driver_ops_type &= ~DRIVER_CKO;
      ...
      qreply(q, mblkp);
      return;
    }
  }
}